Lukas Gressl, BSc

Design and Implementation of a Java Card Cross-Compilation Framework

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Engineering and Management

submitted to:

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Institute for Technical Informatics

Advisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Dipl.-Ing. Andreas Lessiak, NXP Semiconductors Austria GmbH

Graz, December 2016

Statutory Declaration

Date	Signature

Kurzfassung

Smart Cards und *Embedded Systems* im Allgemeinen werden für unsere heutige stark vernetzte Welt immer wichtiger. Für beinahe jeden Einsatzbereich, z.B. Finanztransaktionen, Identifizierung, etc., existiert bereits eine Vielzahl von speziellen Smart Card Applikationen. Da Smart Card Chips von Hersteller zu Hersteller unterschiedlich sind, müssten diese Applikationen für jede Chipart adaptiert und extra kompiliert werden. Daher wurde ein System angestrebt, das diese Applikationen von der Hardware unabhängig macht und somit auf unterschiedlichen Smart Cards ausgeführt werden kann. Durch die *Java Card Virtual Machine* und deren Laufzeitumgebung wird ein solches System realisiert, indem es den Code zur Laufzeit interpretiert. Jedoch wird durch diese Interpretation die Ausführungszeit der gesamten Applikation erhöht. Die Performance der Applikationen ist ein wichtiger Faktor in vielen Einsatzgebieten und daher versucht man alternative Methoden für deren Ausführung zu finden.

Das Ziel dieser Masterarbeit war, *Java Card* Applikationen vor derer eigentlichen Ausführung zu kompilieren, anstatt sie während der Laufzeit zu interpretieren. Nach einer Recherche ähnlicher Projekte wurde ein open-source Framework gefunden, mit dem *Java* Applikationen zu nativem Maschinencode kompiliert werden können. Dieses Framework übersetzt den byte code der Applikation in *C* Source Code und verwendet anschließend einen Standard-C Compiler zur weiteren Kompilierung. Dieses Framework wurde als Basis verwendet und so modifiziert, dass Klassen der *Java Card* Laufzeitumgebung sowie Applets nativ kompiliert werden können.

Der Übersetzungsprozess musste dahingehend verändert werden, dass das erzeugte Programm den Speicheranforderungen der angestrebten Hardwareumgebung entspricht. Des Weiteren musste der Lebenszyklus der Objekte an die Spezifizierung von *Java Card* angepasst werden und ein *Java Card* Betriebssystem implementiert werden. Die Korrektheit des Übersetzungsprozesses wurde durch eine spezielle Applikation, die die Eigenschaften der *Java Card* Programmiersprache testet, verifiziert. Zusätzlich wurde die Laufzeit der kompilierten Applikationen mit der Laufzeit ihrer Interpretation verglichen. Es zeigte sich, dass in Abhängigkeit vom Ausmaß des Einsatzes nativer Funktionen nativ kompilierte Applikationen bis zu 21-mal schneller als deren interpretierte Versionen laufen.

Um die Einsetzbarkeit des Frameworks besser zu zeigen, wurde eine Referenz-Applikation, die im Bereich der Finanztransaktionen eingesetzt wird, nativ kompiliert. Obwohl diese Applikation native Funktionen im hohem Maße eingesetzt hat, konnte die Ausführungszeit um 27% reduziert werden. Im letzten Kapitel der Masterarbeit wird noch auf den Einfluss des Kompilierungsprozesses auf die verschiedenen Sicherheitsmechanismen eingegangen und die Einschränkungen des Frameworks sowie zukünftige Ziele und mögliche Weiterentwicklungen beleuchtet.

Abstract

Smart cards and embedded systems gain more and more importance in our today's highly connected world due to their enormous field of application. For each area of operation, there exists a range of specialized smart card applets. As the smart card chips differ from manufacturer to manufacturer, each applet would have to be adapted and particularly compiled for the card's distinct hardware. Therefore, a system was desired to gain independence of the underlying hardware, such that an applet can be executed on multiple different smart cards. The *Java Card* virtual machine and runtime environment offers such an approach. It takes applets written in *Java Card* and executes them by interpreting their code during runtime. This method makes the *Java Card* applet independent from the hardware of the smart card, however, the interpretation during runtime slows down the applet's execution speed. As performance is critical in most of the smart card's fields of application, an alternative approach is desired.

The goal of this thesis was to find a feasible way to compile *Java Card* applets in advance. Investigating similar projects, a open source framework capable of ahead-of-time compilation of *Java* applications was found. This framework translates *Java* byte code to *C* source code which is then compiled using a standard *C* compiler. This open source project was used as a basis for building an ahead-of-time compiler framework able of precompiling *Java Card* runtime classes and applets executable on smart card chips.

Therefore, the framework's translation process had to be modified in such a way that it produces an executable small enough to fulfil the targeted system's memory restrictions. Furthermore, the object life cycle of the overall system had to be modified to meet *Java Card*'s specification as well as an small *Java Card* operating system had to be implemented. The correctness of the final framework's compilation process was verified utilizing a dedicated applet that tests all the features provided by the *Java Card* programming language. Furthermore, the performance of the precompiled applets was compared to the execution speed of their interpretation performed by a *Java Card* virtual machine. Depending on their utilization of native methods, the precompiled applets run up to 21 times faster than their interpreted versions.

To further evaluate the feasibility of the proposed framework, a reference applet, used in the field of financial transactions, was compiled and the gained speed up was measured. Even though this applet heavily relies on native function calls, the execution time was decreased by 27%. Finally, a security evaluation of the produced executable was performed and the framework's restrictions as well as future goals and development were outlined.

Danksagung

Diese Masterarbeit wurde im Jahr 2016 am Institut für Technische Informatik an der Technischen Universität Graz, in Kooperation mit NXP Semiconductors Austria GmbH, durchgeführt.

Mein besonderer Dank gilt Herren MSc Matvey Mukha und Dipl.-Ing. Andreas Lessiak von NXP Semiconductors Austria GmbH. Herr Mukha hat mich in der technischen Umsetzung meiner Masterarbeit maßgeblich unterstützt. Herr Lessiak unterstützte mich in der Dokumentation, sowie der Formulierung der Arbeit. Des Weiteren bedanke ich mich bei Herrn Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger, der mir bei der Finalisierung dieser Arbeit geholfen hat.

Mein weiterer Dank gilt meiner Familie, die mich während des Verfassens dieser Arbeit unterstützt hat. Besonders bedanke ich mich bei meiner Mutter, Frau Mag. Bettina Gressl, meinem Vater, Herrn Dipl.-Ing.- Dr. techn. Franz Gressl, sowie bei Frau Dipl.-Ing. Lisa Maria Mitterhuber, für das sorgsame Korrigieren meiner Masterarbeit.

Graz, Dezember 2016 Lukas Gressl

Contents

1 Introduction **10**
1.1 Motivation . . . 10
1.2 Document Structure . . . 11
1.3 Background . . . 11
1.3.1 Terminal to Smart Card Communication Protocol . . . 12
1.3.2 Basic Concepts of Compilers and LLVM Compiler Framework . . . 12
1.3.3 Java and Java Card . . . 15
1.3.4 The Java Card Open Platform System . . . 17

2 Related Work **19**
2.1 Standalone AOTCs . . . 19
2.1.1 GCJ . . . 19
2.1.2 Toba . . . 20
2.1.3 Java-through-C Compilation . . . 22
2.2 Mixed Mode AOTCs . . . 24
2.2.1 Design and Optimization of a Java Ahead-of-Time Compiler . . . 25
2.2.2 Bytecode-to-C Ahead-of-Time Compilation . . . 28
2.2.3 TurboJ . . . 32
2.2.4 A Method-Based Ahead-of-Time Compiler for Android . . . 35
2.2.5 A Selective Ahead-Of-Time Compiler on Android Device . . . 39
2.3 Practical Research . . . 40
2.3.1 LLVM Java Frontend . . . 41
2.3.2 GCJ . . . 41
2.3.3 VMKit . . . 42
2.3.4 RoboVM . . . 43
2.3.5 XMLVM . . . 43
2.4 Conclusion . . . 46

3 Design of the Framework **47**
3.1 Hardware Setup . . . 47
3.2 Framework Modification steps . . . 48
3.2.1 XMLVM Capabilities and Minimization of used Java Runtime . . . 48
3.2.2 Boarding on STM32 and Modification of Memory Allocation . . . 51
3.2.3 Boarding to the NXP Chip and Adaptation of Object Lifecycle . . . 53
3.2.4 Java Card Runtime/Framework API and Applet . . . 56

3.3 Conclusion 60

4 Implementation of the Framework 61

4.1 XMLVM Conversion Process 61

4.1.1 Intermediate Representation 61

4.1.2 Creation of C Files 64

4.1.3 Memory management of XMLVM 72

4.1.4 Testing of XMLVM conversion process 72

4.2 Java Card Cross-Compilation Framework Modification 73

4.2.1 Reducing the Memory Requirements 73

4.2.2 Boarding to STM32 Discovery Board 77

4.2.3 Boarding to the NXP Chip 84

4.2.4 JCOS and TestApplet 91

4.3 Resulting Compilation Process 97

5 Evaluation 98

5.1 Performance Evaluation and Testing 98

5.1.1 JCF Translation Testing 98

5.1.2 Performance Measurements and Code Size Comparison 100

5.2 Security Evaluation 103

6 Conclusion and Future Work 105

6.1 Restrictions 105

6.2 Conclusion 106

6.3 Future Work 106

List of Figures

1.1 Structure of c-APDU [33, p.422] 12
1.2 Structure of r-APDU [33, p.422] 12
1.3 LLVM framework's implementation of *Three Phase Design* [23] 13
1.4 The two parts of the JCVM[24, p. 8] 16
1.5 Installation process in JC [24, p. 9] 17
1.6 Architecture of the JCOP OS [24] 18

2.1 Base class to subclass mapping in Toba [32, p. 3] 21
2.2 Normalized speedup of Toba [32, p. 8] 22
2.3 Benchmark scores achieved by the single execution methods. Sieve, Loop, Logic, String, Float and Method are single tests of *Embedded CaffeineMark* [39, p. 165] 24
2.4 Mixed mode method call interface [19, p. 172] 26
2.5 Performance evaluation of the standard *C*, the *Java* operand stack and the mixed call interface in partial AOT environment [19, p. 174] 27
2.6 Performance gain achieved by *Java* optimizations: Null Check Elimination, Bound Check Elimination and Object Copy Propagation [19, p. 174] 28
2.7 Speedup achieved by DVM with AOT compilation compared to DVM with JIT compilation measured by multiple benchmarks [31, p. 1052] 32
2.8 Superiority of virtual and static method inlining over static method inlining. Testcases come from the *EEMBC* benchmark [31, p. 1053] 32
2.9 TurboJ's components and their interactions during compile- and runtime [43, p. 120] 34
2.10 Icing's performance evaluated with CaffeineMark 3.0 and BenchmarkPi [42, p.21] 38
2.11 Selective AOTC's speedup measured with CaffeineMark and BenchmarkPi [25, p.5f] 40
2.12 Latest commit message in `aot` branch 43
2.13 Cross-compilation flow of *XMLVM* framework [46] 44

3.1 Simplified class diagram of test project 49
3.2 Step 1 modification flow 50
3.3 Changes in Step 1 to project layout 51
3.4 Test project extended with `TestRamFlash` test case 52
3.5 Step 2 modification flow 53
3.6 Changes to project layout performed in Step 2 54

3.7 Test project extended with `TestCounter` test case 55
3.8 Step 3 modification flow 56
3.9 Changes to project layout performed in Step 3 56
3.10 Test applet extended with test case for transient array and adapted exception handling test case 58
3.11 Step 4 modification flow 59
3.12 Changes to project layout performed in Step 4 59

4.1 Memory layout of XMLVM produced *C* code 72
4.2 Memory layout of *Java Card* object life cycle simulation 83
4.3 Memory layout of *Java Card* object life cycle simulation 90
4.4 Execution Flow Diagram of the *Java Card* OS 95
4.5 Execution Flow Diagram of the *Java Card* OS 96
4.6 Execution Flow Diagram of the *Java Card* OS 97

5.1 Table of r-APDUs of interpreted and compiled `TestApplet` 100
5.2 Time measurements of SHA512 execution 101
5.3 Code size considering the TestApplet 102
5.4 Time measurements of the reference *banking applet* 103
5.5 Time measurements of the reference *banking applet* 103

List of Tables

1.1 Supported and unsupported features of *Java Card* [18] 16

4.1 C structures generated by XMLVM representing *Java* structures 64
4.2 Naming Convention of XMLVM 65

Chapter 1

Introduction

As each of the topics discussed in this thesis, such as compilers, smart cards and virtual machines offers a huge research field on its own, this chapter aims at providing the reader with the necessary background information as well as explaining the goal and the motivation for this master thesis.

1.1 Motivation

As the smart card (SC) technology becomes more and more powerful and such devices become ever more applicable, its influence on our every day's life increases steadily. Nowadays, SCs are deployed in many different fields of application that range from personal identification to banking and from medical healthcare to tourism. This huge variety of applications, which all have different requirements on the SC, is provided to the customer through the implementation of specialized SC applets [33].

To decrease the amount of programming effort, which comes with each applet, the SC software is typically split into multiple layers. Each layer provides functionality to the layer on top of it and thus, offers an interface that encapsulates the details of its implementation. The applets form the peak of this hierarchy. To become independent of the SC's hardware, the applets are typically written in *Java Card*. The compressed code of the applet (known as byte code), which is loaded to the SC, is then interpreted by a virtual machine running on the card. As the interpretation process is cumbersome and does not allow much optimization of the applets' code, it heavily decreases its performance [24].

The virtual machine of *Java* mitigates this performance loss by means of *Just-in-Time* compilation. However, as only parts of the code are visible to this compilation process, the range of the optimizations is limited. Furthermore, code optimization and compilation is much more demanding than mere interpretation and thus, its utilization must be well-considered. Especially for very time critical applets in the field of e.g. banking, this additional overhead might cause issues [31].

To overcome the virtual machine's interpretation process, the applet can also be compiled to native machine code ahead-of-time. This compilation would be performed before the applet is being loaded to the SC. *Android*, as an example, introduced a similar ahead-of-time compilation process, called *ART*, which compiles *Android* applications during their installation process to mitigate the performance loss caused by the byte code's *Just-in-*

Time compilation or interpretation [5].

The goal of this thesis was to find a framework that is suitable for compiling *Java* byte code into native code and modify it to produce machine code executable on a SC. The capability of this framework will be determined by implementing a test project covering the features of the *Java Card* programming language. Afterwards, the framework will be further adapted to compile *Java Card* applets. The resulting framework is referred to as the *Java Card Cross-Compilation Framework* (JCF). Additionally, an evaluation will be performed that shows the performance difference of an interpreted applet in comparison to a natively compiled applet. Furthermore, the impact of the precompilation on the security mechanisms implemented by the *Java Card* virtual machine is discussed. In the last chapter of this thesis the advantages and disadvantages of the JCF as well as its future development is outlined.

1.2 Document Structure

The thesis is split into the following chapters. Chapter 1 explains the basic concepts of compilers and gives an overview about state of the art compilation strategies. Furthermore, it provides a short overview of *Java Card* and introduces the reader to the *Java Card Open Platform System.*

In Chapter 2 several papers considering the *Ahead-of-Time* compilation are summarized and their impact on the design choices of the JCF are explained. Furthermore, the results of a practical research on open-source frameworks and compilers are described in this chapter.

In Chapter 3 the design of the modification process is explained that changes the open-source framework found during the practical research into the resulting JCF.

Chapter 4 explains in detail the translation process of the found open-source framework and the modifications introduced in Chapter 3.

In Chapter 5 the testcases, with which the resulting compilation process is tested, are explained in more detail. Furthermore, the performance impact caused by the precompilation of the *Java Card* applets is evaluated. In this chapter an evaluation on the security mechanisms of the *Java Card* Virtual Machine is performed as well.

In Chapter 6 a conclusion of the thesis is drawn and the future work is outlined.

1.3 Background

The topics discussed throughout this thesis build upon different techniques and technologies in the fields of compilers, programming languages (such as*Java* and *Java Card*), operating systems (OS), Virtual Machines (VM) and Central Processing Units (CPUs). This section is targeted on introducing the reader to these topics and on providing some basic background information about these technologies. These topics are referenced by the projects presented in Chapter 2, as well as by the JCF presented in this thesis. Furthermore, also the communication protocol of smart cards is shortly introduced, as it serves as interface between the terminal an the smart card itself.

1.3.1 Terminal to Smart Card Communication Protocol

The basic protocol, which is used to establish a connection between a smart card and a terminal, is the *Application Protocol Data Unit* (APDU). This APDU is independent from the lower layer of the communication and therefore, can be utilized for both contact-based and contactless communication. There exist two types of APDUs, one for sending commands from the terminal to the smart card (c-APDU), and one for sending responses from the smart card to the terminal (r-APDU). Each c-APDU contains a header, which is mandatory, and a body, which is optional. The header consists of the following bytes. The `CLA`, the `INS`, the `P1` and the `P2` byte. Furthermore, the c-APDU might contain an additional body, which contains one or more bytes. The length of this `data` section is indicated by a mandatory `LC` field put between the `P2` byte and the `data` field. Furthermore, the c-APDU might contain an `Le` field at the end of the `data` field, which indicates the expected length of the r-APDU. The r-APDU only consists of an optional response body and a mandatory response trailer. The response body (`data` field) is followed by the response trailer and contains one or more bytes. If the c-APDU contained a `Le` field, the length of this response body would correspond to the expected length. The response trailer uses two bytes and is used to send the status word from the smart card to the terminal [11]. Figure 1.1 and 1.1 show the structure of the c-APDU and the r-APDU.

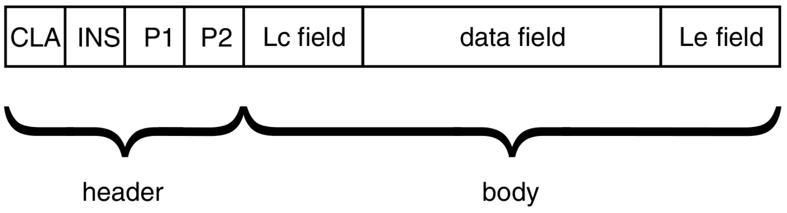


Figure 1.1: Structure of c-APDU [33, p.422]

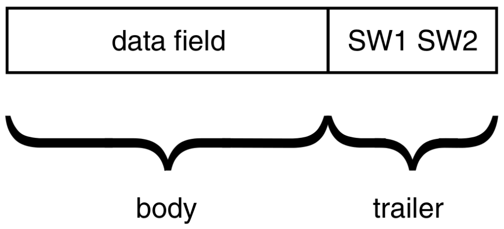


Figure 1.2: Structure of r-APDU [33, p.422]

1.3.2 Basic Concepts of Compilers and LLVM Compiler Framework

A compiler translates a computer program, which is written in a high level programming language, into machine code, which can be executed by the computer. This process is typically split into the following phases: *lexical analysis, a syntax analysis, type checking, intermediate code generation, register generation, machine code generation, assembly*, and *linking*. In the first phase, the *lexical analysis*, *tokens* are generated from the source code.

They correspond to symbols such as variable names, keywords or numbers. During the *syntax analysis* or *parsing* the tokens are taken and arranged into an *abstract syntax tree* (AST), which represents the program's structure. In the *type checking* phase, the AST is checked for consistency requirement violations. After the *type checking*, the *intermediate representation* (IR) is generated, which is machine independent. In this phase most of the optimizations are carried out as well. In the *register allocation* phase the symbolic names of the IR are replaced with number corresponding to registers in the target CPU. During the *machine code generation*, the IR is translated to the actual assembly of the target machine. This assembly code is then further translated to binary representation and the addresses of the various variables and functions are calculated. The *lexical analysis, syntax analysis* and the *type checking* are referred to as the compiler's frontend. The *register generation, machine code generation, assembly* and *linking* are called the compiler's backend. Dividing the compiler's workflow into frontend, IR and backend is commonly known as the *Three-Phase Design* [23]. The IR is the interfacing language linking the frontend and the backend, thus allowing the combination of various frontends with different backends [30].

The Low Level Virtual Machine (LLVM) infrastructure project consists of highly modular and reusable toolchains, as well as compilers. Although its name suggeststhat the LLVM framework is a virtual machine (VM), it has not much in common with traditional VMs, but only supports the libraries to create them. Because of its modular design, various frontends and backends exist that are able to interact with each other. Therefore, when adding a new programming language or a new targeting system to the framework only the necessary front- or backend must be implemented. The rest of the framework can still be used with the LLVM IR as an interface. Also the already existing optimizations on the IR can still be used. Figure 1.3 depicts the interaction of LLVM front- with LLVM backends. Furthermore, the IR is written in a very assembly like way targeted on being well readable by humans [37], [23].

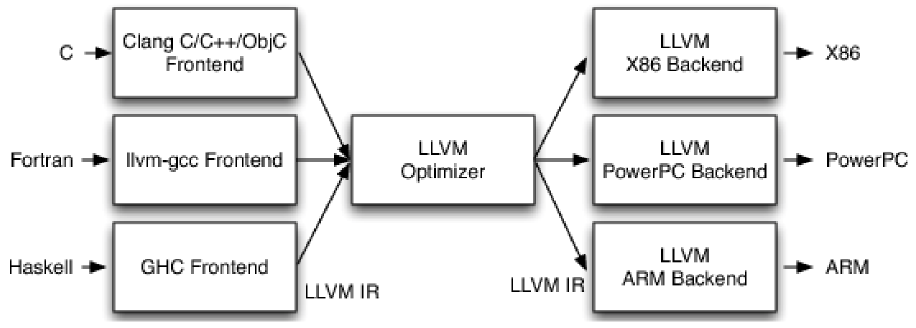


Figure 1.3: LLVM framework's implementation of *Three Phase Design* [23]

Compared to the GNU framework, the LLVM compiler framework provides significantly less frontends. However, the GNU framework does not provide an interface between front- and backend as stable as LLVM's IR. Furthermore, the LLVM framework is considered to be much better extensible, in a stable manner than the GNU' framework. Out of these reasons the LLVM framework was considered as an probable compiler for *Java* to native compilation. A possible frontend for *Java* was searched for in section 2.3.1 [7].

Just in Time and Ahead of Time Compilation Strategies

With the *Ahead Of Time* (AOT) compilation, a complete program is compiled before being executed. The compilation unit, which determines what is being compiled by the compiler, can be as big as a whole file or module, but usually a method is chosen as the unit size. As the AOT-compilation is performed before the program's execution, a range of performance demanding optimizations can be applied. Furthermore, the program is optimized based on information collected from the full amount of the source code. Thus, these optimizations are considered to be more sophisticated than those performed by a *Just In Time* compiler, which only has limited information about the program [14].

The greatest difference between the AOT compilation and the *Just In Time* (JIT) compilation is that a JIT compiler (JITC) compiles the code during runtime. JIT compilation is usually used in combination with an interpreter. The interpreter interprets provided source code or byte code from the syntax tree directly, thus generating machine instructions. Therefore, the interpreter may need to interpret the same piece of code multiple times and thus, producing unnecessary overhead. To mitigate this performance loss, code sections, which are executed many times, are compiled by the JITC. As the compilation needs more performance than the interpretation, the emitted machine code is cached for future access. Therefore, only frequently executed source code/byte code is JIT-compiled. With this technique the interpretation can be skipped by directly executing the cached machine instructions. [30], [14].

The compilation unit of the JITC must be limited as it greatly determines the overall performance of the JIT-compiled program. Therefore, this compilation unit usually does not exceed the size of a single method or a trace, which is a linear instruction sequence within a method with one entry and multiple possible exit points. Greater compilation units would increase the time the program spends for compilation. Thus, the program's execution would be delayed. This decreased compilation unit also limits the optimization that the JITC is able to perform. JITCs are further divided into method-based and trace-based JITCs.[14].

A method-based JITC focuses on the compilation of whole methods using a single method as a compilation unit. During runtime, the compiler detects methods that are frequently executed and marks them as so called hot methods. These methods are compiled and optimized in a standard way using the method's control flow graph. As also AOTCs use single methods as their compilation units, the same optimizations can be theoretically used in method-based JITCs. In practice only fast optimizations are chosen to lower the compilation's impact on the program's execution speed. The optimized machine code is cached and accessed whenever this method is executed. An additional advantage of the method-based JITC is that because of the compilation of single methods/functions at a time, the interaction between interpreted and compiled code is easy, if both use the same calling interface or convention [14].

A trace-based JITC performs its compilation on a trace based compilation unit. Traces are generally not limited by single methods, but may span over a multitude of them. Furthermore, a trace does not have to include all branches in a single or multiple functions. As only parts of methods/functions are compiled, it is not guaranteedthat its behaviour always follows the already compiled trace. Therefore, JITC places a guard on the compiled trace checking if the execution's behaviour leads to a branch leaving the trace. If this is the case, the interpreter mode is fallen back to or, if the branch leads to a compiled trace, the JITC changes to this targeted trace [14].

1.3.3 Java and Java Card

Java is nowadays one of the most popular programming languages. It is a general purpose object oriented language. It was invented to address the up coming issues introduced by the popularization of the World Wide Web. As from that on, software was built for networked consumer devices, it was necessary to gain independence from the device's hardware architecture and operating system (OS). This independence was gained by using a dedicated virtual machine, the *Java Virtual Machine* (JVM) that is responsible for executing *Java* programs. A *Java* program comes in form of *Java* byte code, which is a compressed representation of the *Java* source code. This byte code is packed into `.class` files by the *Java* compiler. Besides this byte code instructions, the `.class` file also contains additional information, such as references to linked classes. These references are dissolved at runtime using so-called dynamic binding [24], [26].

Inside the JVM the `Class Loader` loads the *Java* classes into the memory that is divided by the JVM into several sections. The program counter (PC) register stores the address of the JVM instruction currently executed. As the JVM supports multi-threading, each thread contains its own PC register. The JVM stack also exists on a per thread basis. It is analogue to the *C* stack and contains the stack frames (also called the operand stack). The operand stack is used for storing variables and partial results. Every instruction in the byte code takes operands from this stack and pushes its results back on the operand stack. Therefore, it also resembles the general purpose registers of a native CPU. For each called method a new frame is created .The heap is shared by all JVM threads. In *Java* all objects, which are allocated with the `new` keyword, are automatically placed in the heap memory section of the JVM. The allocation and deallocation of the objects are managed by the Garbage Collection (GC). Furthermore, the JVM also contains a method area, in which the constant pool, the field data and the method data is stored. It is shared among all threads and is analogue to the `text` segment of a *C*program. The constant pool is a representation of the `constant_pool` table, which stores constant values already known at compile time (such as `String` variables or pre-initialized arrays). Additionally to the *Java* stack there also exists a native method stack, which allows *Java* methods to access compiled methods written in another language (such as *C*). Such method calls are realised using a *Java* Native Interface (JNI) [12], [26].

Java uses a syntax similar to *C++*, but was designed to be more programmer friendly. In contrast to *C++*, it does not support pointers as data types, does not allow multi-inheritance and forbids operator overloading. Furthermore, only primitive data types are passed by value. Objects are passed by reference. Another *Java* feature isthat the programmer does not have to take care of freeing allocated memory. The memory management is left to the JVM, which automatically deallocates memory no longer referenced using the GC [24].

Nowadays, *Java* is also available for smart cards in the form of *Java Card*. *Java Card* can be seen as a subset of *Java*, meaning that it does not support all its features. Generally speaking, the development of the *Java Card* programming language was mainly influenced by its targeting platform's resource restrictions. Table 1.1 depicts a range of supported and unsupported features of JC [24].

Supported Java properties	Unsupported Java properties
primitive types: *byte, boolean, short, int*	primitive types: *long, double, float*
Classes: Object and Throwable	Dynamic Class Loading
Exceptions	Finalization
Interfaces	Threads
Dynamic Object Creation	Object Cloning
Generics	Variable-length Argument Lists

Table 1.1: Supported and unsupported features of *Java Card* [18]

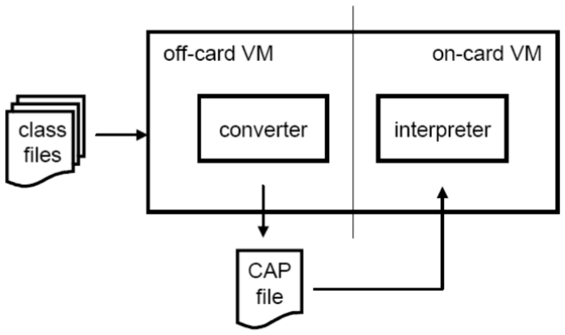


Figure 1.4: The two parts of the JCVM[24, p. 8]

Furthermore, *Java Card* also comes with its own version of the JVM, the *Java Card* VM (JCVM). The JCVM is, in contrast to the JVM, divided into two separate parts, an on-card and an off-card one. As indicated by its name, the on-card part of the JCVM is located on the SC. It is responsible for executing the applet, which is provided by the off-card part in form of `.cap` files. The `cap` stands for converted applet. This execution is performed by interpreting the byte code contained in the `.cap` files. Furthermore, it also takes care of the object and memory management. The off-card part, which is situated on a workstation or desktop, converts the `.class` files generated by the *Java* compiler into `.cap` files. Figure 1.4 shows the two parts of the JCVM and their interaction. The conversion tool checks, whether the features used in the applet are supported by *Java Card*, performs the dynamic class loading including class reference dissolving and static variable initialization. Furthermore, it optimizes the encoding of the byte code instructions to make them more memory efficient. Additionally, *Java Card* also supports the installation of applets after the smart card's distribution. This includes loading the byte code to the smart card, replacing the references of already installed classes with the according addresses and initializing necessary data structures. Figure 1.5 shows the installation of an applet on a distributed smart card [24].

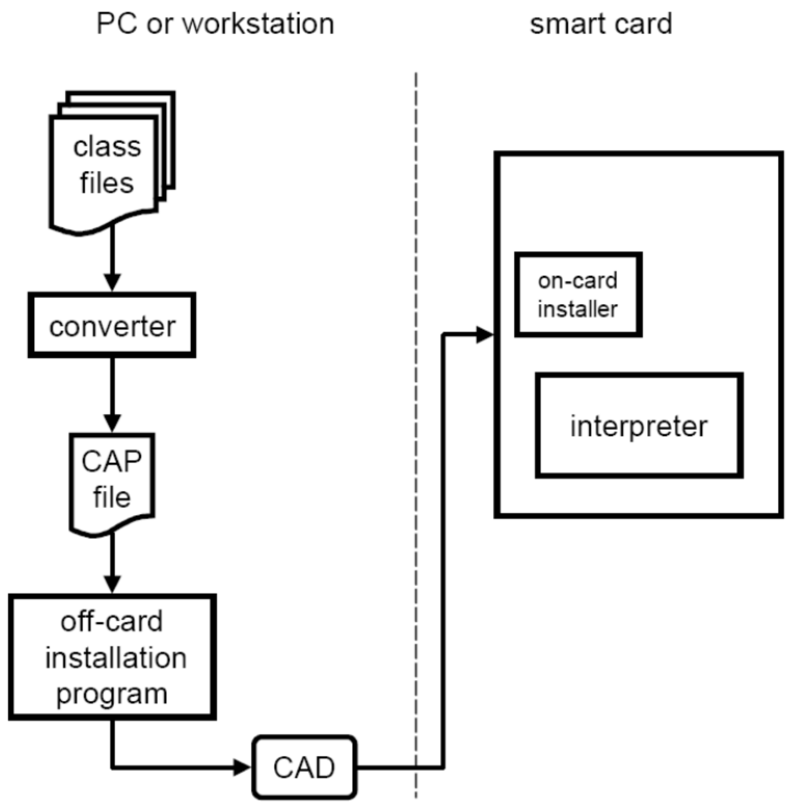


Figure 1.5: Installation process in JC [24, p. 9]

A very important difference between *Java* and *Java Card* is its object life-cycle. As mentioned before, *Java* places class instances (objects) onto the heap. As the heap resides in the device's Random Access Memory (RAM), which is a volatile storage, these allocated objects are only present as long as the device is powered. However, in *Java Card* a distinction is made between persistent and transient objects. The data of persistent objects is allocated in non-volatile memory, which means that its information is preserved over multiple applet executions. Even if a power loss occurs during the applets execution, these objects should be accessible when the applet is run again. Additionally *Java Card* knows transient objects. These objects do not differ from the life-cycle of persistent objects, but their data is not preserved over multiple applet executions [18], [26].

1.3.4 The Java Card Open Platform System

As described in the master thesis of Andreas Lessiak, the *Java Card Open Platform* JCOP was presented in 1998 as the first smart card operating system developed by the IBM BlueZ

Secure Systems team. It was steadily improved by IBM since then. NXP obtained a license for its source code in 2007 and since then NXP develops its own version of JCOP. Figure 1.6 shows the architecture of the JCOP OS [24].

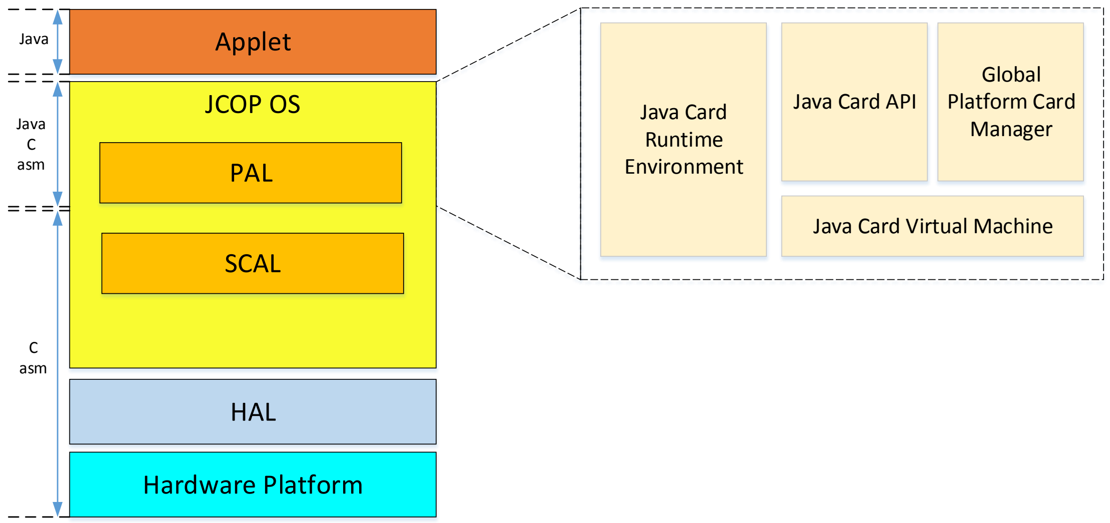

Java
Java
C
asm
C
asm
Applet
JCOP OS
PAL
SCAL
HAL
Hardware Platform
Java Card Runtime Environment
Java Card API
Global Platform Card Manager
Java Card Virtual Machine

Figure 1.6: Architecture of the JCOP OS [24]

The architecture of NXP's JCOP basically consists of three layers. Starting from the smart card's hardware, the first layer is the *Hardware Abstraction Layer* (HAL). This HAL serves as an interface to the hardware and maps generic functions to it. It is used by the layer on top, the JCOP OS layer, to interact with the smart card's physical parts and guarantees hardware independence. It is written in assembler (`asm`) [24].

The JCOP OS itself comprises the *Java Card Runtime Environment* (JCRE), the *Java Card Virtual Machine* (JCVM), the *Java Card Application Programming Interface* (JCAPI) and the *Global Platform Card Manager* (GPCM). The JCRE is responsible for everything related to the runtime, such as handling the applet lifetime, object sharing, etc. The JCVM's main task is the interpretation of the *Java Card* byte code. This interpretation is very performance demanding. Therefore, this part of JCOP is mostly written in `asm` as well. The GPCM manages the card's life-cycle, the upload, the installation and the deletion of applets as well as their security domains. This part is mainly written in *Java Card*. The JCAPI provides the necessary functionality of the OS to the applet layer, which is defined by the *Java Card* specification and is completely written in *Java Card* [24].

The layer on top of the JCOP OS is the applet layer. This layer consists of the user written applets, which are loaded on the smart card and executed by the JCOP OS. The programs in this layer are purely written in *Java Card* [24].

Chapter 2

Related Work

The first step of this master thesis was to find related projects aiming on ahead-of-time (AOT) compilation of *Java*. This research includes both a practical and a theoretical approach. This chapter describes the single projects and frameworks, which have been investigated during this phase. The first section of this chapter contains the theoretical research part of the thesis. During this phase many papers and projects addressing the task of compiling *Java* byte code to native machine representation were examined. The most important works are described in here.

As depicted by Chih-Sheng Wang *et al.*, AOT compilers (AOTCs) can be divided into two classes. The *standalone-mode* class consists of AOTCs, which produce a standalone executable. This means that *Java* programs converted with such compilers are compiled as a whole. Therefore, no interactions with the underlying VM are needed. The second class is the so-called *mixed-mode* class. AOTCs in this class only partly compile *Java* programs in advance. This means that the AOT compiled classes of the program must interact with classes interpreted or compiled during runtime, as well as with the VM [42].

The papers found during this research were classified into *standalone* and *mixed-mode* AOTCs.

2.1 Standalone AOTCs

Standalone AOTCs compile whole applications to native machine instructions. The compilers described in the following sections compile *Java* programs. Compared to *mixed mode* AOTCs they are usually capable of performing better optimizations as every aspect of the program is under the control of the AOTC. However, this method prohibits the interaction with any VM present on the target device [43].

2.1.1 GCJ

When it comes to *Java* to native AOT-compilation, GCJ of the GNU-compiler collection is the best known compiler front-end for this job. GCJ is able to generate *Java* byte code, but also native code from *Java* source code. According to GCJ's manual, it is not only capable of compiling *Java* source but also byte code [9]. The compiler comes with a runtime library called *libgcj*, which contains garbage collection, a byte code interpreter as well as the core class libraries. GCJ can be used to produce pure or mixed native/interpreted applications.

Furthermore, it is utilized in major GNU/Linux distributions to support programs such as *OpenOffice* and *Eclipse* [44].

In an article of 2003, Per Bothner [3, p.1] describes the principle of GCJ as "radically traditional by simply viewing *Java* as any other object-oriented programming language and compiling it as such". As GCC was already well established and used as a compiler for multiple other programming languages such as *C*, C++, Fortran, Pascal, etc., GCJ builds on GCC for providing the compilation infrastructure. Therefore, GCJ basically represents the *Java* program as an AST by utilizing the same structures, which GCC uses as well. Furthermore, it represents each *Java* construct as it would be represented, if it were written in C++. GCC then takes care of the rest [3].

Therefore, GCJ is able to make use of the already existing optimization techniques of GCC, such as loop optimization and register allocation. As the whole *Java* application is considered, GCC's optimization is capable of performing more time-consuming optimization than JITCs. Furthermore, GCJ profits of faster startup speed. Per Bothner also states that the code size of the created executable is not that much bigger than the size of the corresponding byte code. This is because the class files containing the byte code do not only contain instructions but also symbolic information, which is not necessary for ELF executables or libraries [3].

Java supports executing code written in other languages via calling the *Java Native Interface* (JNI). This JNI is also supported by GCJ. Furthermore, it also offers another interface called the *Cygnus Native Interface* (CNI), which can be seen as alternative to JNI. Instead of using a table of functions, such as in the JNI, the CNI is based on the idea that *Java* is a mere subset of C++. Therefore, GCC uses the same calling conventions for both languages. Thus the CNI does not need to transform *Java* calls to native functions or vice versa [3].

In his article, Stefan Krause performed benchmarks comparing *Java* applications, which were AOT-compiled, to their pendants running on a JVM and to *C* implementations resembling these *Java* applications. The AOTCs tested with these benchmarks were GCJ and ExcelsiorJet, which is a commercial AOTC [8]. The JVMs, on which the *Java* applications were run, were JET 5, JET 6, Apache Harmony's VM, and others. Over all, four benchmarks were performed. In one of these benchmarks, GCJ was run without checking array bounds, nor array storage. The benchmarks show that in all four tests GCJ was able to achieve a top four ranking [21].

GCJ is one of the most elaborate AOTCs targeting on *Java* to native compilation. It is a front-end to the GNU-compiler framework. Thus, it makes use of many of its compiler optimizations. As it is well known and a very sophisticated AOTC, it was taken into further consideration for the practical research.

2.1.2 Toba

In their paper, Todd A. Proebsting *et al.* propose an AOTC called Toba capable of compiling *Java* applications to standalone executables by translating byte code to *C* files, which are then compiled using a *C* compiler. This generated *C* files are linked with the Toba runtime-system forming an executable. The concepts of the translation process and the generated code are well explained in their paper. The following paragraphs only summarize the most important design choices [32].

For renaming *Java* methods and fields to *C* functions and variables, Toba removes not supported characters and adds hash-code suffixes to enable *Java*'s method overloading. Thus, all the fields and methods of the different *Java* classes can live in the same namespace [32].

Primitive types in *Java* are mapped one-to-one to *C* types. Objects are referenced via pointers. Each class is translated to a special structure. This structure holds a table of function pointers pointing to the class' virtual methods, an indicator if the class is an array and their static fields. Each class method table contains both the methods of the inherited class and the methods added by the class. This mapping is shown in Figure 2.1 [32].

Array classes also have a length and a vector of elements added to their structures. The objects are translated into structures containing a pointer to their class structure as well as their virtual fields. Class structures are only allocated once, object structures per class instantiation [32].

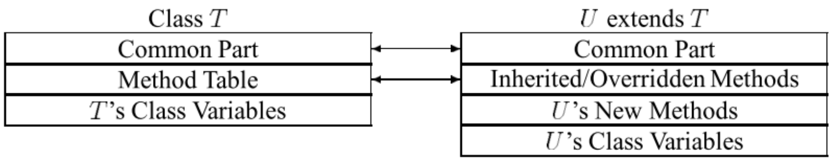


Figure 2.1: Base class to subclass mapping in Toba [32, p. 3]

To generate the *C* code from the *Java* byte code, the Toba translator turns all stack accesses in the byte code to local variables in *C*. The generation of the *C* code for these variables is naive. Therefore, Toba relies on the optimization of the *C* compiler. After the local variables are determined, Toba generates *C* code for each instruction one at a time. All unconditional direct jumps of *Java* become `goto`s to the according labels. Indirect jumps are resolved by using a `switch-case` structure, which models the branches in the *Java* code. The right `case` is chosen by setting a `program counter` variable [32].

Exception handling is solved by setting another local `program counter` variable and a `jmpbuf` structure before the code section enclosed by a `try-catch` is executed. If a exception is triggered, Toba solves this by using `setjmp` and `longjmp` instructions and restoring the previously saved `jmpbuf`. Furthermore, given the type of exception and the `program counter`, Toba checks if the current function can handle the exception. If this is the case, the execution jumps to the appropriate exception handler. Otherwise, the previous `jmpbuf` is restored, with which the `longjmp` was performed [32].

Garbage collection is performed using a modified version of the *Boehm-Demers-Weiser* algorithm. Threads are implemented by using *Solaris* threads. *Java*'s ability, which allows threads to suspend each other and to send asynchronous exceptions, is solved by using UNIX's signal mechanism to handle asynchronous events. The monitors associated with the *Java* threads are modelled using a special structure in Toba consisting of a lock, a reference counter and the identity of the thread holding the lock [32].

The benchmarks used show that the generated code of Toba runs 2.6 to 4.2 times faster than interpreted and 1.5 to 2.5 times faster than the byte code JIT-compiled by the JVM. Figure 2.2 depicts the normalized speedup achieved by Toba compared to the other systems [32].

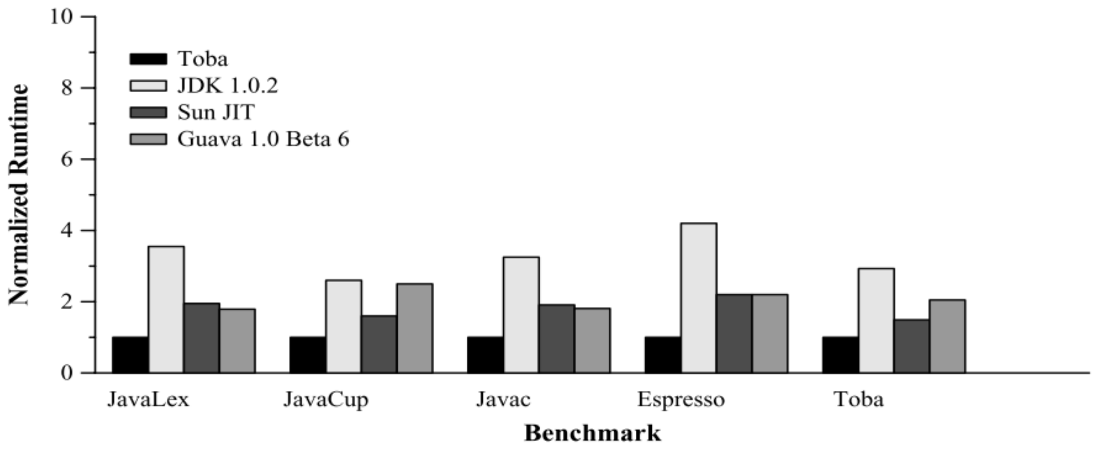


Figure 2.2: Normalized speedup of Toba [32, p. 8]

With their paper, Proebsting *et al.* show that translating *Java* byte code to *C* and utilizing an existing *C* compiler and a self written runtime environment to produce an executable is a feasible way of running *Java* applications outside a JVM environment. This process comes with no restrictions on the *Java*'s language capabilities. Unfortunately they do not mention if or what optimizations their system performs during the translation step. Therefore, it is not certain to what extend the speed-up achieved by Toba is due to the underlying optimization techniques of the *C* compiler. A large performance gain is certainly achieved by omitting the interpretation or JIT-compilation of the JVM. As the authors did not target embedded systems, the size of the produced executables is quite big. The size of the benchmarks which were used for measuring the speedup of Toba range from 200 KB up to 869 KB. However, this does not include the runtime system library, which contains 915 KB of code. The Toba framework can still be found at its developers' webpage[1], but as its maintenance stopped in the year 1998 and it only targets *Java* version 1.1, it was not taken into consideration for a practical review.

2.1.3 Java-through-C Compilation

With their paper, Ankush Varma and Shuvra S. Bhattacharyya propose a *Java* to *C* compiler producing executables small enough to be run on embedded systems. It compiles all classes including the necessary runtime classes and comes with its own runtime environment, which makes any JVM unnecessary. The described compiler is built on some of Toba's concepts, which is described in section 2.1.2, but the authors put much effort in decreasing the large runtime Toba comes with. The compiler takes class files as its input and converts them to *C* code, which is then compiled by a *C* compiler. To convert the

[1]Toba webpage:`http://www.cs.arizona.edu/projects/sumatra/toba/`

Java byte code to an intermediate representation, the authors utilize *Jimple*. Based on this intermediate representation, the *C* code generation is performed [39].

To achieve the compilation of an executable small enough to be deployed on an embedded system, certain restrictions were defined. Therefore, the *Java-through-C* compiler does not support reflection or dynamic loading, nor threads. As the executable runs on a user process, programs, which rely on the JVM as a buffer between them and the underlying system for security reasons, are not supported either. In their paper, the authors gave a detailed description about what concepts and data structures were used to depict *Java*'s object model, which provides a set of features describing object types and operations. These structures are very similar to those proposed by Todd A. Proebsting *et al.* [32]. The most important are summarized in the following paragraphs [39].

Similar to Toba [32], the *Java-through-C* converter removes characters in method names not supported by *C* and uses unique hash-codes as suffixes, which allows all the converted methods to exist in the same name-space [39].

Also the data layout is obviously strongly influenced by Toba [32]. Primitive types in *Java* are mapped to their equivalents in *C*. *Java* objects are transformed into pointers pointing to a instance structure. This structure holds all virtual fields of the object plus a pointer to the class structure. The class structure holds all static fields of the *Java* class, a *class descriptor table*, which contains the class-name, its size and a pointer to the super class, a *method table*, which holds the function pointers, and a *class variables table*, which contains all the static fields of the represented class. The *method table* has a special structure allowing polymorphism and method overloading [39].

To allow for dynamically created arrays, a special object was created. Therefore, all arrays share this class. Furthermore, native *C* functions were written to provide the basic array functionality. As any class may implement multiple interfaces, polymorphic method calls are not handled via a pointer, as it would be the case with a base class, but instead performs a per-class lookup taking the hash-code of the interface method and returning a pointer to the appropriate function to call [39].

Exception handling is transformed by using `setjmp` and `longjmp` instructions with an emulated *exceptional program counter*, which is updated whenever a branch is executed, containing instructions corresponding to the exception. Furthermore, also user written native code is packed into the resulting *C* code. Considering the memory management, the developers utilize *Boehm-Demers-Weiser conservative garbage collection* [39].

Considering code pruning of *Java* code, an algorithm was developed allowing the *Java-through-C* compiler to omit not only whole classes, but also unused class fields and methods. As memory space is very limited in embedded systems, compiling all classes referenced in a *Java* program was no option. The code pruning is based on the *Soot* framework, which creates a call graph of the application, which was trimmed using a *Variable Type Analysis*. The outcome of this analysis was used to compute a set of required entities, from which a dead-code elimination was performed. The result of this elimination is an application, which uses a minimal set of classes, methods and fields and thus, allows the application to be translated to a *C* program fitting on an embedded system [39].

The results of the *Java-through-C* compiler were evaluated using the *Embedded CaffeinMark* and the *Linpack* benchmark. The *Embedded CaffeinMark* consists of six testcases dedicated on measuring various aspects of *Java* applications. The *Linpack* benchmark analyses and calculates linear equations and linear least-square problems. The other execution methods, the *Java-through-C* compiler was compared against, were JVM in interpreter- and JIT-mode as well as the GNU *gcj*. The results showed that, the compiler proposed in this paper performed much better than JVM in interpreter- or JIT-mode in all test-cases but one. Considering the comparison to *gcj*, the *Java-through-C* compiler produced slightly faster code in all but one test-case. This performance gain is visualized by Figure 2.3. As far as the code-size is concerned, the resulting executable produced by gcj is at least ten times bigger than the one produced by the compiler proposed in this paper, when considering the *CaffeinMark* benchmark [39]. The *Java-through-C* compiler shows a very promising approach to natively compile *Java* for embedded systems. As it is stated in the paper [39, p. 1 f.]:

> "We show that a *C*-based optimized compilation strategy can meet the size constraints inherent in embedded systems and provide performance comparable to the best of other implementations without sacrificing *Java* functionality."

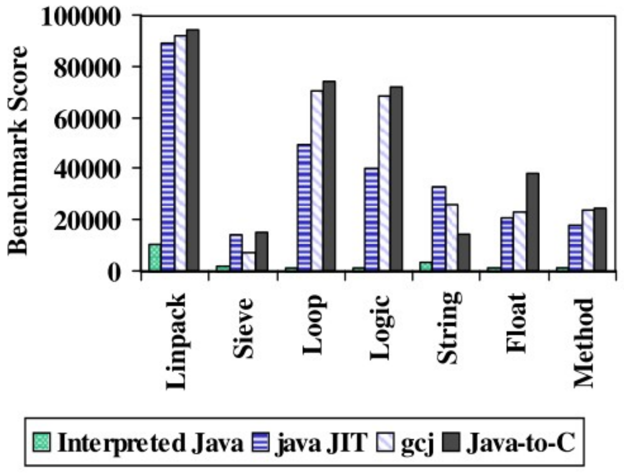


Figure 2.3: Benchmark scores achieved by the single execution methods. Sieve, Loop, Logic, String, Float and Method are single tests of *Embedded CaffeineMark* [39, p. 165]

Especially due to their code pruning method, a vast code-size reduction without many restrictions concerning the functionality was achieved.

2.2 Mixed Mode AOTCs

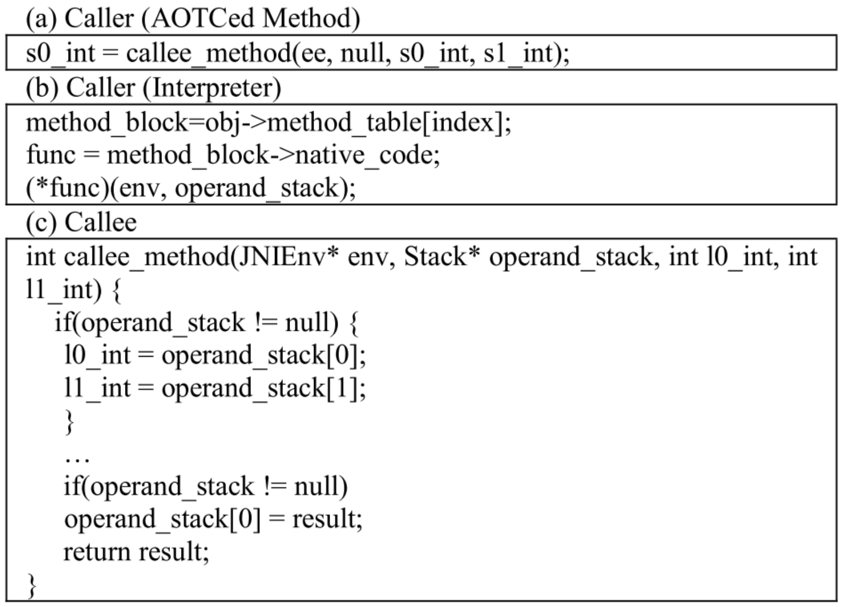
In contrast to *standalone* AOTCs, the *mixed mode* AOTCs do not compile the whole *Java* program including all referenced classes to native machine code. Instead, only certain classes or only methods are compiled, which makes it necessary to integrate AOT-compiled parts of the program with the interpreted/JIT-compiled part. The advantage of this is that the resulting programs are much more memory efficient as *Java* byte code is more compact in respect to native processor instructions. To maximize the effect of the AOTC, only methods consuming a lot of the program's runtime should be pre-compiled [42].

The following sections summarize projects, in which *mixed mode* AOTCs were designed and implemented. Most of them focus on mobile devices with *Android* used as platform.

2.2.1 Design and Optimization of a Java Ahead-of-Time Compiler

In their paper, the authors discuss an *AOT* compilation process, which builds upon the conversion from *Java* byte code to *C*. This paper considers two different ways of compiling a *Java* program into native machine code. The *Combined Compilation* and the *Separate Compilation*. With the *Combined Compilation*, the *Java* program is compiled as one executable, which omits linking the program at runtime. The drawback of this approach is that all the *Java* classes must be present at compile time as the program is already linked in this phase. Therefore, inserting a new class into an *AOT* compiled program, which already runs on an embedded system, requires the whole executable to be recompiled and replaced. With the *Separate Compilation*, each class can be compiled on its own. Thus, inserting a new class can be accomplished without replacing the other parts of the executable. The new class is simply compiled and loaded to the system while running the program. As the linkage is performed during runtime, the new class can be integrated without relinking the whole executable. For this runtime linkage the authors propose an indirection table, which includes an entry for each unresolved reference. Each of these references is then replace by the real address of the referenced field or method, be it a static or object field/method. Furthermore, they point out that this runtime resolution would slow down the execution by adding a lot of access overhead. Due to the expectation that the compiled program does not change frequently and the direct calling of methods and fields is much faster, the authors took the *Combined Compilation* approach [19].

Method Call Interface: As the compiler focuses on a hybrid environment, which allows interpreting *byte code* and executing it using an *AOT* compiled middleware, an efficient method calling interface had to be defined. The caller function in the *interpreter mode* pushes the parameters used by the callee on the caller's operand stack. This stack is provided to the callee via a pointer in such a way that the callee can now copy the variables from the operand stack and assign them to its local variables. This is possible, as the number of parameters is known before runtime. The return value is also pushed to the caller's operand stack. Therefore, the caller can retrieve it, after the callee returned. The second proposed method, the *standard* C *mode*, uses the normal *C* function calling convention. Hereby, the local variables of the caller are used as parameters of the callee and a local variable of the callee is used as return variable. However, this method creates much overhead, if an interpreted method calls an *AOT* compiled method, as the arguments stored in the operand stack must be copied to the *C* stack or to register. The developers chose a mixture of both method calling interfaces allowing the usage and the benefits of both. AOTC-to-AOTC calls are solved using the standard *C* function call interface. The Interpreter-to-AOTC calls provide a pointer to the operand stack. Figure 2.4 depicts a very simple example of this mixed call interface [19].



```
(a) Caller (AOTCed Method)
s0_int = callee_method(ee, null, s0_int, s1_int);
(b) Caller (Interpreter)
method_block=obj->method_table[index];
func = method_block->native_code;
(*func)(env, operand_stack);
(c) Callee
int callee_method(JNIEnv* env, Stack* operand_stack, int l0_int, int
l1_int) {
   if(operand_stack != null) {
    l0_int = operand_stack[0];
    l1_int = operand_stack[1];
    }
    …
    if(operand_stack != null)
    operand_stack[0] = result;
    return result;
}
```

Figure 2.4: Mixed mode method call interface [19, p. 172]

Optimization: For optimization, the authors do not only rely on the optimization methods provided by the utilized *gcc* compiler. Additionally, java-specific optimization, such as eliminating redundant *null pointer checks* and *array bound check* as well as *method inlining* were implemented. The *method inlining* aims towards minimizing the method calling overhead by inlining small static and final methods. This optimization directly puts the methods' bodies at the places they are invoked. Static and final methods are perfectly suited for inlining, as the callee is known and no derived class can override them. A further optimization is the *object copy propagation*. In the *AOT* compiled *Java* program many copies between the operand stack and the local *C* variables are performed. A lot of these copies are redundant and will be removed by the compiler optimization of *gcc*. As the *AOT* compiler also includes garbage collection, a reference *C* variable is updated whenever a *Java* object reference is copied. Such a copy statement also demands a *Java* stack frame save, which is performed by copying the reference *C* variable onto the stack frame. The example provided by the authors:

```
s0_ref=l1_ref;
frame[0]=s0_ref;
```

would be optimized to this piece of code:

```
frame[0]=l1_ref;
```

As *gcc* is not always able to remove these unnecessary statements, the authors implemented their own *object copy propagation*, which is performed during *Java* byte code to *C* translation [19].

Performance Evaluation: The design choices implemented by the authors were compared to the proposed ones. This evaluation was performed by using the *Embedded Microprocessor Benchmark Consortium* (EEMBC). Using the *Combined Compilation*, the compiled executable is 5% faster than a program compiled with *Separate Compilation*. This speed up is due to not needing an indirection table for looking up functions. For evaluating the performance of the mixed call interface, the authors compared it to the *standard C function* and the *Java operand stack* call interface using two different environments. In the first environment, the *full AOTC environmnent*, both application and the middleware are AOT compiled. The tests show that in this environment the *standard C function* is faster than the *Java operand stack* call interface. However, the mixed call interface mitigates the performance loss [19].

In the second environment, the *partial AOT environment* only the middleware is *AOT* compiled, which means that the function parameters must be copied from the provided operand stack. Also the return value must be pushed on this stack. As expected, the *Java operand stack* outperforms the *standard C function* call interface in this environment. In this scenario the mixed call interface provides similar performance as the *Java operand stack* call interface does. This second performance evaluation is depicted by Figure 2.5 [19].

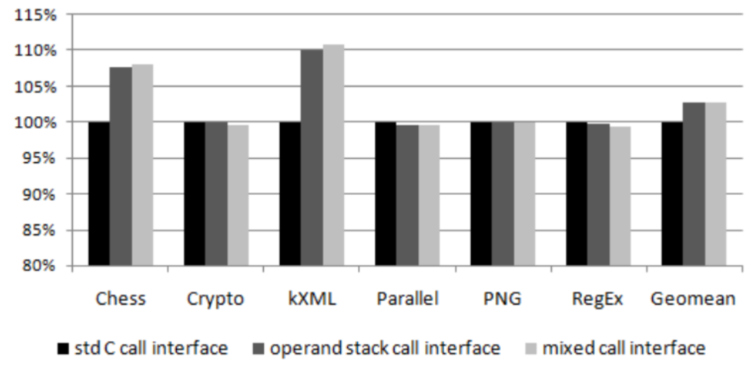


Figure 2.5: Performance evaluation of the standard *C*, the *Java* operand stack and the mixed call interface in partial AOT environment [19, p. 174]

Furthermore, also the *Java* optimizations implemented by the authors were tested on performance gains. To not influence the results, the compiler optimizations of *gcc* were turned off. Turning on all *Java* optimizations, the average performance boost is 65%. From all these optimizations, the object copy propagation proofed to be the most efficient one providing a speed up of 25%. Figure 2.6 shows the performance gain of the single *Java* optimizations achieved on the benchmark tests. Considering the compiler optimization, the standard optimization (*gcc* flag -O2) made the execution four times faster. Flag -O3 (-O2 and also function inlining) showed no performance increase, compared to the standard optimization, at all [19].

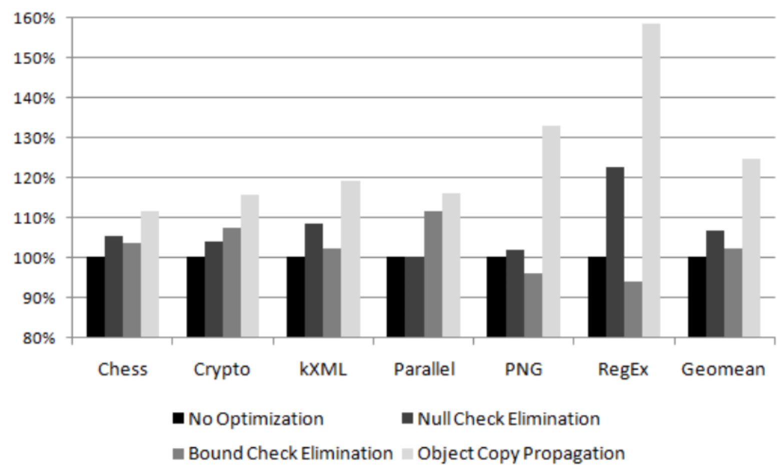


Figure 2.6: Performance gain achieved by *Java* optimizations: Null Check Elimination, Bound Check Elimination and Object Copy Propagation [19, p. 174]

Conclusion: This paper shows a sophisticated approach for *Java* byte code to native compilation. Due to using the implemented *Java* optimizations in combination with the *C* compiler optimizations, the authors were able to achieve a major speed up for applications running on embedded systems. Such optimizations should also be taken into consideration, when looking for an appropriate framework for natively compiling *Java Card* applets. Furthermore, the authors point out that *AOT* compiling *Java* byte code to native machine instructions, by first converting it to *C*, is a very promising approach for embedded *Java* acceleration [19].

2.2.2 Bytecode-to-C Ahead-of-Time Compilation

In this paper, the authors present a very efficient way for AOT compilation for the *Dalvik* Virtual Machine (DVM) employed by *Android*. The motivation behind creating this compiler is that the existing Just-in-Time compilers (JITC) used by *Android* are not able to produce well optimized machine code. Although the DVM uses *dexopt* for byte code optimization by statically linking the application during installation time, the JITC is not able to perform fast execution. The JITC, which compiles the byte code during runtime, is trace based. This means that the compiler only compiles paths in the program code, which are known to be hot traces. However, for producing high performance code, the traces are too short. Also the *Android RunTime* (ATR), introduced with *Android* version 4.4, is not able to produce high performing executable code. Although it is able to precompile applications during installation phase and thereby reducing the compilation overhead during runtime, it only performs weak method-based optimizations. The AOTC, as proposed by the authors, aims on converting hot methods of the pre-installed applications as well as the framework, to *C*. Afterwards, these converted parts are compiled together with DVM source code [31].

The compilation and the optimization of the hot methods is performed on a server, rather than on the target device. The flow of the compilation is as follows:

- Optimization of the byte code with *dexopt*
- Translation of optimized byte code to *C*
- Update of DVM source code with translated methods
- Compilation of DVM together with translated methods
- Installation of DVM on device

The AOT compiled methods in the *Android* framework classes are then marked and loaded in the *zygote* process, where they are linked to the appropriate native code [31].

The *zygote* process creates and initializes the DVM on the device's startup. Whenever an application starts, a new process is forked from this *zygote* process and thus, creating a new DVM. This new DVM comes with all the necessary classes and resources needed by any application. As Android runs on a Linux kernel, Copy on Write (COW) is performed. This means that pages are initially shared by all processes. A page is only copied to a new memory address, when one of the processes attempts to modify it. As the *Android* libraries are not writeable, this means that all *zygote* processes share the same copy of the system classes and resources [29]. Furthermore, the heap is shared by all forked processes [31].

With this approach, the AOT compiled methods do not have to be relinked with the native code of the framework classes at runtime. However, for the application classes, marking and linking of AOT-compiled methods must be done, whenever it is loaded by a forked process. This means that also the memory overhead is reduced, as native code of the AOT compiled methods of the framework classes are shared by all processes and therefore, only instantiated once. A problem is that the DVM JITC is not invoked by the *zygote* process and thus, no native code is stored in the heap memory. If during runtime the JITC compiled the traces for the same hot methods of the framework classes, which were already AOT-compiled, they would be allocated as duplicates in the heap. Therefore, the authors check during the execution of the application, if a method is already AOT compiled and call the linked native code of the method [31].

The AOTC starts after the optimization performed by the statically linkage of *dexopt*. As the optimized byte code is translated, no constant pool (CP) lookup must be performed, as the CP resolution was already performed by the *dexopt*. Byte codes, which access an object-field or a virtual method via the virtual method table (VMT), also include a special index. At this index in the CP, the field or method name is stored. With this name, the offset in the object or in the VMT can be obtained respectively. The *dexopt* pre-fetches these offsets and replaces the indices to the CP with the offsets directly in the byte code. This optimized byte code is then translated into an intermediate representation (IR) and a control flow graph (CFG) is produced by the AOTC. Based on the IR and the CFG the *Java* specific optimizations are performed, before it is converted into the final *C* code [31].

In *Android*, the byte code is interpreted by the DVM interpreter and thereby executed. Each *Java* thread is assigned an interpreter stack, which is divided into stack frames. For each method, a new stack frame is created. Furthermore, the interpreter stack also holds virtual registers used for computation as well as a status information, which is needed for garbage collection, method invocation and exception handling. On a method call, the stack frame of the caller is pushed to the interpreter stack, the caller's program counter (PC) is saved on the caller's frame and the arguments are passed by copying the according virtual registers of the caller to the registers of the callee. The return value is also passed using the virtual register of the caller and the callee. In the conversion to the *C* code, each virtual register is translated using a *C* variable. As the DVM's virtual registers can hold values of different types, the used *C* variables depend on these types. Therefore, the authors used structures containing elements for the basic types (int, float, double, long) and a pointer for object reference. For example, `v0_reg.i` represents a local variable storing an integer. The status information is replaced by a special environment variable, which is used for accessing the interpreter stack (called `ee`) [31].

Method Call Interface: Similar to *Design and Optimization of a Java Ahead-of-Time Compiler for Embedded Systems*, also the authors of this paper introduced a mixed call interface for being able to perform both AOTC-to-AOTC as well as interpreter-to-AOT method calls. Considering an ARM CPU, the native code of a call to a function would pass the first four arguments using *physical registers* and the rest would be passed using the *native stack*. For an AOT-to-AOT method call the standard *C* calling convention would be suitable. However, for a interpreter-to-AOT call this would not be possible, as the interpreter uses the interpreter stack for passing arguments. Therefore, the authors chose to use the physical registers for passing the first four arguments and the interpreter stack for passing all other arguments. To make the interpreter-to-AOT call possible, the interpreter must copy the according arguments from the interpreter stack to the registers before calling the AOT compiled function. Listing 2.1 shows, how an AOT compiled method invoked by either an AOT compiled or an interpreted method handles the argument passing. This produces some overhead, as memory to register copies must be performed. The AOT-to-AOT call on the other hand is very efficient as only the *physical registers* are used [31].

Listing 2.1: AOTC callee method argument copy [**p.1051**, 31]

```
int callee_AOTCMethod(Env* ee, Object* a0, int a1, int a2)
{
  Frame* frame = getFramePointer(ee);
  v0_reg.ref = a0;
  v1_reg.int = a1;
  v2_reg.int = a2;
  v3_reg.int = frame[0];
  v4_reg.int = frame[1];

  //Code of the method

  return result;
}
```

A method call from an AOT compiled method can either be a call to another AOT compiled method, which would simply invoke the native code of this method, or to a non

AOT compiled method. In the latter case the method will be called by the interpreter [31].

To make the native code of the AOTC methods accessible to the DVM, it is linked to its method table by using an *AOTC table* holding pointers to the native methods. These pointers are based on the class name and the method table offset, which is obtained during the *dexopt's* static linking. The modification of the method table is done during loading the framework class, which links the *AOTC table* with the method table. Furthermore, also the AOT compiled methods of the application are linked to the DVMs of the single application processes [31].

Optimization: The authors implemented *Java* optimizations such as *null pointer check elimination* and *copy propagation*, but also *method inlining* similar to [19]. Furthermore, they also implement *spill optimization*, which helps reducing the slots used for saving references to possible alive objects. This optimization performs an analysis when the GC is triggered and thus, makes sure an object is alive before saving its reference.

Performance Evaluation: To evaluate the advantages of the AOTC, the authors used an analysis tool to decide on hot methods, which should be compiled. The benchmarks used for evaluating the performance increase of the AOT compilation show that it is able to produce much better optimized native code than the JIT compilation. The benchmarks used for the evaluation are the *EEMBC*, the *AnTuTu-UI*, the *Quadrant-CPU*,the *Linpack-single*, the *Linpack-multi*, and the *BenchmarkPI* benchmark. The most significant speedup was achieved by using *BenchmarkPI*, which spends most of its runtime in one single hot method, which is therefore compiled to native code. The *BenchmarkPI* is executed six times faster using AOT compilation than using the standard DVM JIT compilation. Figure 2.7 shows the speedup achieved by the AOT compilation compared to the JIT compilation in the single benchmarks. Considering the *Java* optimizations, the method inlining proofed to have the heaviest impact on the performance, with a speed up of 21% for both static and virtual method inlining. Static method inlining alone only gave a performance gain of 12%. This performance test was performed using the tests of the *EEMBC* benchmark. The optimization comparison is depicted by Figure 2.8. Comparing the DVM AOT compilation with the compilation performed by *Android*'s ART showed that the compiler proposed by the authors still gives an performance boost of 44% on the average [31].

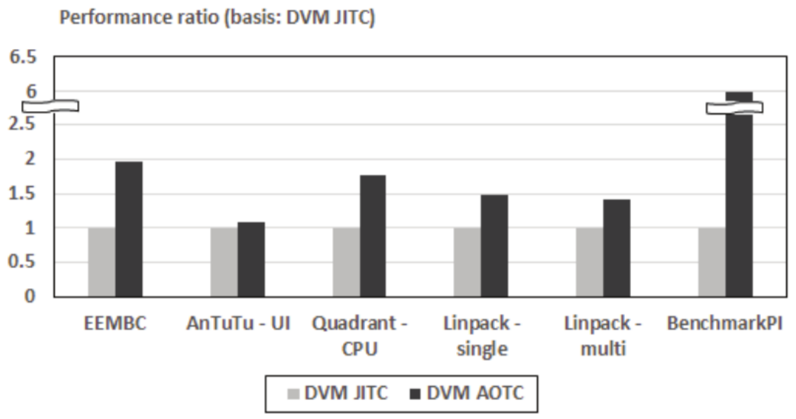


Figure 2.7: Speedup achieved by DVM with AOT compilation compared to DVM with JIT compilation measured by multiple benchmarks [31, p. 1052]

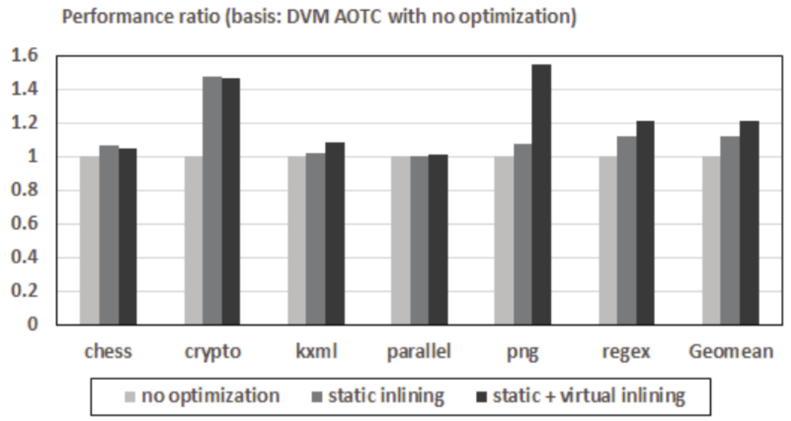


Figure 2.8: Superiority of virtual and static method inlining over static method inlining. Testcases come from the *EEMBC* benchmark [31, p. 1053]

Conclusion: This paper shows a way of performing AOT compilation on *Android* and gives a good insight into problems concerning the interaction between the native compiled methods and the DVM. Especially the calling interface for interpreter-to-AOT, which makes use of the interpreter stack and the virtual registers to achieve the mapping for the physical registers and the native stack of the ARM CPU, is a useful part also for *JavaCard* to native compilation. On top of that, this paper emphasises the importance of the *Java* specific optimizations for enhancing the performance even further.

2.2.3 TurboJ

In this paper, the authors describe their design and implementation of an AOTC compiling *Java* byte code to native code interacting with an standard JVM and therefore, allowing a mixed mode execution of interpreted/JIT compiled and AOT compiled classes. As TurboJ mainly focuses on embedded systems, its main targets are *closed* applications. This means that all necessary classes of the application are already present at compile time. However, the compiler is able to compile applications, which dynamically load classes from outside (such as the internet) during runtime, as well. Furthermore, the embedded environment comes with space and performance constraints both targeted by TurboJ [43].

The limited memory space of an embedded system restricts the amount of classes, which can be AOT compiled. This method prohibits TurboJ of optimizing the *Java* application to the same extend as *standalone* AOTCs. However, this means that the compilation framework does not need to take care of the thread management and the GC, but instead relies on the VM for performing these tasks as well as object management, class/library loading and execution of the byte code not compiled in advance. A main goal of TurboJ is, to be compliant with off-the-shelf VMs. The AOTC converts the *Java* classes into *C* files and so called interlude files, which have a class file format. The *C* files are then compiled to native machine instructions using a native *C* compiler. The interludes are used to call the natively compiled methods from the JVM. The JVM provides entry points to its services for the compiled classes [43].

In TurboJ, there are three different types of calling conventions between methods. An AOT-compiled method calling an AOT-compiled method uses the standard calling convention of *C*. An interpreted/JIT-compiled method calling an AOT-compiled method makes use of the interludes. An AOT-compiled method calling an interpreted/JIT-compiled one uses special entry points in the VM. An interlude is a class file containing the method declarations of the AOT compiled class with a special attribute added to them. The authors then made use of the *invoker attribute* provided by the JVM for each method of a class. A TurboJ initialization routine sets the attributes to a special TurboJ invoker, for each interlude. Figure 2.9 visualizes TurboJ's single components and how they interact during compile- and runtime [43].

Method Call Interface: For applying the correct calling convention, the caller must know if the callee is AOT compiled or not. If this information is not known to the caller, it calls a specific routine which asks the JVM to check the callee's method's state. The VM knows the state as it has all the interludes loaded. After obtaining the information, this routine converts the method invocation accordingly and saves the method's state for future checks [43].

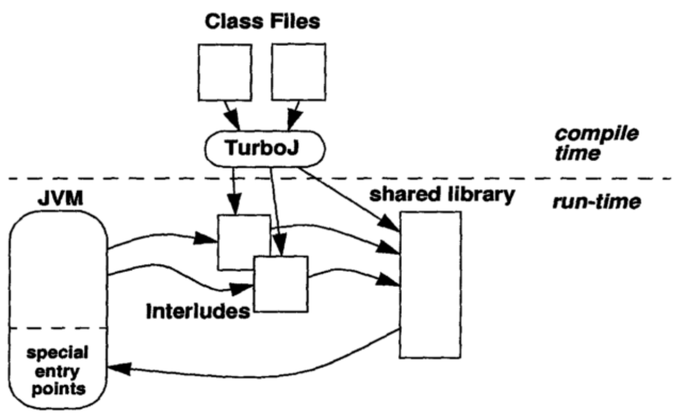


Figure 2.9: TurboJ's components and their interactions during compile- and runtime [43, p. 120]

Optimization: On a whole-program-basis, TurboJ tries to compute the entirety of the input classes to gather global information. Therefore, it follows all dependencies including external classes and adds them to the input set. In the case of a closed application, all referenced classes are available for creating the native code and interludes. If the application is not closed, TurboJ makes certain conservative assumptions about the missing classes. Using this global information, TurboJ performs a range of optimizations. It tries to identify, if a method is final, which means that it is not overridden by a subclass. If this is the case, a static method dispatch is used instead of a virtual one. Furthermore, already at this stage, the AOTC determines what methods to call using the standard *C* calling convention, as it knows which methods are AOT compiled and which not. Also inlining certain methods and determining field offsets can be performed on basis of the global information [43].

On a class-basis, the AOTC makes use of the field offsets collected on a whole-program-basis. Usually field references, which are contained in the *Java* byte code, are converted to offsets during the *constant pool resolution* by the JVM. This constant pool, in which the field offsets are stored, is then used by the `getfield_quick` instruction, which is a optimization of the `getfield` instruction. The `getfield` instruction uses an index into the constant pool identifying the constant pool item, which is then resolved for retrieving the field width and offset. The `getfield_quick` instruction directly uses the field offsets, which are already determined by TurboJ. For the field offsets, which cannot be determined up front, the AOTC generates routines performed at both class and method initialization, which try to compute as much field offsets as possible. This field offset resolution "reaps the same resolve once benefit as the `getfield_quick` optimization" [43, p.124].

On a method-basis, TurboJ avoids the Java-Stack simulation by transforming the byte code instructions into many small expression trees. On basis of these trees, the TurboJ performs the conversion to the native *C* files. The register allocation is left to the *C* compiler. Thus, the simulation of a Java-Stack using local variables can be avoided in most cases. Similar to the other described projects also the developers of TurboJ made use of the tighter semantics of the *Java* byte code for performing specific *Java* optimizations, such as elimination of redundant null pointer checks [19] or the optimization of field accesses. Considering field access, the developers of TurboJ made use of the *Java* byte code to optimize the accesses of fields within an object. TurboJ converts a field access of a object to an access in an array using the according index. Thus, `this.count` becomes `this[count_offset]` in the resulting *C* file. In contrast to the *C* compiler, TurboJ knows that "different fields in different objects cannot reference the same memory location" [43, p.125]. Therefore, considering accesses of fields within objects, TurboJ optimizes the resulting *C* code in such a way that it stores accesses to arrays with constant indexes in separate local variables. This is especially useful, when using object fields as loop invariants. This optimization prohibits unnecessary accesses into the array using the constant index [43].

Performance Evaluation: Concluding, the authors of the paper compare AOT-compiled applications with their JIT-compiled and interpreted versions. Using *CaffeineMark 3.0* benchmark, they show that the applications compiled with TurboJ perform five times better than the JIT-compiled and 32 times better than their interpreted versions. This great impact on the performance is largely traced back to the shape of the resulting *C* code. As TurboJ generates code with "sufficient nesting depth" [43, p. 128], the used *C* compiler is able to optimize the *C* code by performing the register allocation [43].

Conclusion: Although the paper was published in the year 1998, it provides a good overview of obstacles, which need to be tackled when designing a *Java* byte code to native AOTC. However, in comparison to the other papers, the authors of TurboJ do not provide detailed information about how exactly they performed a call from an AOT-compiled method to an interpreted/JIT-compiled one and vice-versa. They only state having implemented a TurboJ specific routine managing these calls. It would have been very interesting to get a deeper look into this process. Furthermore, they did not really go into detail about the performed optimizations, besides the optimization of field accesses. Also, in which way they use the JVM for GC and object management is not described in detail. Unfortunately, the project's webpage `http://www.opengroup.org/openitsol/turboj.html` is no longer available [38].

2.2.4 A Method-Based Ahead-of-Time Compiler for Android

In their paper, the authors introduce an AOTC called Icing, which focuses on the compilation of *Java* classes in *Android* applications. Icing is a byte code to *C* converter, which takes DEX byte codes as input and converts them to *C* code, which is then compiled using *GCC* as *C* compiler. This method is very similar to the one proposed by Hyeong-Seok Oh *et al.* [31]. However, instead of using the interpreter stack for passing arguments when calling an AOT-compiled method from an interpreted one, Icing makes use of the *Java Native Interface* (JNI) library to call the AOT-compiled methods through the DVM. The goal of the developers was to implement an AOTC, which is capable of reducing the compilation overhead of the DVM and to perform more aggressive optimizations. As the resulting native code should execute in cooperation with the DVM, a further goal of Icing is to avoid compilation of core-library methods [42].

Hot Method Detection: The conversion process of Icing starts by profiling the methods of the *Android* application. The result of this first step is the detection of hot methods, which means that only methods, which take up a significant amount of the runtime, are proposed as candidates for the AOT-compilation. Afterwards, the method's byte codes are compiled to efficient native instructions. The interaction between the DVM and the native methods is performed by utilizing a bridge library. To further enhance the performance, a range of optimizations were implemented. These steps are described by the authors in much detail. The following paragraphs depict the most important information about the compilation process of Icing [42].

For profiling the methods in an *Android* application, Icing combines already existing tools for measuring the amount of execution time each method consumes during an probable application usage. To simulate this usage of the application, potential user behaviours are generated up front. Based on this information Icing tries to detect hot methods. For each candidate method, Icing runs through its execution flow and calculates the execution time spent in user-defined methods. The ratio between the runtime spent in user-defined methods and the total execution time is determined. If it exceeds a predefined threshold, the candidate method is put into an AOT-list. Furthermore, each of the hot methods in this AOT-list is examined on its JNI calls. As these kind of calls are very time consuming, Icing tries to avoid converting methods with frequent JNI invocations. To accomplish this, the ratio between the method execution time and number of these invocations is determined and again compared to a predefined threshold. If it exceeds this threshold, the method is removed from the AOT-list. Only the methods contained in the AOT-list are compiled by Icing [42].

The conversion process of the byte code starts by disassembling the DEX files to get the necessary information for further generating the *C* code. In this phase, the JNI headers are generated for bridging the DVM and the native code as well. Furthermore, the native methods are chained together in such a way that as many native calls as possible are produced. This helps in decreasing the overhead of the JNI calls. This is done by modifying the indirect-jump instruction in such a way that the native code of the callee is directly jumped to. This short-cut can only be performed, if the callee is marked for precompilation as well. Therefore, all user-defined methods called from the methods in the AOT-list are converted [42].

Similar to the AOTC proposed by Hyeong-Seok Oh *et al.*, Icing makes use of a standard *C* compiler [31]. Therefore, the compilation part of Icing only consists of converting the *Java* byte code to *C* code. One major part of this process is to reflect the object oriented features of *Java*, such as object creation, static/virtual method invocation, etc. These features were implemented in the bridge library as part of the JNI. Another issue that had to be tackled by the developers, was the method overloading. In Icing this is solved by giving each method an unique name by combining the package, the class and the method name in *Java* to create a corresponding function name in *C*. The third obstacle to overcome, when generating the *C* code, was to map the virtual registers in the DEX byte code to *C* variables. As virtual registers are basically typeless, Icing maps them using a union of eight basic variable types and a void pointer. Thus, the assignment of a virtual register to, e.g., an integer and an object in the byte code can be mapped by assigning the integer variable and the void pointer to the union in *C*. This is depicted by Listing 2.2 [42].

Listing 2.2: Union representing a virtual register [**p.19**, 42]

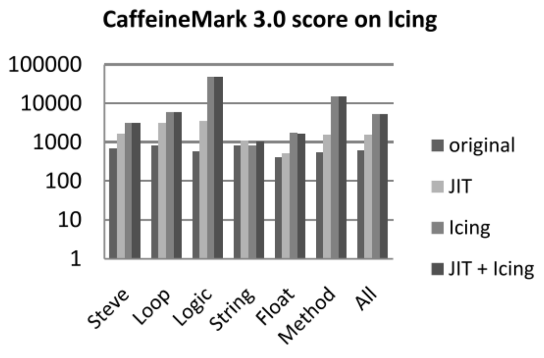
```
typedef union_jValue{
    jboolean z;
    jbyte b;
    jchar c;
    jshort s;
    jint i;
    jlong j;
    jfloat f;
    jdouble d;
    void* l;
} _JValue;
```

Method Call Interface: As Icing uses the JNI interface to call the native compiled methods, the performance suffers from the overhead of this mechanism. The call-out mechanism is used to call a native function from the VM and needs argument passing, native initialization and returning. Dynamic inlining can be used to reduce this overhead. The call-back mechanism is used to access resources of the VM from a native function. This operation produces more overhead than the call-out mechanism. This overhead contains resolving the offset or the index into the constant pool, before performing the context. Furthermore, also referencing the JNI environment variable may produce additional overhead. Icing especially tackles the overhead produced by the call-back procedure by implementing three optimizations: the AOT resolution, caching and method-cloning [42].

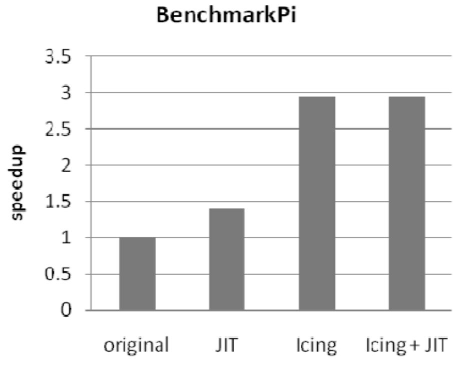
Optimizations: As already described in previous papers, instructions such as method invocations and field accesses use references in symbolic form. These references are usually converted to indexes by the constant pool resolution during runtime. These indexes point into the respective constant pool. To reduce this overhead, this resolution is performed ahead of time. Therefore, the offsets of the variables in their respective constant pools are already available at execution time. This method is used to speed up the call-back operations in the generated native code [42].

However, this static constant pool resolution is not applicable for static field accesses and static method invocations. The standard way of a JNI to access static fields or methods is to obtain the class name and then to compare the field/method name to obtain its ID. Icing caches this ID using a hashtable to speed up future static field accesses and method invocations [42].

Another optimization implemented by the developers is the so called method cloning. As already mentioned, Icing tries to avoid JNI calls by making the chain of AOT-compiled methods as long as possible. Furthermore, calling an native method from an native method is done by using direct jumps, instead of using the JNI interface. However, also calling a native method from a non-native method creates call-out overhead. This overhead can be omitted by keeping an AOT-compiled version and a byte code version of the same method. Thus, a native method can call the AOT-compiled version and a not natively compiled method can call the byte code version of the method. This enhances the performances but consumes more memory. Therefore, this optimization can only be used for specific methods [42].



(a) Score of CaffeineMark 3.0 using Icing



(b) BenchmarkPi comparing Icing to JIT

Figure 2.10: Icing's performance evaluated with CaffeineMark 3.0 and BenchmarkPi [42, p.21]

Performance Evaluation: The performance of Icing was measured by using a range of open source benchmarks. Several version were compared. A version, in which the DEX byte code was interpreted, one, in which it was JIT-compiled by the DVM, one, in which the interpretation of the byte code was combined with the AOT-compilation of Icing, and one, in which the JIT-compilation was combined with Icing. The benchmarks show that for most tests Icing in combination with interpretation or JIT-compilation achieves a major speedup compared to normal execution, even without the removal of hot-methods with too many invocations. Running the Icing version of the *CaffeineMark* benchmark with hot method identification performed 2.83 times better compared to the JIT-compiled version. This is depicted by Figure 2.10a The Icing version of *BenchmarkPI* was enhanced by a factor of 2.1 compared to the JIT-compiled version, which is shown by Figure 2.10b. However, one testcase shows worse results when applying Icing. This is due to the significant overhead produced by the JNI calls [42].

Especially interesting results are shown by the comparison of Icing to GCJ. As already discussed in section 2.1.1, GCJ is a standalone AOTC compiling *Java* programs to executables. As the *Java* programs targeted by GCJ are executed using a JVM, but Icing targets *Android* applications executed by the DVM, the authors chose the *CaffeineMark 3.0* benchmark, which was developed for both JVM and DVM. The comparison focused on both performance and code size. Particularly the resulting code sizes of the AOT-compiled benchmarks show major differences between Icing and GCJ. Using Icing, the code size increased from 17KB without optimization to 69KB with optimization. However, GCJ blew up the codes size from 13KB without optimization to 44.1MB with optimization. Furthermore, the authors showed that the performance of GCJ's AOT-compiled version of the benchmark was in some cases even worse than when using Icing [42].

Conclusion: Icing shows that using the JNI interface for integrating AOT-compiled methods into an *Android* application executed by the DVM is practically usable. Although this invocation method creates considerable overhead, it does not influence the performance gains achieved by the AOT-compilation severely, when performing a costmodel to mark the methods suitable for precompilation. In particular, the comparison of Icing with GCJ emphasizes the advantage of a mixed-mode to a standalone AOTC. As in the domain of embedded systems memory is very limited, this advantage is even more crucial.

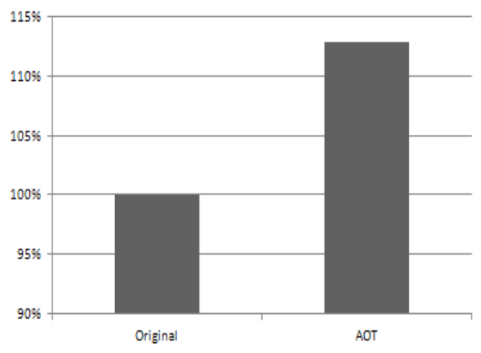
2.2.5 A Selective Ahead-Of-Time Compiler on Android Device

In their paper, Yeong-Kyu Lim *et al.* describe the design and the implementation of an AOTC, which make use of the already implemented JITC of the DVM. AOTC and JITC both transform byte code to native processor instructions with the difference that AOTC compiles the byte code before it is being executed, but JITC compiles the code during execution. The DVM utilizes a trace-based JITC. At first, the authors extended this trace-based JITC to a method-based one. This version still suffered from the warming-up delay, until a method is identified as hot. This disadvantage was overcome by the final AOTC, as it compiles the identified hot methods in advance. In contrast to the method-based AOTC developed by Chih-Sheng Wang *et al.* [42] described in section 2.2.4, the selective AOTC does not convert the *Java* byte code to *C* before compiling it to machine code using GCC, but instead directly compiles it. The final AOTC consists of three major components: the *Hot Method Profiler*, the *Static Selective Ahead-Of-Time Compiler*, which applies certain optimizations and the *Dynamic Loading and Linking Framework*. These components are described in the following paragraphs [25].

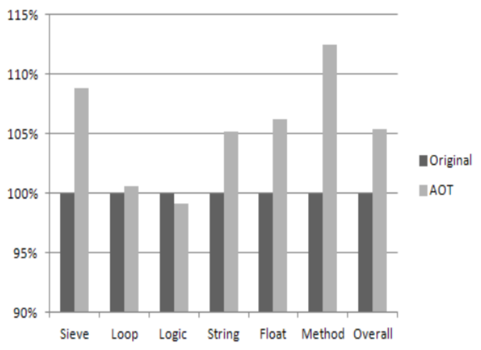
The *Hot Method Profiler* detects hot methods in application or the *Android* core framework code by tracing the execution frequency of all methods of a process. It returns a list of methods, which exceed a predefined threshold of executions [25].

The *Static Selective Ahead-Of-Time Compiler* builds upon the already existing JITC of the DVM, which provides both method- and trace-based compilation techniques. The AOTC uses the method-based compilation and compiles the hot methods provided by the *Hot Method Profiler*. As the JITC is optimized for trace-based compilation, the initial method-based AOTC suffered from branch range and data access range limitation. Therefore, the compiler was modified in such a way that it generates unconditional instead of conditional branch instructions supporting a greater range of offset. Furthermore, it rearranges the placement of the data. In its original form the compiler placed the data at the end of the code section. As the data access instructions are limited to an offset 1020 byte from the program counter (PC) and the compiled methods might exceed this offset, the AOTC rearranged the data placement to certain safe-spots within the code.

Optimization: Furthermore the developers implemented several optimizations, such as the *method inlining*, *array optimization* and *loop optimization*. For the *loop optimization* the developers implemented process which moves the loop invariant. Thus, the non changing computations performed inside the loop are recognised and moved outside the loop's scope. The *array optimization* is a redundancy check performed to avoid unnecessary null pointer and array-bound checks. The AOT gets the original DEX file containing the byte codes plus the list of hot methods as input, compiles the byte code of these methods to platform dependent assembly and adds this assembly to the end of the DEX file. The output is then an optimized DEX (ODEX) file containing both byte- and assembly code [25].



(a) Speed improvement of BenchmarkPi



(b) Performance increase of CaffeineMark

Figure 2.11: Selective AOTC's speedup measured with CaffeineMark and BenchmarkPi [25, p.5f]

The *Dynamic Loading and Linking Framework* is the runtime environment, which makes sure that the AOT compiled code provided by the ODEX file is cached and executed. Therefore, the linking framework resolves the addresses to the pre-compiled methods and handles the mode-change among AOT compiled, JIT compiled and interpreted code. The byte code which is not AOT compiled is interpreted or JIT compiled [25].

Performance Evaluation: The speedup of the *Static Selective Ahead-Of-Time Compiler* was measured using the *CaffeineMark* and the *BenchmarkPI* benchmarks, the same benchmarks Chih-Sheng Wang *et al.* were using to evaluate Icing as well [42]. Using the selective AOTC, *BenchmarkPI* runs 13% faster in average. The *CaffeineMark* benchmark performs 5.3% better in average compared to the original version. The performance increase measured by the *CaffeineMark* benchmark is depicted by Figure 2.11b and the speedup of the *BenchmarkPI* is shown by Figure 2.11a [25].

Conclusion: The selective AOTC described in this paper shows a very interesting way on how to enhance the performance of *Android* applications on the basis of the already implemented JITC of the DVM. Yeong-Kyu Lim *et al.* state that their selective AOTC produces even more efficient code than Icing [25].

2.3 Practical Research

The first part of this thesis was to find a suitable compiler framework to generate an executable from a *Java* application. As already seen during the literature research, there are different approaches on how to natively compile *Java* applications. Therefore, both AOTCs, but also converters transforming *Java* byte code to other programming languages such as *C*, on which already existing compilers can be used, were taken into consideration. Frameworks translating *Java* source code to any other programming language were deliberately ignored, as compiling *Java* source code instead of byte code instructions was no option. The goal was to find a converter/compiler with such a reasonable complexity that it would feasible to adapt it to convert/compile *Java Card* applications or modules and to execute them on an NXP platform. The following sections describe the single frameworks that were tried out practically.

2.3.1 LLVM Java Frontend

The first compiler fronted, which was taken into consideration was the *LLVM Java Frontend*. As its name suggests, it is a frontend for the *LLVM* compiler framework. According to its documentation, it translates *Java* byte code to *LLVM* byte code and uses the *LLVM* backend to perform further compilation. The byte code translation is performed in two steps. In the first one, the *Java* basic blocks are transformed to *LLVM* basic blocks. A basic block is a line of code sequence without any branches leading into it except the entry point and no branches leaving it except one exit branch [13]. Furthermore, the *Java* operand stack is modelled. Although primitive variable types, such as long and double, take up two slots in the *Java* operand stack respectively, the developers decided to only use one slot in the model. The operand stack is saved by the compiler at the end of each basic block. For compiling the *Java* methods, the compiler runs through the list of the basic blocks and sets the final operand stack of each block as initial information for its successor [27].

The developers of the *LLVM Java Frontend* claim that their compiler is able to handle all *Java* functionality and only lacks exception handling and garbage collection. The compiler uses the *GNU Class Path*, which is a free and clean implementation of the standard core class libraries for compilers and runtime environments for *Java*. As the *LLVM Java Frontend* is outdated (the last commit is from 07-29-2007), it was not possible to build it with the newest version of *LLVM* in the trunk (3.9), because a range of class files were missing. The latest *LLVM* version containing these class files was built (version 2.6). However, building the *LLVM Java Frontend* with this version lead to the error messages, which are depicted by listing 2.3. The missing types were removed from *LLVM* with version 1.9. As the *LLVM Java Frontend* is not build-able using a current version of *LLVM* and furthermore lacks the support of GC as well as exception handling it was not considered any further.

Listing 2.3: Error messages building *LLVM Java Frontend* with *LLVM* version 2.6

```
Compiler.cpp: ConstantSInt has not been declared
Compiler.cpp: IntTy is not a member of llvm::Type
Compiler.cpp: ConstantUInt has not been declared
Compiler.cpp: UByteTy is not a member of llvm::Type
```

2.3.2 GCJ

As already described in section 2.1.1, GCJ is one of the oldest and most sophisticated *Java* to native AOTCs. However, a very simple `HelloWorld` test program depicted in Listing 2.4, which was compiled using GCJ, produced an executable with a size of 15.8 KB. This has to be considered as a large file. A main goal of the thesis is to find a tool modifiable to compile *JavaCard* applications natively. Furthermore, the pre-compiled code has to be usable from within the JCOP OS. For this purpose, a range of changes would have to be performed on GCJ. As performing modifications on its compilation process would take an expert knowledge of its internal operations, this approach was not considered feasible for creating a compiler for an NXP platform and JCOP in the first place, even though GCJ is target-able on embedded systems as well [36].

Listing 2.4: `HelloWorld` test program

```
class HelloWorld {
        public static void main(String args[]) throws Exception
        {
                System.out.println(" *** bip bip *****");
        }
}
```

2.3.3 VMKit

The goal of the developers of the *VMKit* was to create an easy to use common substrate for creating and developing high-level *Managed Runtime Environments* (MRE). They created this development kit, as building a MRE such as the JVM is a very cumbersome task when considering all features, such as GC and JIT compilation. The *VMKit* provides a JITC, a memory manager and a thread manager. For the JIT-compilation *VMKit* makes use of the *LLVM* compiler and its intermediate representation. The *LLVM* compiler was chosen, as it does not impose any object model, type system or call semantics. Furthermore, it is able to generate very efficient code [10].

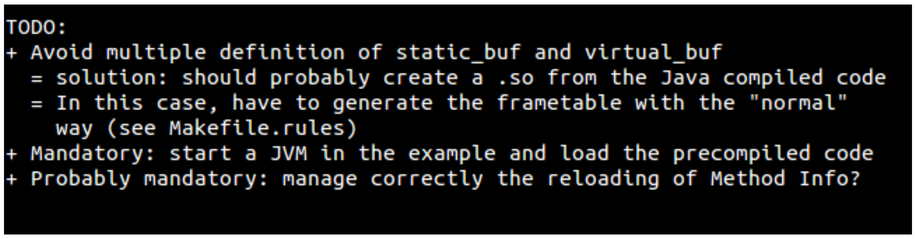
With this substrate, the developers were able to implement a high level MRE similar to JVM. This MRE is called *J3*. Besides this JITC, the *J3* also provides an AOTC for *Java*. The basic idea was to make use of this AOTC and try to compile a simple *Java* program to native code. If this first test was successful, we would proceed with customizing *VMKit* to our needs [10].

The first step was to download the latest version of *VMKit* [2]. This version builds with *LLVM* version 3.3. After building *VMKit*, the JIT-compilation was tried in a first attempt to verify that *VMKit* works as described. Therefore, the same `HelloWorld` test program as in section 2.3.2 was translated to *Java* byte code using the `javac` compiler. To be compatible with the latest version of *VMKit*, version 1.6 of *Java* was used. Afterwards, the emitted `.class` file was used as input to the `j3` VM. As this test was successful, the AOTC was tried. This compiler comes in form of the `llcj` program. A first attempt to AOT-compile the test program with the `llcj` failed. According to a mail conversation involving a developer of *VMKit* [3], the *llcj* is deprecated and not maintained any longer. The developer suggested invoking the programs, which would have been invoked by the *llcj* by hand. Therefore, the compilation scheme would be using the `javac` to create the *Java* byte code from the test program. This byte code would then be fed to the `vmjc` program resulting in a *LLVM* byte code representation of the test program. The *LLVM* byte code would then be linked using the `llc` linker to generate a native executable. However, this procedure lead to linking errors in the last step. The solutions to the arising problems mentioned by the *VMKit* developer during the e-mail conversation[2] were applied. Furthermore, the extra branch (`aot`) created by the developers of *VMKit* to solve the

[2]SVN repository of VMKit: `http://llvm.org/svn/llvm-project/vmkit/trunk/`

[3]See *[LLVMdev] VMKit is retired (but you can help if you want!)* [28]

issues in the AOT-compilation was checked out and built. However, using this version of the *VMKit* did not solve the errors. Also the example mentioned in the latest commit message, which is depicted by Figure 2.12, did not AOT-compile the *Java* test program. The latest commit messages in the SVN repository shows a `TODO` list which does not indicate that the AOT-compilation process was fixed by the developers.



```
TODO:
+ Avoid multiple definition of static_buf and virtual_buf
  = solution: should probably create a .so from the Java compiled code
  = In this case, have to generate the frametable with the "normal"
    way (see Makefile.rules)
+ Mandatory: start a JVM in the example and load the precompiled code
+ Probably mandatory: manage correctly the reloading of Method Info?
```

Figure 2.12: Latest commit message in `aot` branch

Although the latest changes made to the project's `aot` are quite recent (September 2014), this version does not provide a stable AOT-compilation as well. Therefore, this project was not considered any further, as the effort of bringing the AOT-compilation to a usable state was considered too high for utilizing it as a basis for further modifications targeting embedded systems.

2.3.4 RoboVM

RoboVM is a commercial framework for implementing truly native applications for both *Apple's iOS* as well as *Android*. Of this framework, the IDE can be purchased. However, the underlying software is free to use. This framework also provides an AOTC. The compiler was tried using the `HelloWorld` test program [34]. The pre-built version of *RoboVM* can be downloaded from the `maven` repository[4]. After adding the needed libraries and the `HelloWorld.class` file to the `robovm-dist-compiler-1.8.0.jar`, the program was invoked by invoking it with the following command:

```
$ java -jar robovm-dist-compiler-1.8.0.jar -cp . -home .. HelloWorld
```

This triggers the compilation of the given `.class` file. As soon as the output folder (`HelloWorld`) is created, this operation can be aborted. The result of this compilation process is an executable `HelloWorld` program with a size of 10.7 MB. Such an executable would be too big to be put on an NXP platform, on which already a JCOP OS is deployed. Thus, and because *RoboVM* aims on *iOS* and *Android* development, it was not considered as basis for a *JavaCard* compiler framework.

2.3.5 XMLVM

The *XMLVM* project is a flexible cross compilation framework, which aims at transforming byte code instructions of two widely used VMs, the JVM and the Common Language Runtime (CLR), which is part of the .NET framework. The main goal of this framework is to translate *Android* projects to *iOS* applications and vice versa. Therefore, it is capable

[4]http://mvnrepository.com/artifact/org.robovm/robovm-dist-compiler/1.8.0

of transforming byte code in both directions. *Java* byte code instructions can be translated to *CLR* byte code instructions as well as *CLR* byte code instructions to *Java* byte code instructions. To achieve this, the *XMLVM* project translates the single byte code instructions into XML tags and utilizes XSL stylesheets for cross-compilation. The first step of the compilation process is transforming the given byte code into the according XML tags. *Java* byte code is transformed to $\text{XMLVM}_{\text{JVM}}$ tags and *CLR* byte code is transformed to $\text{XMLVM}_{\text{CLR}}$ tags. To compile $\text{XMLVM}_{\text{CLR}}$ to $\text{XMLVM}_{\text{JVM}}$, a special intermediate data flow analysis format, $\text{XMLVM}_{\text{DFA}}$ is used. Figure 2.13 depicts the possibilities of cross compilation with *XMLVM* [46].

The $\text{XMLVM}_{\text{JVM}}$ format can be seen as an internal IR forming the connection between *XMLVM*'s frontend and backend, as it allows the translation from the byte code representation to various other programming languages, but also to $\text{XMLVM}_{\text{CLR}}$. Not depicted by Figure 2.13 is the possibility to cross compile *Java* byte code to *C* files [46].

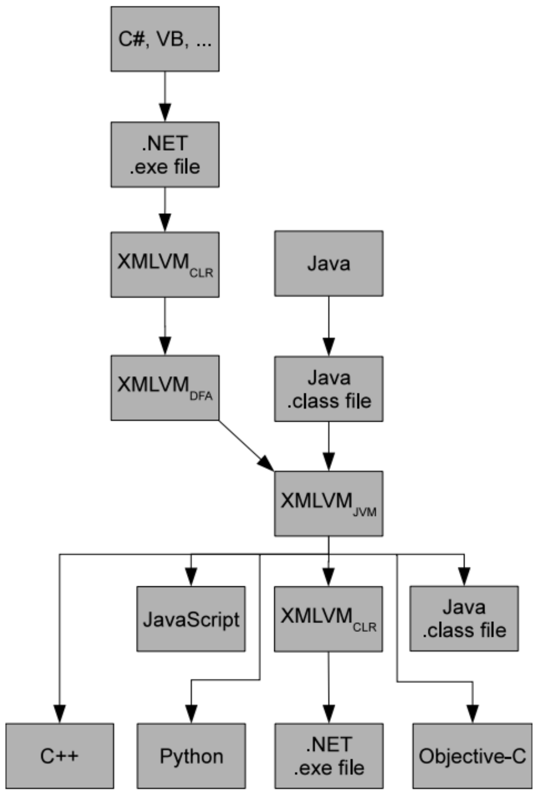


Figure 2.13: Cross-compilation flow of *XMLVM* framework [46]

Additionally to the byte code translation, the *XMLVM* framework offers a range of compatibility libraries. With these libraries, the framework allows translation from ,e.g., *C#* desktop applications using *Windows Forms* graphical interface to a *Java* application by utilizing these compatibility libraries written in the *Java* [46].

The *XMLVM* framework can be used with a range of commands selecting the target to which a provided program should be translated. The framework recognizes the programming language of the input files by investigating the file' suffices and translates them to the chosen target. Of these available targets, the `posix` target is the most interesting one for testing *XMLVM*'s capability of translating *Java* applications to an executable *C* program. *XMLVM*'s documentation describes the `posix` target as follows:

"`posix`: The input files are cross-compiled to a self-contained *C* program that includes all dependent classes such as `java.lang.System`." [45].

As it was uncertain how the *XMLVM* framework would translate a *Java* program, a very simple *Java* test program was written containing only one `main` function, in which two `int` variables are added. The test class was added to a specific `xmlvm.test` package and put into a corresponding folder structure. The *XMLVM* framework was then called with the following command:

```
$ xmlvm.jar --in=./in/xmlvm/test/ --out=./out --target=posix
```

The `--in` option must be targeting the folder, in which the *Java* class files to be translated are located. Therefore, the *Java* test program has to be compiled to byte code before. This process has the effect that in the folder targeted by the `--out` option a `src` and a `dist` folder is created. In the `dist` folder only a `makefile` and an empty `build` folder is created. In the `src` folder, all the *Java* class files, on which the test program depends, are transformed to *C* header and source files. Furthermore, the *XMLVM* related source files are put into this folder. To verify, if the translated code would not only build, but also execute, a simple `printf()` has been added to the translated main function, which is located in the `xmlvm_test_HelloWorld.c` file, to print out the following char sequence: `"Hello world!\n"`.

Before being able to build the translated project, the *Boehm-Demers-Weiser conservative garbage collector* (`bdwgc`) has to be downloaded and built. After this last prerequisite, the translated *C* program can be built simply by using the `makefile`. This leads to the *C* files being compiled and linked to create an executable called `out`. Running this executable produces the output depicted by Listing 2.5 on the console.

Listing 2.5: Console output of executable compiled from *XMLVM* translated `HelloWorld` test program

```
out/dist/build/$ ./out
out/dist/build/$ this is a test!
out/dist/build/$
```

The created executable has a size of 12.8 MB. The huge size of the executable is largely caused by converting the *Java* runtime and all the classes from multiple libraries (such as `jaxp.jar`, `harmony6-build.jar`, etc.) it depends on.

Although the executable's size is the largest compared to the other projects, with which a compilation of a *Java* test program was achieved, the way in which *XMLVM* translates the single *Java* class files is very transparent. It is clear what *Java* class files were added to the translation process and therefore, it was assumed that downsizing the amount of translated *Java* classes would be achievable. Furthermore, a look at the source code of the framework revealed a well structured design as well as a detailed in-line documentation. Because of these factors, the *XMLVM* framework was chosen as basis for further modifications to adapt the translated code to be executable on an NXP platform.

2.4 Conclusion

Both literature and practical research yielded a range of interesting projects that gave certain inputs to both the design and the implementation of the *Java Card* to ARM framework. Unfortunately, most of the open source projects examined during the practical research did not fulfil their developers' promises. Only the *XMLVM* and the *RoboVM* project were able to compile a small *Java* project to an executable on x86 out of the box. As the *RoboVM* was not build-able from its source code, the *XMLVM* project was considered as a basis for further adaptations to natively compile a *Java* test program and later a *JavaCard* application for ARM. This decision was largely influenced by *XMLVM*'s well structured and documented source code as well as its transparency considering what *Java* classes are translated and further compiled and linked to the final executable.

Chapter 3

Design of the Framework

The process for creating the JCF on basis of the XMLVM project is divided in several steps. The final goal of this modification process is to develop a JCF capable of compiling a *Java Card* applet along with the *Java Card* framework, natively utilizing the JCOP HAL layer and parts of the JCOP OS written in *Java Card*. These steps are described in detail in section 3.2

The individual steps of the creation process of the JCF aim at different hardware platforms. These platforms are described in detail in section 3.1.

It must be pointed out that the JCF is meant as a proof of concept to show that it is possible to AOT-compile a *Java Card* applet for an NXP platform and link, as well as execute it utilizing the already existing infrastructure of JCOP. Therefore, the JCF may not be seen as a complete framework. Furthermore, the JCF lacks GC and other features which would be necessary to develop the framework into a readily shippable product. As the XMLVM project is a standalone AOTC, the JCF is also designed to be an standalone compiler. This implies that the final JCF is not designed to interpret or JIT-compile *Java Card* byte code or to integrate partially AOT-compiled code into interpreted byte code. Therefore, interaction with a JCVM is not possible with the current state of the framework.

3.1 Hardware Setup

The single steps of the creation process of the JCF aim at different hardware platforms. These platforms are:

- Desktop computer
  - The desktop computer used as hardware platform runs Ubuntu 14.04 as OS in 32 bit version. This desktop computer is equipped with an Intel i5 processor and 4 Gbytes of RAM
- STM32F407VG board
  - The hardware platform used for intermediate evaluation is the STM32F407VG [35] board. This board is equipped with the following hardware:
    - Core: ARM® 32-bit ARM ® -M4 CPU

    * RAM Memory: 192 Kbytes
    * FLASH Memory: 1 Mbyte
    * Debug Mode: Serial wire debug (SWD) & JTAG interfaces
    * Communication Interface: Up to 4 USARTs/2 UARTs (10.5 Mbit/s, ISO 7816 interface, LIN, IrDA, modem control)

- A current Smart Card Chip of NXP Semiconductors Austria GmbH
  - This is the final hardware platform on which the natively compiled *Java Card* applet will run
    * Core: ARM® 32-bit ARM ® SC300 CPU
    * RAM Memory: 52 Kbytes
    * FLASH Memory: 2 Mbyte
    * Debug Mode: ULINK2/ME Cortex Debugger
    * Communication Interface:DMA controller supporting UART

3.2 Framework Modification steps

The adaptation of the XMLVM project consists of four steps. Each step in this chain can be seen as prerequisite of its successor step. These adaptations aim at creating a framework capable of compiling a *Java Card* applet on an NXP platform. The single steps are described in the following sections.

3.2.1 XMLVM Capabilities and Minimization of used Java Runtime

The modification process starts with examining the XMLVM project's capabilities of compiling a *Java* program. Preceding this step, the conversion tool is already tested on basic *Java* operations, such as simple integer arithmetic, boolean logic, object instantiation and bitwise operators. Therefore, in this first step a test project is created which already focused on the more complex concepts which *Java* but also *Java Card* offers. These contain features such as polymorphism, exception handling, native method invocation, etc. A more detailed list is depicted below:

- Inheritance
  - Interface extending one or multiple interfaces
  - (Abstract) classes extending one or multiple (abstract) classes
- Method overloading
- Arrays
- Native method invocation
- Exception handling
  - Exception thrown and caught in the same method

  - Exception thrown in called method and caught in calling method

- Instance of
  - Checking, if derived class is from type of base class
  - Checking, if base class is from type of derived class

Features such as reflection or threading which are only supported by *Java* but not *Java Card* are deliberately omitted. The project basically consists of three test cases. One test case which verifies that `exceptions` can be thrown and caught in the same method and in a calling method. One test case checks if a `DerivedClass` extending a `BaseClass` is correctly recognized as an instance of the `BaseClass`, but also if `DerivedClass` casted to a `BaseClass` is recognized as an instance of the `DerivedClass`. This same test is performed on a `BaseClass` implementing a simple `Interface`. The last test case tests if the `DerivedClass` correctly overloads virtual methods of the `BaseClass` and the `Interface`. This test project is depicted in Figure 3.1 as a UML class diagram.

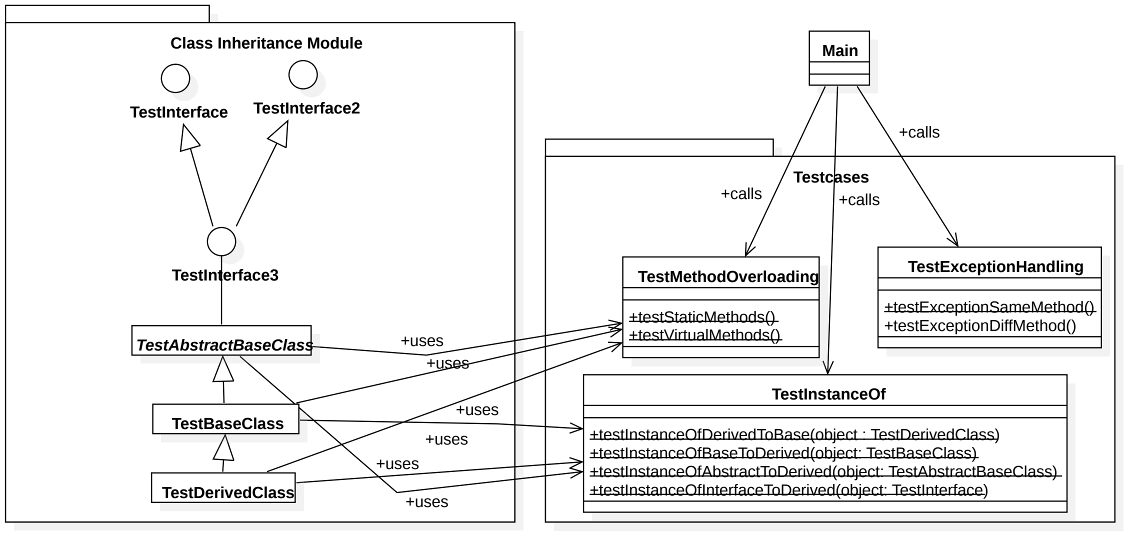

Class Inheritance Module
TestInterface
TestInterface2
TestInterface3
TestAbstractBaseClass
TestBaseClass
TestDerivedClass
+uses
Main
+calls
Testcases
TestMethodOverloading
+testStaticMethods()
+testVirtualMethods()
TestExceptionHandling
+testExceptionSameMethod()
+testExceptionDiffMethod()
TestInstanceOf
+testInstanceOfDerivedToBase(object : TestDerivedClass)
+testInstanceOfBaseToDerived(object: TestBaseClass)
+testInstanceOfAbstractToDerived(object: TestAbstractBaseClass)
+testInstanceOfInterfaceToDerived(object: TestInterface)

Figure 3.1: Simplified class diagram of test project

Furthermore, as already mentioned in section 2.3.5, the amount of the *Java* classes converted by the framework must be cut down lavishly. The goal is that at the end of the minimization process the framework should only convert the following classes:

- From the *Java* Runtime
  - `java.lang.Object`
  - `java.lang.String`
  - `java.lang.Throwable`
  - `java.lang.Exception`
- From the *XMLVM* compatibility library

 – `org.xmlvm.XmlvmClass`
 – `org.xmlvm.XmlvmArray`

The `java.lang.String` class will only be used for debugging reasons and exclusively in the first steps of the modification process. Furthermore, also the classes of the test project must be converted. As the test project is comparatively, small the utilized GC, the *Boehm-Demers-Weiser conservative garbage collection* utilized by the XMLVM framework is also excluded from the conversion process.

The C standard library is used for providing the basic functionality and to interact with the OS for memory management, input/output processing, etc. For debugging the test project a native function utilizing the standard C `printf` function is used.

To summarize the first step of the modification process, refer to Figure 3.2. It shows the planned process of minimizing the size of the converted test project by constantly verifying the correct representation of the *Java Card* features, by checking the correct execution of the test project. The first action is to remove the reflection and the multithreading features. Afterwards, the minimization of the produced executable is targeted.

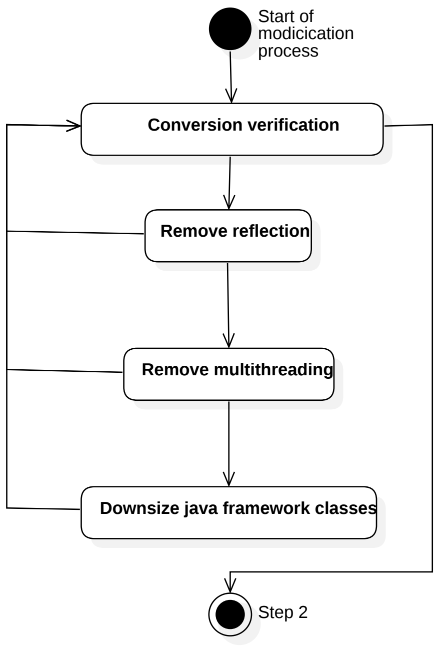


Figure 3.2: Step 1 modification flow

To explain the layout of the produced *C* project better, Figure 3.3 is utilized.On the left side, it shows the original layout of the code generated by the unmodified XMLVM framework. The top layer is formed by the *TestProject* supported by the converted classes of the *Java* runtime and certain compatibility classes provided by XMLVM. The native functions are implemented in the *native compat lib*. The *XMLVM Framework* provides additional functionality and contains the entry point of the executable. Beneath this framework layer, the standard *C* library utilizing the *Posix* system provides basic functionality. The lowest layer already displays the hardware memory type being utilized. As can be observed, in the first step only *Java* runtime classes and the XMLVM compatibility classes, as well as the XMLVM framework, are adapted. The *Java* runtime classes and the XMLVM compatibility classes are changed for minimizing the executable. The XMLVM framework is used for removing multithreading and reflection.

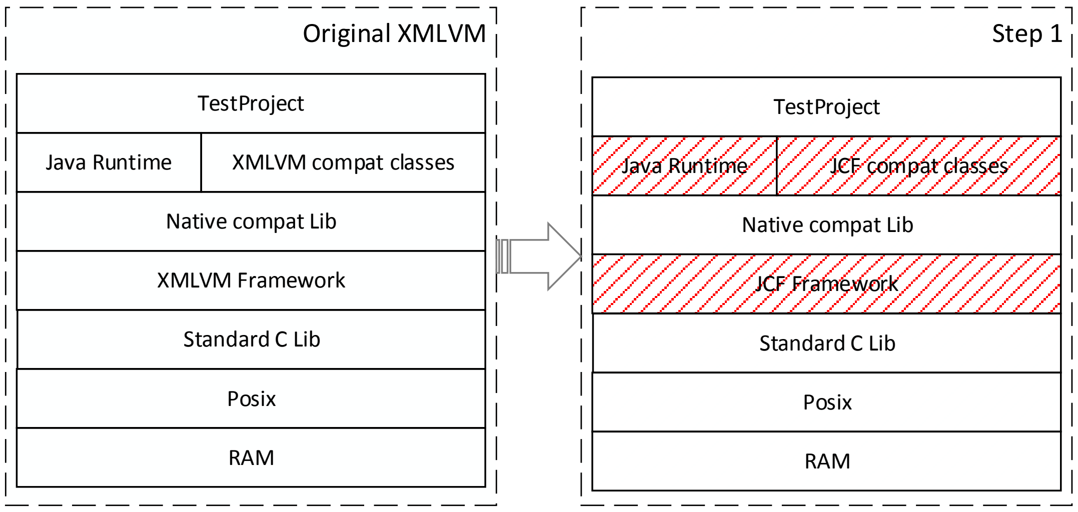


Figure 3.3: Changes in Step 1 to project layout

3.2.2 Boarding on STM32 and Modification of Memory Allocation

After verifying that *XMLVM* is able to correctly translate the necessary features, shared by *Java* and *Java Card*, and minimizing the amount of converted *Java* Runtime classes, the next step is to move the resulting code to a hardware platform more similar to the NXP platform. This hardware platform is the STM32F407VG board, which from the hardware perspective, is similar to the NXP platform.

The second goal of this step is to change the way objects and static class fields are allocated by the resulting program. As already described in section 1.3.3, one of the main differences between *Java* and *Java Card* is its object lifecycle. As *Java Card* allocates objects and class fields on non-volatile memory (FLASH), it must be ensured that these structures are retrievable after a card reset.

In the converted program, classes and objects are represented by dedicated structures, static class fields are transformed into global variables. Therefore, in this step the object and class structures as well as the static class fields are no longer allocated in the stack, but instead a special array simulating the FLASH memory is used. This also includes fields holding objects which are translated into global pointer variables and assigned with the address of the object structure. This array is still placed in the RAM at this point of the modification process. Another big difference is that assignments of variables located on the FLASH memory must be performed by special memory writing operations.

Although in this step the FLASH memory is only simulated, the memory allocation as well as the write operations, performed on the allocated memory, is performed by using functions similar to those provided by the JCOP HAL on the targeting system. These operations are encapsulated by interface functions which guarantee that in the further steps the same interface can be used and only the underlying functions must be replaced with those implemented in the JCOP HAL.

To identify the changes which have to be applied to the conversion process, a test case is added to the test project. This test case covers the following operations:

- Static class field allocation
- Object allocation
- Assignments:
  - Assigning local variable to static class field
  - Assigning static class field to local variable
  - Assigning local variable to object field
  - Assigning object field to local variable

Figure 3.4 shows the extended test project.

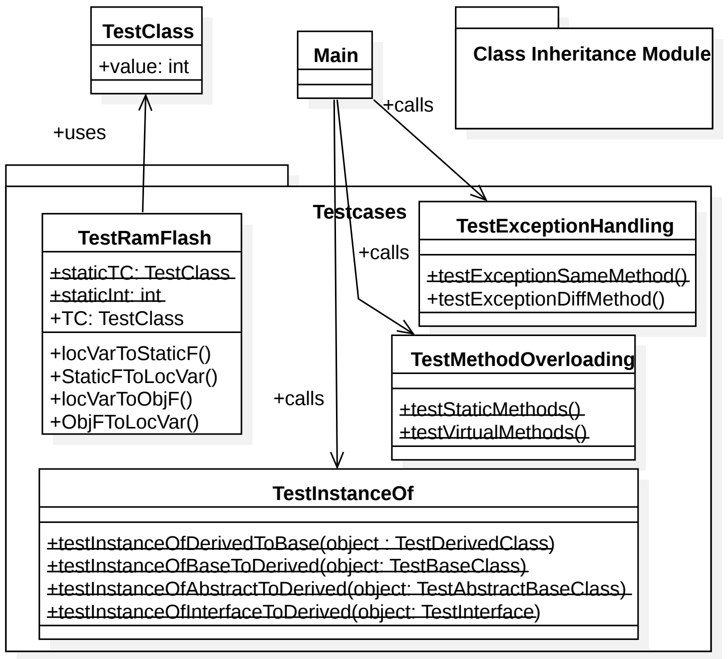


Figure 3.4: Test project extended with `TestRamFlash` test case

To become more independent from the C standard library concerning the memory management, an own simplified memory management is implemented, covering the processes of allocating memory and copying values from a given memory address to a targeted memory address. Furthermore, instead of using the standard C `printf` function the debug information is emitted by using the UART module of the STM32F407VG. The UART module is accessed by means of functionality provided by the STM32 HAL.

The main actions in this step are to replace the used functions of the C standard library by implementing JCF proprietary functions and to change the object allocation from the Stack to the FLASH simulating global array. This process is visualized by Figure 3.5.

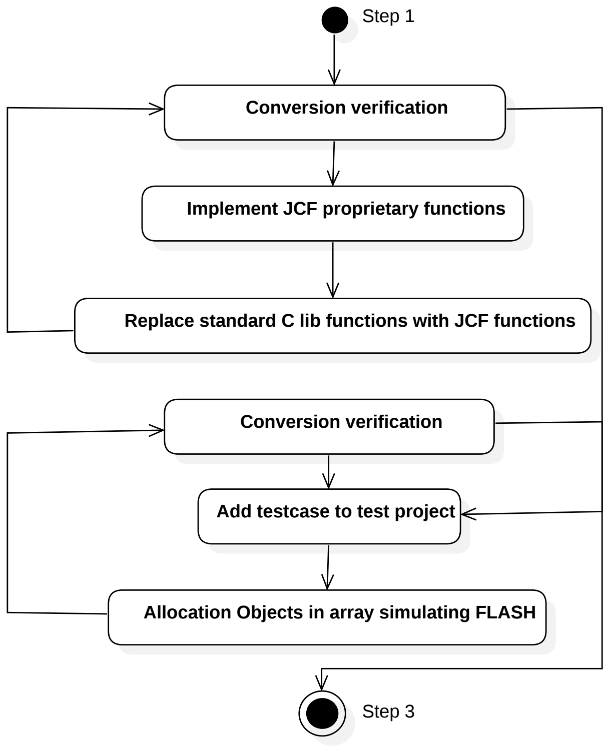

Step 1
Conversion verification
Implement JCF proprietary functions
Replace standard C lib functions with JCF functions
Conversion verification
Add testcase to test project
Allocation Objects in array simulating FLASH
Step 3

Figure 3.5: Step 2 modification flow

Figure 3.6 depicts the changes to the project layout that need to be performed in this step. The modifications are performed in the lower layers of the produced *C* code. The standard *C* library is replaced by a JCF proprietary library that utilizes the STM32 HAL functionality instead of the *Posix* system. Furthermore, the class and object structures, as well as the static class fields, are placed in a dedicated array situated in the RAM.

3.2.3 Boarding to the NXP Chip and Adaptation of Object Lifecycle

The next step of the modification process is to transfer the converted test project to the NXP platform. To finalize the adaptation of the object lifecycle, the memory area in which the object and class structures are allocated is shifted from RAM to FLASH. To be more precise, the structures representing the single object as well as the structures representing the *Java* classes, are allocated in FLASH. Furthermore, all class fields are put into persistent memory, as well.

To ensure that once assigned static class fields referencing an object are not newly assigned after a card reset, the conversion process is modified in such a way that a null pointer check is performed which prevents overwriting the field. Furthermore, to guarantee that all allocated objects are accessible after the card is reset, a so called *mother object*

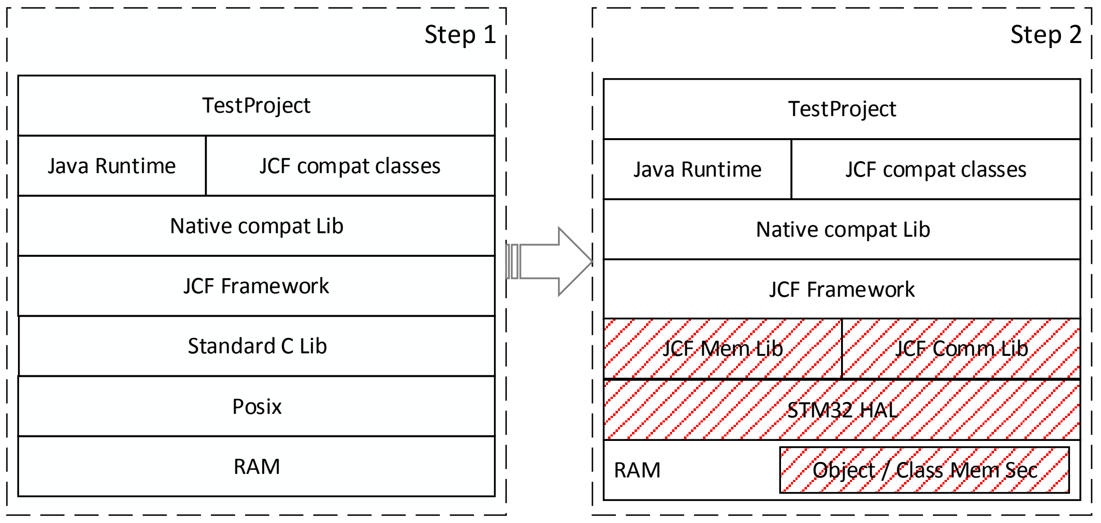


Figure 3.6: Changes to project layout performed in Step 2

is allocated at the first execution of the program. The *mother object* will simply be a reference to the test project. From this reference the object tree is spanned.

Additionally to these modifications to the object lifecycle, also the underlying layer of the infrastructure is changed in respect to STM32F407VG. To make use of the JCOP HAL running on an NXP platform, it is initialized in the same way the JCOP OS would initialize it. But instead of booting the JCOP OS, our converted test project is called directly. To correctly write to FLASH, the memory copy function provided by the JCOP HAL is used. The utilization of the JCOP HAL memory functionality is simplified by the interface implemented in Step 2. Furthermore, also the usage of the UART is adapted to the JCOP HAL.

A new test case is added which is used for testing whether the static class fields and object structures are correctly allocated and retrieved after a card reset. This test consists of a class with an integer counter as its class field and a test case holding an instance of this class as well as an additional integer counter. The counters are increased during each execution of the test case. After a card reset both counters must still hold the value they were set to in the preceding execution. Figure 3.7 depicts the test project with this new test case.

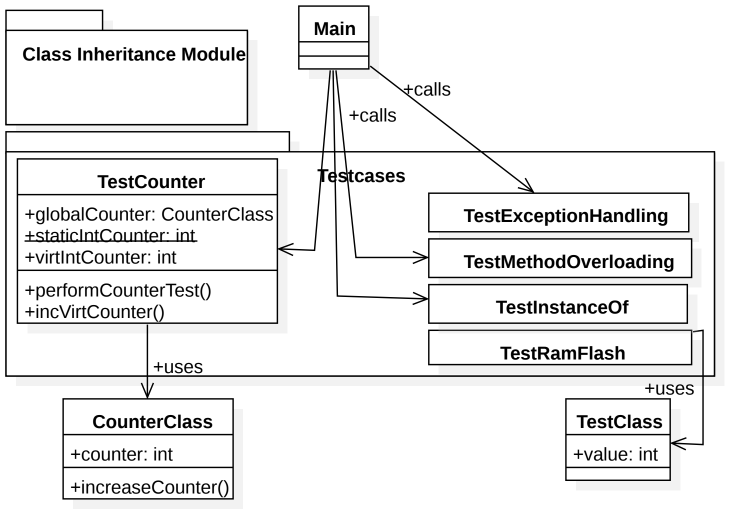


Figure 3.7: Test project extended with `TestCounter` test case

In this step the change from the modified *Java* Runtime to the *Java Card* Runtime is prepared as well. This change is performed by adapting the conversion process in such a way that it does not use the `java.lang.String` class any longer and by replacing all strings with `char` arrays in the test project.

In Figure 3.8 the single actions for the adaptations in this step are pictured. Firstly, the JCOP HAL functionality must be initialized. Secondly, the JCF proprietary functions, implemented in the step before, must make use of the functionality provided by the JCOP HAL. The array, which simulates the FLASH memory section for allocating the objects, the class representations and the static fields, must be put into real FLASH memory. Thereafter, a simple reference, called *mother object*, must be implemented and pointed at the test project. The last action is to implement a further test case as described above.

Figure 3.9 visualizes these modifications based on the layout of the generated *C* project. In this step the adaptations only regard the initialization and utilization of the JCOP HAL, which replaces the HAL of the STM32. Furthermore, the project uses, additionally to the RAM, the FLASH memory for persistently allocating objects, class structures and static class fields.

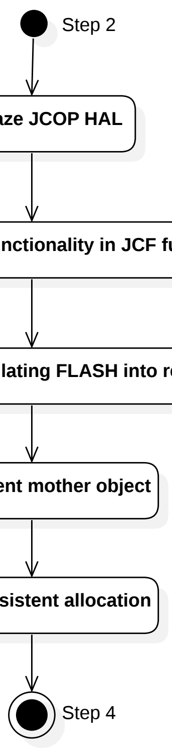

Step 2
Initiliaze JCOP HAL
Use JCOP HAL functionality in JCF functions
Transfer array simulating FLASH into real FLASH
Implement mother object
Test persistent allocation
Step 4

Figure 3.8: Step 3 modification flow

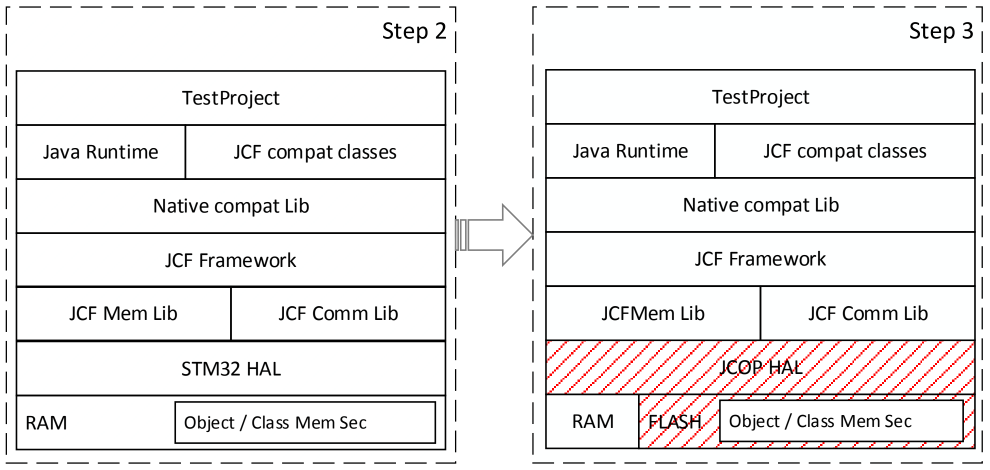


Figure 3.9: Changes to project layout performed in Step 3

3.2.4 Java Card Runtime/Framework API and Applet

After verifying that the JCF is able to convert the test project to a C program, which complies to the *Java Card* object life cycle and is executable on an NXP platform, the *Java* framework is replaced by the *Java Card* framework. Considering this replacement, it is important to disentangle the *Java Card* framework from the *Global Platform* (GP) framework, as only the *Java Card* framework is sufficient for running an applet. If necessary, calls from the *Java Card* framework to GP will be replaced by stubs. Thus the outside dependencies to GP will be cut.

As an applet utilizes classes which are not provided by the *Java Card* framework, but also by JCOP OS layer, these are converted by the JCF as well. Both the *Java Card* framework as well as the JCOP OS classes heavily rely on methods natively implemented in C. These methods, which are usually provided by the JCOP VM, must now be implemented and provided by the JCF.

Furthermore, in this step the test project is replaced by using a test applet. Therefore, the *mother object* must now reference the applet instead of the test project, as in the steps before. To run the applet in the same way as it would be executed by the JCOP VM, an additional layer is developed. This layer (the *Java Card* OS) is placed between the applet and the underlying *Java Card* framework. The main task of this *Java Card* OS is to:

- receive and dispatch incoming APDUs.
- initialize all data structures and components which are needed by the applet and the JCOP OS classes.
- install the applet.
- provide the applet with the received APDU and run the applet.
- send response APDU.

To execute these tasks it is necessary that the *mother object* not only holds a reference to the applet but also to the APDU object. This object is then provided to the applet during its execution. Furthermore, a method in the *Java Card* OS (JCOS) is necessary for running the applet referenced by the *mother object.* The applet's execution is started by calling a dedicated method from the JCOS. At the end of its execution an appropriate response APDU is created according to the applet's behaviour. Furthermore, the JCOS must make the needed structures the JCOP HAL available to the applet. This is especially relevant for the native APDU structure.

Furthermore, the JCOS must initialize the HAL of the JCOP subsystem. After this initialization, it listens for an incoming APDU and provides it to the applet. The applet processes this APDU and executes its implemented tasks. According to the applet's behaviour the response APDU is created and sent back. Then the JCOS continues listening for the next incoming APDU.

As the creation of new objects during the execution of the applet is omitted, exceptions are created as singletons during the initialization of the JCOS. Furthermore, the JCOS must provide the possibility to create transient arrays. The data of these arrays are stored in transient memory. This functionality is also provided by the class. These two new features are tested by two additional test cases. Additionally, a test case is added for testing receiving and sending of APDUs.

The test cases from the predecessor steps are still maintained. However, instead of sending debugging information via the UART, the test applet sends response APDU containing information about the result of the single test cases. Figure 3.10 shows the test applet containing the already existing test cases and the new ones. The test project is replaced by the test applet to run the single test cases.

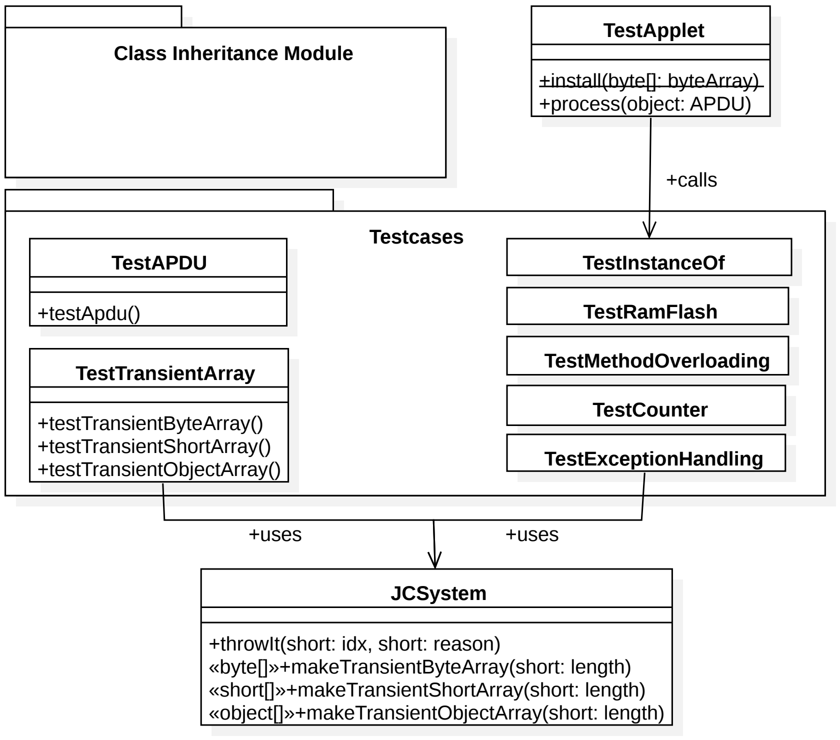

Class Inheritance Module
TestApplet
~~+install(byte[]: byteArray)~~
+process(object: APDU)
+calls
Testcases
TestAPDU
+testApdu()
TestTransientArray
+testTransientByteArray()
+testTransientShortArray()
+testTransientObjectArray()
TestInstanceOf
TestRamFlash
TestMethodOverloading
TestCounter
TestExceptionHandling
+uses
+uses
JCSystem
+throwIt(short: idx, short: reason)
«byte[]»+makeTransientByteArray(short: length)
«short[]»+makeTransientShortArray(short: length)
«object[]»+makeTransientObjectArray(short: length)

Figure 3.10: Test applet extended with test case for transient array and adapted exception handling test case

In Figure 3.11 this final step's actions are depicted in a flow diagram. As a first action the GP dependencies are removed from the single *Java Card* framework classes. The *Java* framework is then replaced by this *Java Card* framework classes. Furthermore, also the relevant JCOP classes are added to the conversion process. The produced executable relies on the implementation of native *C* functions. The utilized ones are implemented, the unused ones are stubbed. Before replacing the test project with a real *Java Card* applet, the JCOS is implemented. To achieve this goal also the mother object must be extended as described above. As soon as these targets are achieved, the test project is replaced with a test applet. This last step is tested by implementing a new test case verifying correct APDU handling by both the JCOS as well as the test applet.

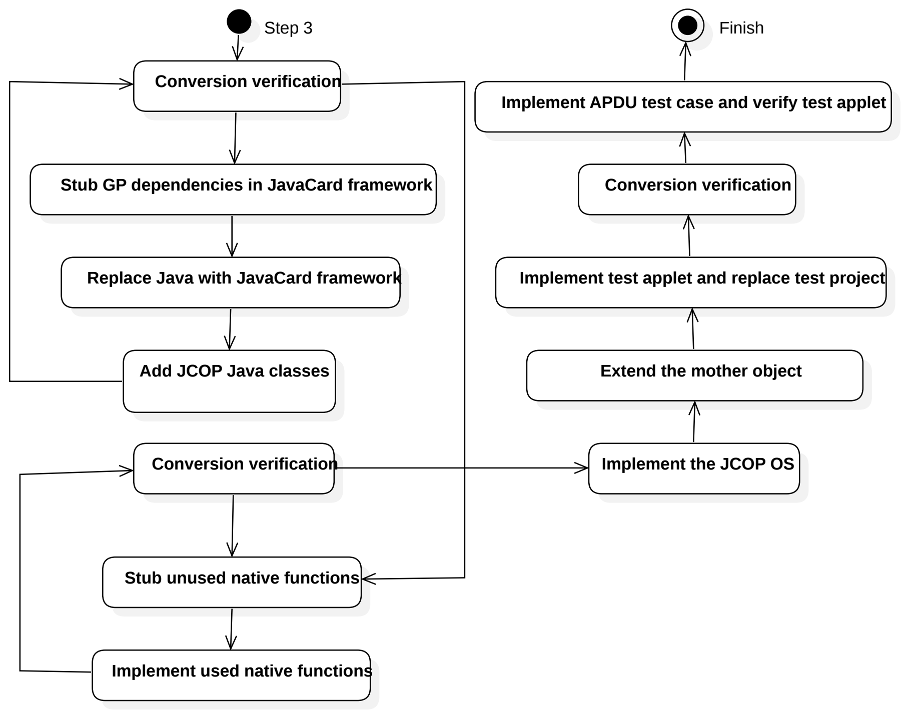

Step 3
Conversion verification
Stub GP dependencies in JavaCard framework
Replace Java with JavaCard framework
Add JCOP Java classes
Conversion verification
Stub unused native functions
Implement used native functions
Implement the JCOP OS
Extend the mother object
Implement test applet and replace test project
Conversion verification
Implement APDU test case and verify test applet
Finish

Figure 3.11: Step 4 modification flow

The impact of the last step's modifications on the single layers of the generated project are visualized in Figure 3.12. In this step a new layer is introduced representing the JCOS. Furthermore, the *TestProject* is replaced by the *TestApplet* and the *Java* by the *Java Card* runtime classes. Additionally, the native functions needed by the *Java Card* runtime must be implemented in the former *native compat lib* layer.

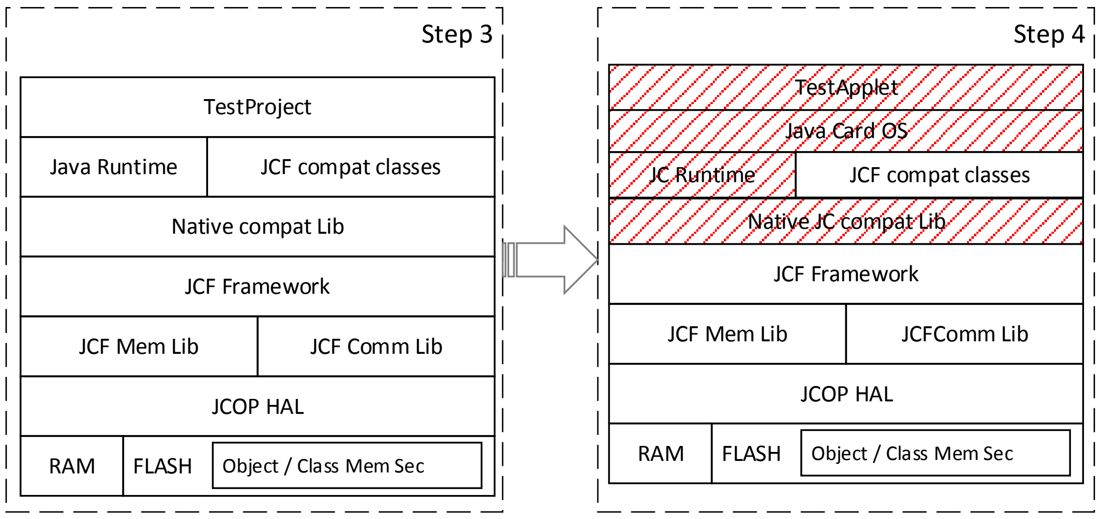


Figure 3.12: Changes to project layout performed in Step 4

3.3 Conclusion

With the process described in this chapter and its division into the presented steps, the final goal of natively compiling a *Java Card* applet along with the *Java Card* framework and executing it on an NXP platform is achieved. Each step makes sure that the already defined requirements and the features tested in its predecessor step are still fulfilled and maintained. With the final step a small applet is natively converted, on which a range of test cases is performed. This last step does not induce that any arbitrary applet is convertible using the JCF, as the missing native functionality must be implemented first.

Chapter 4

Implementation of the Framework

In this chapter more details about the XMLVM framework and about the modifications in the single steps of the adaptation process are presented. In Section 4.1 the conversion process of the XMLVM framework is described. In Section 4.2 the details of the adaptations performed in the steps described in Chapter 3 are explained. Section 4.3 gives an overview about the whole conversion process of the created Java Card Cross-Compilation Framework.

4.1 XMLVM Conversion Process

The Java Card Cross-Compilation Framework designed and implemented in this thesis is based on the XMLVM framework. To understand the adaptations performed for reaching a compiler framework which is able to compile *Java Card* code to native machine instructions targeting embedded systems, the XMLVM conversion process must be understood first. Certain *Java* features supported by the XMLVM framework such as reflection and multi-threading are not explained as these features are omitted completely by the Java Card Cross-Compilation Framework as *Java Card* does not support them [18].

4.1.1 Intermediate Representation

The conversion process of XMLVM is split into three parts. In the first phase the XMLVM framework converts the *Java* byte code into a *dex* format. This format is used by the *Dalvik Virtual Machine* (DVM) which is running on the *Android* platform. The byte code used in the DVM differs in several aspects from the byte code used by the JVM but the biggest differences are to be found in the register structure and the instruction set.

As the DVM is not a stack-based VM, such as the JVM, but rather a register-based one, the local variables of a method are assigned to one of the 2^{16} registers. These virtual registers are type-less. Because of the DVM working with type-less virtual register rather than manipulating the local variables on the *Java* stack, which is the case with the JVM, the op-codes in DVM's instruction set also differ from those used by the JVM [4].This pre-conversion from *Java* byte code to the *dex* format makes the further conversion to *C* code easier, as the machine model and the calling conventions imitate real architectures and the standard *C* calling convention [6].

In the second part the *dexed Java* byte code is represented by the XMLVM proprietary IR. For this intermediate step the *EXtensible Markup Language* (XML) is used. This language was designed for being machine and human readable as well as for storing and transporting data [41]. Each *Java* class is represented by a XML file. Listing 4.1 depicts a *Java* class (`TestClass.java`) which is converted. This class contains a final static field, a static field and a virtual field. It extends a class and overrides one of its methods. It is itself extended by another *Java* class. Furthermore it contains a static method in which an `Exception` is thrown and caught. In addition, it implements an interface.

Listing 4.1: TestClass which is converted to C using XMLVM

```java
package test.doc;

public class TestClass extends TestBaseClass implements TestInterface
{
  public byte virtualField = 0;
  public static byte staticField = 0;
  public final static byte finalField = 0;

  public static void staticMethod(byte b)
  {
    try
    {
      staticField = b;
      throw new Exception();
    }
    catch(Exception e)
    {
      staticField = 1;
    }
  }

  public void virtualMethod(byte b)
  {
    virtualField = b;
  }

  private void privateMethod(byte b)
  {
    virtualField = b;
  }

  protected void protectedMethod(byte b)
  {
    virtualField = b;
  }
}
```

The XMLVM proprietary IR of this class can be seen in the Listings 4.2 and 4.3. Unnecessary information is omitted in this type of representation with "...". Listing 4.2 shows how a *Java* method is represented in IR. Each method contains information about its accessibility, its name, its signature and if it is static. The signature contains the parameters as well as the return type. The body of the method contains the single byte code instructions. In this example the `byte` parameter is assigned to a local register which is then assigned to the `virtualField` field of the `TestClass`.

Listing 4.2: IR of *Java* method

```
<vm:method name="virtualMethod" signature="(B)V" isPublic="true">
  <vm:signature>
    <vm:parameter type="byte" />
    <vm:return type="void" />
  </vm:signature>
  <dex:code register-size="1">
    <dex:var name="var-register-1" register="1" param-index="0" type="byte" />
    <dex:iput-byte "..." type="byte" name="virtualField" vx="1" vx-type="int"/>
    <dex:return-void />
  </dex:code>
</vm:method>
```

Listing 4.3 shows how a *Java* class is typically represented in the IR. Each class representation contains information about the class name, its package, its accessibility, the class it extends and the interfaces it implements. Furthermore, it contains an entry for each field holding information about its name, its type, its accessibility and if it is *static* or not. It does not store any information whether a field is *final*. At the end of the IR the referenced *Java* classes and their usage within the converted class are listed. Each class contains two dedicated methods. The `<init>` method initializes all static fields. The `<clinit>` method initializes all virtual fields and is called when a new instance of the class is allocated.

Listing 4.3: IR of *Java* class

```
<vm:class name="TestClass" package="test.doc" extends="test.doc.TestBaseClass"
  isPublic="true" interfaces="test.doc.TestInterface">
  <vm:field name="virtualField" type="byte" isPublic="true" />
  <vm:field name="staticField" type="byte" isStatic="true" isPublic="true" />
  <vm:field name="finalField" type="byte" isStatic="true" isPublic="true" />
  <vm:method name="<init>" signature="()V" isPublic="true">
    "..."
  </vm:method>
  <vm:method name="<clinit>" signature="()V" isStatic="true">
    "..."
  </vm:method>
</vm:class>
<vm:references>
  <vm:reference name="java.lang.Exception" kind="usage" />
  <vm:reference name="test.doc.TestBaseClass" kind="super" />
  <vm:reference name="test.doc.TestClass" kind="self" />
  <vm:reference name="test.doc.TestInterface" kind="interface" />
</vm:references>
```

The XMLVM framework is furthermore capable of correctly translating *Java* exception handling. The information needed for converting a `try - catch` block to *C* is also contained in the IR. In Listing 4.4 the IR of the exception handling performed in the `staticMethod` of the `TestClass` is depicted. An important detail worth mentioning is the `target` value in the `dex:catch` tag which indicates to which label to jump to, if no exception was thrown.

Listing 4.4: IR of Exception Handling

```
<dex:catches>
  <dex:entry start="0" end="8">
    <dex:handler type="java.lang.Exception" target="8" />
  </dex:entry>
</dex:catches>
<dex:try-catch>
  <dex:try>
    <!-- instructions -->
    <dex:new-instance value="java.lang.Exception" vx="java.lang.Exception"/>
    <dex:throw vx-type="java.lang.Exception" class="java.lang.Exception"/>
  </dex:try>
  <dex:catch exception-type="java.lang.Exception" target="8" />
</dex:try-catch>
<dex:label id="8" />
```

With the listings in this section the XMLVM proprietary IR of all the important features of *Java Card* are described.

4.1.2 Creation of C Files

The XML representations of the single *Java* classes are converted to C source and header files using an *Extensible Stylesheet Language* (XSL) file. This file is used for defining XML file transformation and representation [40]. Table 4.1 shows how the single structures in *Java* (such as classes, objects, fields, ...) are represented in the C files created by XMLVM. In this section each *Java* structure representation in the C code is explained in more detail. Furthermore, the conversion of the exception handling and the `instanceof` instruction is described.

Java structure	C structure
Local variable (virtual register)	XMLVMElem{char, int, double, float, short, void*}
Class	TIB_struct{}
Object	ClassName_struct{TIB_struct* class, obj_field1, obj_field2, ...}
Static class fields	Global variable
Static + virtual methods	C functions

Table 4.1: C structures generated by XMLVM representing *Java* structures

Local Variables

Local variables represented in the *dexed Java* byte code use type-less virtual registers. This means that a local variable is not bound to any variable type. Therefore, direct translation from a *Java* variable to a *C* variable is not feasible. To keep the flexibility of the type-less virtual registers XMLVM uses a designated union in C containing fields for all primitive types, as well as a pointer which is used for referencing objects. Listing 4.5 depicts this *C* union, as well as the mapping of the primitive *C* types to the primitive *Java* types. As unions use the same address in the memory for all its fields each local variable does not use more memory than the biggest field of the union. In the case of the `XMLVMElem` this field is the `JAVA_LONG l` field. Therefore, each local variable takes up 64 bit of memory [15]. Furthermore, referencing an object is performed by using a `void` pointer to the according structure. For a better understanding this `void` pointer is defined as `JAVA_OBJECT`.

Listing 4.5: C structure representing virtual registers

```
typedef int JAVA_INT;
typedef long long JAVA_LONG;
typedef float JAVA_FLOAT;
typedef double JAVA_DOUBLE;
typedef void* JAVA_OBJECT;

typedef union {
    JAVA_OBJECT o;
    JAVA_INT i;
    JAVA_FLOAT f;
    JAVA_DOUBLE d;
    JAVA_LONG l;
} XMLVMElem;
```

Naming Convention

Java puts classes into different packages. This feature allows two classes with the exact same name to be present in a *Java* project as long as they are in different packages. Classes and also objects are represented in XMLVM as *C* structures. However, *C* does not allow two or more structures of the same name in one executable. Therefore, XMLVM appends the class name to the package name and uses it to distinguish between classes of the same name. This name is used not only for class representations but also for methods, fields as well as objects. Table 4.2 lists examples of this XMLVM proprietary naming convention. Furthermore, *Java* also allows methods with the same name but with different parameters (called *Method Overloading*). However, *C* needs every function to have a different name. Thus, XMLVM appends the parameters to the *C* function name.

Type	Package	Java	C
Class \ Object	test.doc	TestClass	test_doc_TestClass
Field	test.doc	TestClass.someField	test_doc_TestClass_someField
Method	test.doc	TestClass.someMethod (int i)	test_doc_TestClass_someMethod_int (int i)

Table 4.2: Naming Convention of XMLVM

Classes

Java classes and interfaces are represented with an XMLVM proprietary *C* structure called `TIB`. Listing 4.6 shows a simplified form of this structure, only depicting the most important fields held by the `TIB`. This `TIB` is based on the example `TestClass` *Java* class described in Section 4.1.1. The `classInitializationBegan` and `classInitialized` flags are used for indicating the status of the class' or interface's initialization. As the `TIB` structure only contains flags associated with its *Java* class or interface, the initialization is only performed once per structure.

The `extends` field is used for referencing the `TIB` structure of the extended class. The `baseType` field indicates whether the class is of a primitive type, a class or an interface. As arrays are also represented using a special *Java* class, the `TIB` also contains a field indicating the type of the array. The array can either hold primitive variables or objects. In the latter case the array simply stores references to the object structures. Furthermore, each `TIB` contains a `class` field pointing to the *Java* `Class` object assigned to the class.

The `numImplementedInterfaces` indicates how many interfaces a class implements. XMLVM computes this number by adding up the number of all interfaces the class directly implements as well as the number of interfaces extended by this interface. In the `implementedInterfaces` array the references of all implemented interfaces and extended interfaces are stored. E.g. a *Java* class implements interface `Int1` which extends two other interfaces `Int2` and `Int3`. The `implementedInterfaces` array of the class would contain references to the `TIB`s of all three interfaces `Int1`, `Int2` and `Int3`.

Furthermore, the `TIB` structure contains both an `vtable` as well as an `itable` array for storing function pointers. In both tables the function pointers are generalized with the `VTABLE_PTR`. Only if a function is called using the `vtable` or `itable`, the selected function pointer is casted into its correct format. The `vtable` is needed for realizing *Java*'s polymorphism. *Java* allows that a class extending another class overrides its methods if they have the same signature. XMLVM represents this by checking which methods of a class are overridden by a child class and adds them to the `vtable` of the class. The `vtable` of the extending class contains all function pointers of its base class but replaces the references of the overridden methods with its own functions. The `itable` is used in a similar way, but instead of references of overridden methods the references of methods defined by the interface and implemented by the implementing class are stored.

Listing 4.6: C structure representing a *Java* class

```
typedef void (*VTABLE_PTR)();

typedef struct __TIB_test_doc_TestClass {
    int classInitializationBegan;
    int classInitialized;
    const char* className;
    const char* packageName;
    struct __TIB_Template* extends;
    int sizeInstance;
    JAVA_OBJECT clazz;
    JAVA_OBJECT baseType;
    JAVA_OBJECT arrayType;
    int numImplementedInterfaces;
    struct __TIB_Template* (*implementedInterfaces)[1];
    VTABLE_PTR vtable[vtableSize];
    VTABLE_PTR itable[itableSize];
} __TIB_test_doc_TestClass;
```

In XMLVM each class is converted into a header and a source file. Each source file contains, defines and initializes the `TIB` structure representing its class as a global variable. This variable is already partly initialized before code execution as depicted by Listing 4.7.

Listing 4.7: Initialization of TIB global variable

```
__TIB_DEFINITION_test_doc_TestClass __TIB_test_doc_TestClass =
{
    0, // classInitializationBegan
    0, // classInitialized
    "test.doc.TestClass", // className
    "test.doc", // package
    &__TIB_test_doc_TestBaseClass, // extends
    sizeof(test_doc_TestClass), // sizeInstance
    XMLVM_TYPE_CLASS //type
};
```

Furthermore, each source file contains an `INIT` and an `INIT_IMPL` function. The `INIT` function only checks if the `TIB` is already initialized. If not, it calls the `INIT_IMPL` function which sets all the fields not already initialized in the global `TIB` variable. It also initializes all static fields of the class. Another variable contained by every source file representing an *Java* class is the global `__CLASS_` variable. This variable is a pointer storing the reference to a *Java* `Class` object which is used for realizing the reflection feature by XMLVM.

Objects and Fields

Similar to the *Java* class representation, XMLVM also provides a *C* structure for representing its objects. This structure contains fields representing the instance fields of the class. If a class extends another class the instance fields of the base class are also added to the *C* structure representing the extending class. Besides these variables the structure also contains a reference to the global `TIB` variable of its class. Therefore, it is ensured that the `TIB` is only allocated per class and not per instance. Listing 4.8 shows such a structure representing a *Java* object. As in Listing 4.6 the `TestClass` representation is depicted. The structure holds the reference to the `TIB` of the `TestClass` as well as a *C* structure holding its only instance field, the `virtualField`. Furthermore, it contains the `__INSTANCE_FIELDS_test_doc_TestBaseClass` macro which is defined as a *C* structure holding the instance fields of the `TestBaseClass`. With this representation method XMLVM guarantees that all instance fields of a class are inherited by its extending class.

Listing 4.8: C structure representing the instance of a *Java* class

```
struct test_doc_TestClass {
    __TIB_test_doc_TestClass* tib;
    struct {
        __INSTANCE_FIELDS_test_doc_TestBaseClass;
        struct {
          JAVA_BYTE virtualField_;
        } test_doc_TestClass
    } fields;
};
```

These objects structures are allocated by calling the converted constructor of the class. In its basic form the constructor performs the following operations:

- Check if class `TIB` is already initialized
  - If not, initialize it
- Allocate a new object structure
- Initialize all the instance fields of the object structure

After creating and initializing the object structure the constructor function returns a reference to the caller. This reference is either assigned a local variable or a static or instance field.

As already mentioned, static *Java* class fields are represented in XMLVM using global variables. These variables follow the naming convention of XMLVM. To enhance readability and to better distinguish instance fields and static fields the name of a global variable representing a static field is appended with a `_STATIC_` keyword. Therefore, the static field `staticField` in the `TestClass` *Java* class located in the `test.doc` package is represented by the global variable with the name `_STATIC_test_doc_TestClass_staticField`. By default all static fields of a *Java* class become static global variables, although this does not correspond to their actual accessibility. XMLVM also produces for each static global variable a `PUT` and a `GET` function for modifying and accessing them. As XMLVM does not store any information about whether a global variable is *final* or not in its IR, in the *C* code this is not represented, either.

Methods

All methods of a *Java* class are represented by *C* functions following the naming convention explained in the Section *Naming Convention*. *Java* allows to restrict the access to methods in the same way as it restricts the access to fields using *access modifiers*. These modifiers are not represented in the resulting *C* file as all methods are represented by ordinary C functions with no keywords influencing their visibility.

There is one particular difference between the *C* representation of virtual and static methods. Static methods are represented with functions that take exactly the same parameters as their corresponding *Java* methods. *C* functions representing virtual methods, on the other hand, take one additional parameter. This parameter is a reference to the object the represented virtual method is called from. Therefore, the virtual method `virtualMethod()` shown in Listing 4.1 is represented by the function depicted in Listing 4.9. Note the `JAVA_OBJECT me` which holds previously described reference.

Listing 4.9: C function representing a virtual *Java* method

```
void test_doc_TestClass_virtualMethod___byte(JAVA_OBJECT me, JAVA_BYTE n1)
{
    //XMLVM_BEGIN_WRAPPER[test_doc_TestClass_virtualMethod___byte]
    XMLVMElem _r0;
    XMLVMElem _r1;
    _r0.o = me;
    _r1.i = n1;
    test_doc_TestBaseClass_virtualMethod___byte(_r0.o, _r1.i);
    ((test_doc_TestClass*) _r0.o)->fields.test_doc_TestClass.virtualField_ =
      _r1.i;
    return;
    //XMLVM_END_WRAPPER
}
```

In general all *C* functions are called directly from the caller. However, if a method is part of polymorphism, the representing function is called via the function pointer stored in `vtable` of the `TIB` structure representing the class the method belongs to. Listing 4.10 visualizes such a function call.

Listing 4.10: C function call of overriden method

```
r0.o = __NEW_test_doc_TestClass();
(*(void (*)(JAVA_OBJECT)) ((test_doc_TestClass*)_r0.o)->tib->vtable[6])(_r0.o);
```

Native Function Calls

Java allows interaction with *C* source code by means of a *Java Native Interface* (JNI) library. With the JNI it is possible to call *C* functions from inside *Java* methods. This feature must also be supported by the XMLVM. Therefore, all natively defined methods (indicated by the `native` keyword in *Java*) get declared in the corresponding header file of the *Java* class representation. XMLVM searches for *C* source files starting with `native_` and ending with a package plus class name corresponding to a converted *Java* class. If such a file is found, it is added to the converted *C* files. The calling convention of these native functions does not differ from other converted methods.

Exception Handling

The exception handling in the *C* source code created by XMLVM relies on `setjmp.h` standard library. This library is typically used for implementing an exception mechanism. It consists of a special structure (`jmp_buf`) and two functions (`setjmp` and `longjmp`). The `jmp_buf` is a structure dedicated for storing the program state at a certain point of the program execution. The `setjmp` function saves the program's calling environment by setting up the `jmp_buf`. This function returns 0, if it was called directly and a nonzero value if it was jumped to by the `longjmp` function. The `longjmp` function restores the program environment by using the `jmp_buf` which was set up by the `setjmp` function before. Furthermore, it lets the execution continue from the corresponding `setjmp` function.

Using these standard *C* functions the exception handling is performed. XMLVM uses macros for encapsulating the logic behind its exception mechanism. With the help of Listing 4.11 these macros are explained in more detail.

As XMLVM supports also multi-threading the `java.lang.Thread` class is used for storing the program state and the exception which is potentially thrown. `XMLVM_TRY_BEGIN` declares a local `jmp_buf` and copies the execution environment stored in the current thread to it, thus creating a backup. Furthermore, it initializes the `jmp_buf` of the current thread which is used during the `try - catch` block by calling the `setjmp` function and checks the function's return value. If the `setjmp` returns 0 the code encapsulated by the `XMLVM_TRY_BEGIN` and the `XMLVM_TRY_END` is executed. If it returns with a value that is nonzero, the execution would continue at the `XMLVM_CATCH_BEGIN` macro.The `XMLVM_THROW_CUSTOM` macro stores the provided exception using the current thread and calls the `longjmp` function using its `jmp_buf`. The execution jumps back to the `setjmp` function inside the `XMLVM_TRY_BEGIN` which now returns with a nonzero value. As explained before, the `XMLVM_CATCH_BEGIN` macro is now called, which basically marks the beginning of the `XMLVM_CATCH_SPECIFIC` macro. The `XMLVM_CATCH_SPECIFIC` macro checks if the exception stored by the `XMLVM_THROW_CUSTOM` macro is of the correct class. If it is of the correct class, it performs a jump using a `goto` instruction to the `label` with the indicated number. In the case of `XMLVM_THROW_CUSTOM` this would be `label8`. The last macro, the `XMLVM_RESTORE_EXCEPTION_ENV` restores the execution environment from before the `try-catch` block by replacing the `jmp_buf` of the current thread with the one backed up using the local by the `jmp_buf` variable in the `XMLVM_TRY_BEGIN` macro called at the beginning of the exception handling. The backup of the program state using a local variable is necessary as it cannot be known whether the execution already resides within another `try-catch` block. Hence, every translation of a `try-catch` block puts a new `jmp_buf` variable on the stack. Depending on the system the `jmp_buf` allocates from 24 up to 64 bytes of memory [20].

Listing 4.11: Exception Handling in *C* source code

```
XMLVM_TRY_BEGIN(uniqueThreadID)
// Begin try
test_doc_TestClass_PUT_staticField( _r1.i);
_r0.o = __NEW_java_lang_Exception();
java_lang_Exception___INIT___(_r0.o);
XMLVM_THROW_CUSTOM(_r0.o)
// End try
XMLVM_TRY_END
XMLVM_CATCH_BEGIN(uniqueThreadID)
    XMLVM_CATCH_SPECIFIC(uniqueThreadID,java_lang_Exception,8)
XMLVM_CATCH_END(uniqueThreadID)
XMLVM_RESTORE_EXCEPTION_ENV(uniqueThreadID)
label8:;
java_lang_Thread* curThread_uniqueThreadID =
(java_lang_Thread*)java_lang_Thread_currentThread__();
_r0.o = curThread_uniqueThreadID->fields.java_lang_Thread.xmlvmException_;
_r0.i = 1;
test_doc_TestClass_PUT_staticField( _r0.i);
```

XMLVM_EXIT_METHOD()

instanceof

Another important feature of both *Java* and *Java Card* is the `instanceof` instruction. It is represented in the generated *C* by a call to the `XMLVM_ISA` function. This function is provided with the object that is to be checked and the class it is to be checked against. The first parameter is simply a reference to the object structure. The second parameter is a reference to a `java.lang.Class` object. Each *C* source file representing a *Java* class contains a global `__CLASS_classname` variable which points to a `java.lang.Class` object. This `java.lang.Class` contains a virtual `tib` field which points to the `TIB` of the class it belongs to. This global variable representing the class to be checked against is provided as second parameter to the `XMLVM_ISA` function. The function checks the `TIB` of the object with the `TIB` referenced by the `__CLASS_classname`. If it matches, `true` is returned. If it does not match, it loops through the `TIB`s of the implemented interfaces. If no match is found, the process examines the `TIB` of the object's extended class and its implemented interfaces. This procedure is repeated until the end of the class hierarchy is reached. If no match was found, the function returns `false`.

Array Representation

Java arrays are represented by using a special class. This `XMLVMArray` class is converted along with the user written classes and the *Java* framework classes. Listing 4.12 shows a simplified representation of this `XMLVMArray` class depicting its fields and methods.

Listing 4.12: XMLVMArray class for array implementation

```java
final public class XMLVMArray {
    private Class type;
    private int length;
    private Object array;
    private XMLVMArray(Class type, int length, Object array) {
        this.type = type;
        this.length = length;
        this.array = array;
    }
    native static private XMLVMArray createSingleDimension(Class type,
      int size);
    native static private XMLVMArray createSingleDimensionWithData(Class type,
      int size, Object data);
}
```

The `createSingleDimensionWithData` creates a new `XMLVMArray` object, sets the length and type accordingly and assigns the `data` reference provided by the caller to the `array` field of the `XMLVMArray` object. The `createSingleDimension` function allocates $type * size$ bytes of memory and zero initializes this area. Then it calls the `createSingleDimensionWithData` function with the allocated memory area as `data` reference.

Especially the `type` field of the `XMLVMArray` is interesting as it utilizes the *Java* `Class` class for defining the type of the elements stored in the array.

4.1.3 Memory management of XMLVM

The *C* code generated by the XMLVM framework puts all variables and structures into the RAM. The object structures are allocated in `heap` section of the RAM. The global variables representing static fields are put into the `data` section of the RAM. In this section also the `TIB` structures are placed which are also declared as global variables. The local variables are typically placed in the stack section of the RAM [22]. Figure 4.1 visualizes this memory layout.

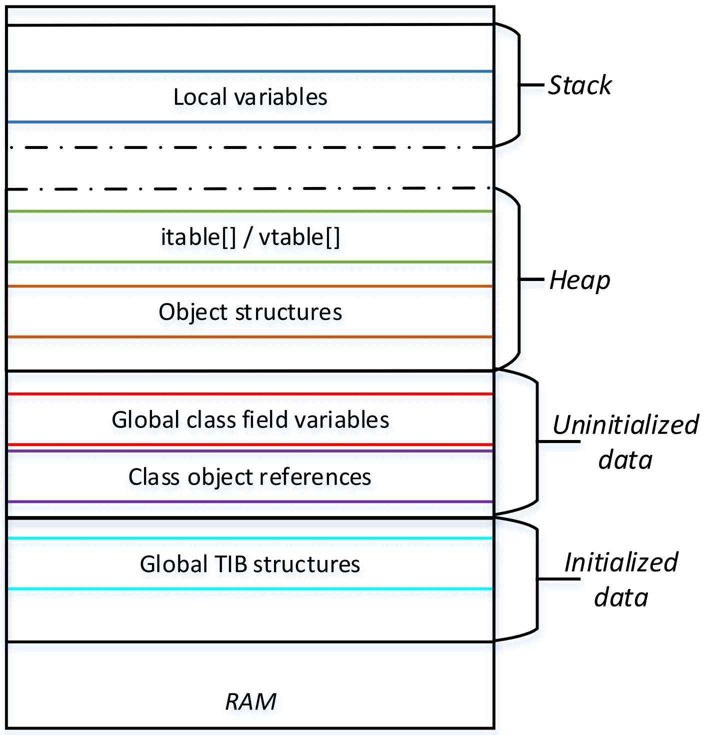


Figure 4.1: Memory layout of XMLVM produced *C* code

Furthermore, the XMLVM make uses the GC implemented by Hans-J. Boehm and Alan J. Demers.

4.1.4 Testing of XMLVM conversion process

Before the XMVLM framework is adapted to produce *C* code loadable onto the NXP platform, the capabilities of the framework are tested by implementing the test project introduced in Section 3.2.1. This test project uses all the important features of *Java Card* such as inheritance, method overloading, native method invocation, `instanceof` and exception handling. These test cases are depicted in Figure 3.1 Only after verifying that all these features are correctly represented in the converted *C* the adaptations can be performed. Basically, this test project contains a `Tester` class which implements the `static void main(String[] args)` function identified by standard *Java* projects as its entry point. This `Tester` class performs all the test cases listed above.

4.2 Java Card Cross-Compilation Framework Modification

After explaining the basic conversion process performed by the XMLVM framework in Section 4.1 and verifying the correct *Java* byte code conversion performed by the XMLVM framework, the modifications characterized in the single steps in Chapter 3 are explained in more detail in this section.

4.2.1 Reducing the Memory Requirements

In the first step of the modification process the executable produced by the XMLVM framework is examined. Based on this investigation, the set of converted classes located in the various libraries is determined and a way to minimize this set is found. As the results of this first downsizing are not satisfying, the used *Java* framework classes are modified and features not supported by *Java Card* are removed.

Determining and Minimizing Set of converted Classes

Using the XMLVM framework for converting a very simple `HelloWorld` *Java* program produces an executable taking up 12.8MB of memory. Such an executable would be way too large to be loaded onto an NXP platform as defined in Section 3.1. Therefore, a way has to be found to minimize this executable. Investigating into the produced *C* source files shows that over all 853 *Java* classes are converted. This includes, next to the *Java* classes of the test project, all the classes from the *Java* framework including the additional libraries coming with the XMLVM framework.

XMLVM offers the opportunity to define the set of converted *Java* classes by using either a so called `redlist` or a `greenlist`. The `redlist` contains classes which must not be converted. Therefore, XMLVM converts all classes from the various libraries, the *Java* framework as well as the user written classes omitting those defined in the `redlist`. Using the `greenlist` XMLVM does not convert any classes but those listed in the `greenlist`. Classes which are not converted because of one of the two lists excluding them are called *redtyped*.

Furthermore, XMLVM replaces all references to methods or fields of classes which are excluded from the conversion by calls to a special function printing the name of the expected methodfield of the *redtyped* class plus the classes name to the display. This feature simplifies the debugging.

The first version of the converted *Java* test project is created by using the unmodified `redlist` contained in the XMLVM framework. To minimize the set of the converted classes the `greenlist` is used. The initial `greenlist` only contains the classes of the test project, the primitive types (`double, int, long, float, byte, char, short, boolean` and `void`) as well as the `java.lang.Object` (because every class in *Java* extends the `Object` class) and the `org.xmlvm.runtime.XMLVMArray` class which is used for representing arrays. Trying to convert the test project with this `greenlist` fails.

The goal now is to find the minimal set of classes needed in the conversion process to create a compileable and executable program. Therefore, a script is implemented for automatically extending the set of classes being converted by XMLVM until such an executable is created. There are three reason why the creation or execution of the program might fail. The first reason is that the conversion process stops with an error as a class marked for conversion extends a *redtyped* class. The second reason is that the converted program does not build because a converted class includes a *redtyped* class. The last reason is that the program tries to access a fieldmethod of a *redtyped* class during execution. The script performs the following task:

- Step 1: Run the conversion process of XMLVM. Check its output
  - If it failes because of a *redtyped* class, add class to `greenlist`. Go to Step 1
  - If it passes continue with Step 2
- Step2: Compile the created *C* source code
  - If a compilation error occurs because of a *redtyped* class, add class to `greenlist`. Go to Step 1
  - If the code is built, continue with Step 3
- Step 3: Execute the converted program
  - If it tries to access a fieldmethod of a *redtyped* class, add class to `greenlist`. Go to Step 1
  - Otherwise, it executes as expected

Using this script the set of classes being converted is reduced from the initial 853 classes to 198 classes. This leads to an executable with a size of 1.8MB which would be loadable to an NXP platform but is still very large.

After minimizing the size of the executable, the RAM memory consumed by the executable has to be determined. As not only the FLASH memory but also the RAM of the NXP platform are very limited, this examination is crucial. Therefore, the GC used by XMLVM is disabled and a counter is introduced which is increased by the amount of memory allocated by the `malloc` function. Thus, the amount of heap memory used for the object creation can be tracked. This method shows that only for the object allocation 241KB of RAM are needed. This exceeds the capabilities of the NXP platform by far. Further investigation show that, before the first line of the test project is executed, the program already allocates 227KB of RAM. This leads to the conclusion that the used *Java* library has to be adapted.

Adaptation of *Java* Runtime Classes and Removal of Features

As minimizing the set of converted classes does not lead to the needed result, the features not supported by *Java Card* have to be removed. Furthermore, also the *Java* runtime library has to be adapted.

As a first step the `greenlist` is adapted once more. Only the classes of the test project and the primitive types are allowed to be converted. Additionally, also the `Object`, the

`Exception`, the `Throwable` and the `String` class are added to the `greenlist`. With this configuration the test project is not convertible as the `Object`, the `String`, the `Exception` and the `Throwable` class contain *redtyped* dependencies. Instead of adding these dependencies as done by the script described in the Section before, the failure causing classes are adapted.

The `Object` class as provided by the XMLVM framework contains dependencies to the `Class`, the `ArrayList` and XMLVM proprietary *Java* classes needed for multi-threading. The dependencies are removed by modifying the `Object` class.

The `String` class is modified in such a way that it only contains the basic fields, a default constructor and a constructor provided with a `char` array and the `arrayCopy` method. All other dependencies are eradicated. This class is used for printing out debug information.

The `Exception` class is left untouched. However, the `Throwable` class is rid of implementing the `Serializable` interface and its dependencies to the `IOException`, the `ObjectOutputStream`, the `PrintStream` as well as the `PrintWriter` class. With these small adaptations to the used *Java* framework, the test project is convertible, but not buildable, as the *Java* features multi-threading and reflection depend on classes which are now *redtyped*. To make the project work, the support of reflection is disabled by setting the `genReflectionData` to false in the XMVLM project. Furthermore, also the whole section creating the *C* code representing the reflection in the single source files is removed from the `XSL` file. The multi-threading which utilizes the *Java* `Thread` class is completely eradicated from the `XSL` file as well as the supporting *C* source files provided by XMLVM. This includes the usage of the functionality provided by the `pthread.h` library on which the multi-threading of XMLVM builds. Another issue that needs to be solved is the lack of the *Java* `Class` class. As described in Section 4.1.2, both the representation of the `instanceof` instruction as well as the `XMLVMArray` class for the array representation utilize the `Class` class. The `XMLVM_ISA` function which represents the `instanceof` instruction uses the `tib` field of the `Class` class for checking the type of the provided object. The `XMLVMArray` class utilizes the `Class` class for defining the type of the array. Including the *Java* `Class` class provided by XMLVM to the `greenlist` is no option as it adds dependencies related to the reflection feature. Therefore, a new *Java* class was implemented replacing the `Class` class. This new `SimpleClass` class is depicted in Listing 4.13. It only contains one field for referencing a `TIB`, a constructor setting this field as well as a native method declaration for initializing the native layer of the class. The native part of the `SimpleClass` class contains `TIB` global variable structures as well as `__CLASS_` variables for the primitive types. Furthermore, the `initNativeLayer()` function initializes all the global variables used for representing the primitive types. With this new `SimpleClass` replacing the original `Class` class, the `XMLVM_ISA` function and the `XMLVMArray` class can be built and their implementation needs no further modifications.

Listing 4.13: `SimpleClass` class replacing java.lang.Class

```
public class SimpleClass
{
  public Object tib;
  native private static void initNativeLayer();
  private SimpleClass(Object tib) {
    this.tib = tib;
  }
}
```

The last issue caused by rewriting the *Java* framework classes refers to the exception handling. The macros used by the XMLVM proprietary representation of the exception handling, as described in Section 4.1.2, utilize the *Java* `Thread` class provided by XMLVM. More precisely, the `Thread` class holds two fields, the `Object xmlvmExceptionEnv` and the `Object xmlvmException` field, which are needed for the exception handling. Due to the removal of multithreading there is only need for one global structure holding these fields. Therefore, a small *C* structure was added. It is depicted in Listing 4.14. It only contains the `JAVA_OBJECT environment` and the `JAVA_OBJECT exception` field.

Listing 4.14: `ExceptionContainer` replacing `java.lang.Thread`

```
typedef struct {
    JAVA_OBJECT environment;
    JAVA_OBJECT exception;
}ExceptionContainer;

ExceptionContainer* globExceptionCont;
```

Furthermore, the calls to the `Thread` class in the *C* macros are replaced with calls to the `globExceptionCont` utilizing its fields.

As the creation of a new local `jmp_buf` variable on each call of the `XMLVM_TRY_BEGIN` macro means a huge stack memory consumption, also this design flaw must be tackled. After some investigation it can be seen that this local `jmp_buf` variable is only used from its allocation until the corresponding `XMLVM_RESTORE_EXCEPTION_ENV` macro is called. This means that the visibility of this variable can be limited. In *C* such a scope limitation is performed by using "{ }" brackets surrounding the source code in which the variable should be visible. This limitation allows the compiler to reuse the stack memory used for the local variable, as soon as the variable is visible no longer. Thus, the `XMLVM_TRY_BEGIN` is added a "{" and the `XMLVM_RESTORE_EXCEPTION_ENV` macro is added a "}" bracket. This adaptation ensures that large *Java* methods using multiple nested `try-catch` blocks will not exceed the stack memory of the targeted hardware platforms by constantly allocating new local `jmp_buf` variables. With these adaptations the functionality of exception handling is ensured.

Performing these adaptations to the code generating as well as the *Java* framework classes, enable a minimization of the converted executable. The amount of translated *Java* classes is cut down from 198 to 24 classes. This results in the size of the executable shrinking from 1.8MB to 224.2KB. Further investigation into the memory consumption shows that the converted program without supporting reflection or multi-threading and with the reduced set of *Java* classes only occupies 3154 bytes of RAM for allocating objects.

Testcase Exception Handling

Additionally to the conversion of the exception handling mechanism of *Java* XMLVM also supports automatic checks on null pointers and array bounds. Every instruction which accesses an object is surrounded by a `try-catch` block. Within this block, it is checked whether the object is null or not. If it is null a `NullPointerException` is thrown. Similar to the null pointer check also the an array access is surrounded by a `try-catch` block. Inside this block it is made sure that the index of the accessed element is inside the array's bounds. If not an `ArrayIndexOutOfBoundsException` is thrown. The testcase verifying the exception handling depicted in Figure 3.1 is extended deliberately trying to access a null object and an array element using an index which exceeds the bounds of the array.

Conclusion

After performing the described adaptations to the conversion process as well as the *Java* framework classes the test project can be converted and compiled to an executable with a memory usage not exceeding the limits of the targeted platforms. Therefore, the next step is to board the resulting executable to a ARM based development board. For this step the STM32F407VG discovery board is used.

4.2.2 Boarding to STM32 Discovery Board

This section describes the modifications on the framework considering further downsizing of *Java* class representation as well as changing the class and object allocation to represent the object life cycle used by *Java Card* rather than *Java*.

Minimization of *Java* Class Representation and Local Variables

The `TIB` structure used for the *Java* class representation still contains fields which are not used or unnecessary. Listing 4.15 partially depicts the changes performed on the `TIB` structure. Fields which are removed are crossed out with a red line. Fields that are added are highlighted in green colour.

The `classInitializationBegan` and `classInitialized` field are replaced with one `classInitStatus` field holding the status of the `TIB`. This status indicates whether the `TIB` is already initialized, if its initialization is ongoing or if it is not yet initialized. The `className` and the `packageName` fields are removed as they are only used for debugging reasons but can take up a lot of memory space. Furthermore, also the `clazz` was removed as its usage can be substituted by utilizing the global `__CLASS_` variables of the single class representations.

As *Java Card* does not support a the `long, double` and `float` types of *Java*, also the local variable representation can be minimized. As already mentioned the local variables are represented by the `XMLVMElem` union which reserves the amount of memory necessary for storing its biggest member. In the case of the `XMLVMElem` union this is the `JAVA_LONG l` needing 8 byte of memory. However, the *Java Card* supported primitive types are all represented by solely using the `JAVA_INT i` field. Therefore, the `XMLVMElem` union is adapted as shown in Listing 4.16. Thus, each represented local variable in the *C* source code only takes up 4 byte of memory [15], [24].

Listing 4.15: C structure representing a *Java* class

```
typedef struct __TIB_test_doc_TestClass {
    ~~int classInitializationBegan;~~
    ~~int classInitialized;~~
    int classInitStatus;
    ~~const char* className;~~
    ~~const char* packageName;~~
    struct __TIB_Template* extends;
    int sizeInstance;
    ~~JAVA_OBJECT clazz;~~
    JAVA_OBJECT baseType;
    JAVA_OBJECT arrayType;
    int numImplementedInterfaces;
    struct __TIB_Template* (*implementedInterfaces)[1];
    VTABLE_PTR vtable[vtableSize];
    VTABLE_PTR itable[itableSize];
} __TIB_test_doc_TestClass;
```

Listing 4.16: Adapter `XMLVMElem` union

```
typedef union {
    JAVA_OBJECT o;
    JAVA_INT i;
    ~~JAVA_FLOAT f;~~
    ~~JAVA_DOUBLE d;~~
    ~~JAVA_LONG l;~~
} XMLVMElem;
```

HAL function modification

Boarding the converted program from the *x86* platform onto the STM32 discovery board shows the necessity of replacing the standard *C* functions with self-written implementations. The functions that are removed are the `printf`, the `malloc`, the `realloc`, the `free`, the `memcpy` and the `memset` function. All these functions are replaced but the `realloc` and the `free` function. These functions were omitted as no GC is supported. Once allocated memory is not freed by the system.

Instead of utilizing the heap memory section of the RAM as performed by the `malloc` function, a global array is utilized reserving memory space for allocating up to 1KB of data. This global array is called `gFlashMemArray` in reference to its future location in FLASH memory. Furthermore, a global pointer (`pFreeFlashMem`) is defined pointing to the end of the last allocated memory block within the array. In this step of the implementation this array is still placed in RAM. The `malloc` function is replaced by the `mem_manager_MemAlloc` function. This function receives as parameter the amount of bytes to reserve. It checks if the amount of bytes is still free for allocation in the `gFlashMemArray`. If the free memory space in the array is less than needed, a `null` pointer is returned. If enough space is available, it increases the `pFreeFlashMem` pointer by the amount of allocated bytes and returns the starting address of the allocated block (which is the old address `pFreeFlashMem` was pointing to). The `memcpy` function is replaced by the `mem_manager_MemCpy` function. This function expects a destination and a source pointer and the amount of bytes to be copied.

It checks, if either the destination, or the source address is word aligned (a word equals 4 byte). If not, it copies the single bytes from the source to the destination address, until an alignment is reached. Then, it copies the data word-wise (4 bytes at a time) until the size of the remaining bytes is smaller than the word size. These bytes are again copied one by one. The `memset` function is replaced by the `mem_manager_MemSet` function which receives a pointer to a destination address, a value to be copied and the amount of bytes to be set to this value. It always casts the value to a `char` and sets the amount of bytes provided to the function to this value starting from the destination address.

For debugging reasons also the standard `printf` is substituted with a modified implementation of the `printf` function developed by *Michael Ringgaard*. This function sends the string provided to the function character by character over the UART. The debugging information sent via the UART is shown using the *MiniCom* tool. The UART functionality is provided by the libraries shipped with the STM32 discovery board. The debug information is still provided in form of `String` classes.

With these changes to the usage of the underlying libraries, the changes of the object life cycle were performable.

Object Life Cycle Adaptation

Java puts objects, class representations, local variables as well as virtual and static fields into the RAM memory. This means that after a loss of power to the underlying hardware system all the allocated data is lost. However, *Java Card* comes with the requirement that objects, virtual and static fields are preserved on a card reset or power loss. Therefore, these parts of a *Java Card* program need to be put into persistent memory. For both the STM32 discovery board as well as the NXP platform this memory type is FLASH. As the XMLVM framework targets *Java* applications, the life cycle of *Java* is preserved by the generated *C* executable. This object life cycle now needs to be adapted to the demands of *Java Card*. Basically, this means that everything allocated during the program's execution but local variables must be shifted from RAM to FLASH memory. As already mentioned, the `gFlashMemArray` is used in this step to simulate the FLASH memory section in which the objects and class representations as well as the static variable representations are stored. Another goal of this step is to allocated the global variables as well as the *C* structures in the `gFlashMemArray` in such a way that the memory consumption can be determined more precisely. The new memory functions are further abstracted with *C* macros which call the underlying memory functions based on a *C* define. These macros guarantee that from a caller perspective the function call never changes, even if the underlying memory function is replaced in future. The `mem_manager_MemCpy` is abstracted by the `MEM_COPY` and the `mem_manager_MemAlloc` function is abstracted by the `FLASH_ALLOC` macro. The adaptations concern four different aspects of the conversion.

Static and Virtual Field Manipulation: The first modification concerns the assignment of global variables and virtual fields of the object structures. Data stored in FLASH memory cannot be assigned in the same way as data allocated in RAM. A special *FLASH memory copy* function provided by the firmware layer must be utilized. This FLASH function takes the same parameters as the `mem_manager_MemCpy` function. Therefore, the `mem_manager_MemCpy` function is used for simulating the data manipulation in FLASH memory. However, in this step the `gFlashMemArray` is used instead of a real FLASH memory. Listing 4.17 shows how the manipulation of a object field and of a global variable must be changed respectively. The assignment of instance fields is performed by XMLVM by accessing the field via the object structure and using the `=` symbol for manipulating its value. This is changed by passing the address of the instance field as destination, the address value as source and the size of the value as parameters to the memory copy function. Static fields are exclusively manipulated by calling their generated `PUT` functions. Therefore, the generating of `PUT` functions' implementation must be changed in a similar way. However, the static field is passed directly as destination instead of its address. This difference is caused as the global variable representing the static field is now declared as a pointer. This modification is explained in the next paragraph.

Listing 4.17: Adaptation of static and virtual field manipulation

```
//--------------------- Object field manipulation ---------------------//
//old instance field assignment
((test_doc_TestClass*) _r0.o)->fields.test_doc_TestClass.virtualField_ = _r1.i;

//new instance field assignment
MEM_COPY(&(((test_doc_TestClass*)_r0.o)->fields.test_doc_TestClass.
  virtualField_), &_r1.i, sizeof(_r1.i));

//--------------------- Static field manipulation --------------------//
//old static class field assignment
void test_doc_TestClass_PUT_staticField(JAVA_CHAR v)
{
  _STATIC_test_doc_TestClass_staticField = v;
}

//new static class field assignment
void test_doc_TestClass_PUT_staticField(JAVA_CHAR v)
{
  MEM_COPY(_STATIC_test_doc_TestClass_staticFinalField, &v, sizeof(v));
}
```

Assigning a global variable representing a static field or a instance field to a local variable is still performed by using the `=` symbol. The adaptations are performed by modifying the way the `XSL` file generates the *C* source code.

Global Variable Allocation: The second change adapts the way global variables representing the static class variables of *Java* are allocated. In XMLVM global variables are declared in the `data` section of the RAM. To get a more precise result on how much FLASH memory the converted project consumes, the global variables are allocated in the `gFlashMemArray` in this step. The global variables are changed into pointers holding the address of the actual variables allocated in the `gFlashMemArray`. Listing 4.18 visualizes this change.

Listing 4.18: Adaptation of static field allocation

```
//--------------------- Static field allocation ---------------------//
//old static field allocation
static JAVA_BYTE _STATIC_test_doc_TestClass_staticFinalField = 0;

//new static field allocation
static JAVA_BYTE* _STATIC_test_doc_TestClass_staticFinalField = JAVA_NULL;

void __INIT_IMPL_TestClass()
{
...
//allocate static variable and assign default value to it
long long v = 0;
_STATIC_test_doc_TestClass_staticFinalField = FLASH_ALLOC(sizeof(JAVA_BYTE));
MEM_COPY(_STATIC_test_doc_TestClass_staticFinalField, &v, sizeof(JAVA_BYTE));
...
}
```

Also this modification is performed by adapting the `XSL` file.

TIB Structure Allocation: The third modification which needs to be performed is how the `TIB` structures representing the *Java* classes are allocated. XMLVM declares a global `TIB` variable for each converted class. Furthermore, this global variable is partly initialized already on declaration. To simulate the allocation of the `TIB` structure in the FLASH memory, the global variable is replaced with a `TIB` pointer. The initialization is completely performed in the `INIT_IMPL` function of the class. The partial initialization of the global `TIB` structure is depicted in Listing 4.7. In Listing 4.19 the changes performed to the structure's allocation and initialization is shown. Fields of the `TIB` which were also initialized in the `INIT_IMPL` function by the original XMLVM translation process are still initialized by this function using, however, the `MEM_COPY` function. The memory for the `TIB` structure is allocated using the `FLASH_ALLOC` function in the `INIT` function.

Listing 4.19: Adaptation of `TIB` allocation and initialization

```
//-------------------- Old allocation and initialization --------------------//
__TIB_DEFINITION_test_doc_TestClass __TIB_TestClass = {
  //Init
}

void __INIT_IMPL_test_doc_TestClass() {
  ...
  // Initialize vtable for this class
  __TIB_test_doc_TestClass.vtable[6] =
    (VTABLE_PTR) &test_doc_TestClass_virtualMethod___byte;
  ...
}

//-------------------- New allocation and initialization --------------------//
__TIB_DEFINITION_test_doc_TestClass* __TIB_TestClass = JAVA_NULL;
```

```
void __INIT_test_doc_TestClass() {
  if (__TIB_test_doc_TestClass == JAVA_NULL){
    __TIB_test_doc_TestClass = FLASH_ALLOC(sizeof(__TIB__test_doc_TestClass));
    __INIT_IMPL_test_doc_TestClass();
  }
}

void __INIT_IMPL_test_doc_TestClass() {
  ...
  int zero = 0;
  // classInitialized
  MEM_COPY(&__TIB_test_doc_TestClass->classInitialized, &zero, sizeof(int));
  // extends
  MEM_COPY(&__TIB_test_doc_TestClass->extends, &__TIB_test_doc_TestBaseClass,
    sizeof(__TIB_DEFINITION_TEMPLATE*));
  // Initialize vtable for this class
  VTABLE_PTR pfunc = test_doc_TestClass_virtualMethod___byte;
  MEM_COPY(&__TIB_test_doc_TestClass->vtable[6], &pfunc, sizeof(VTABLE_PTR));
  ...
}
```

Even the changes for the allocation and initialization of the `TIB` are realized by modifying the `XSL` file. Through the adaptations performed to the generating process, the `TIB` structures representing the *Java* classes are now allocated in the memory array simulating the FLASH memory section of the targeted platform.

Object Allocation: The last adaptation, needed for allocating all but the local variables in the simulated FLASH memory section, targets the object creation. XMLVM allocates the structures representing the *Java* objects in the heap section of the RAM by using the standard *C* `malloc` function. This allocation is performed by calling the appropriate `__NEW_` function which represents the targeted *Java* constructor. Instead of performing the allocation using the `malloc` function the `FLASH_ALLOC` macro is utilized. In addition, the assignment of the `tib` field of the object structure is changed from assigning with `=` to using `MEM_COPY` macro. Listing 4.20 visualizes this change in more detail.

Listing 4.20: Adaptation of object allocation and initialization

```
JAVA_OBJECT __NEW_test_doc_TestClass() {
  ...
  //old object allocation
  test_doc_TestClass* me = XMLVM_MALLOC(sizeof(test_doc_TestClass));
  me->tib = &__TIB_test_doc_TestClass;

  //new object allocation
  test_doc_TestClass* me = FLASH_ALLOC(sizeof(test_doc_TestClass));
  MEM_COPY(&me->tib, &__TIB_test_doc_TestClass, sizeof(void*));

  ...
  return me;
}
```

Furthermore, the instance fields assignment to default values must be changed as well. For this modification the assignment which is performed by XMLVM using the `=` operator is also replaced with a call to the `MEM_COPY` macro. The `SimpleClass` object structure allocated for each converted *Java* class has already been changed to the simulated FLASH memory through the modifications performed in this step. Again, the `XSL` file is used for performing the adaptations to the object allocations and initialization.

Testcase FLASH - RAM

A test case is created for checking the adaptations performed in this step. It consists of a *Java* class with static and virtual fields and methods manipulating the values of them. Furthermore, it is verified that the *C* representations of the static and virtual fields as well as the `TIB` structures are allocated inside the `gFlashMemArray` by checking the addresses the individual pointers are pointing at.

Conclusion

By performing the adaptations explained in this step the object life cycle is shifted to the FLASH memory simulated by the `gFlashMemArray`. Figure 4.2 depicts the new memory layout in comparison with the old one in more detail. The modifications are verified by implementing an additional test case. Furthermore, comparing the output of the test project converted with the modified framework to the output of the test project translated with the original XMLVM framework shows that both version of the test project produce identical debug information. Checking the `pFreeFlashMem` after executing the whole test project shows that 7216 bytes of memory are allocated. The biggest part of this memory area is needed for storing the single `String` objects holding the debug information being sent via UART. With this step, the boarding to the NXP platform is prepared which is described in Section 4.2.3

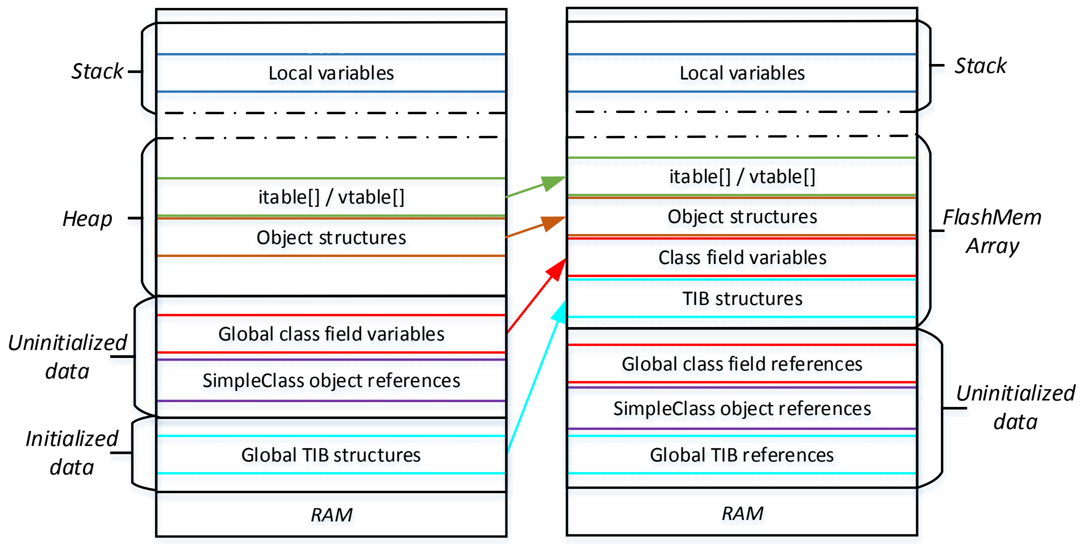


Figure 4.2: Memory layout of *Java Card* object life cycle simulation

4.2.3 Boarding to the NXP Chip

This section describes what modifications are performed to execute the generated test project on the NXP platform. It explains which parts of the JCOP system are used for providing the necessary functionality to the generated *C* source code. Furthermore, the adaptations performed on both the JCOP system and the framework are outlined. These are needed for realizing the persistent object allocation utilized for realizing the *Java Card* life cycle. The modified XMLVM framework in this and the further steps is referred to as the JCF.

JCOP Initialization and Test Project Execution

As described in section 1.3.4, the JCOP system consists of several layers. As the byte code usually interpreted by the JCVM is directly converted by the JCF into compilable *C* source code, a big part of the OS layer of JCOP is not needed. However, the underlying HAL is crucial to be properly initialized. It is needed for providing the necessary functionality to use the UART and manipulate the FLASH memory. Therefore, several parts which are also performed during the start up of the JCOP OS were taken over and called from a `main` function. Similar to a normal *C* function, an executable loaded to the NXP platform needs such a `main` function as its entry point, as well.

This start up routine initializes and checks hardware registers as well as the memory. Hardware registers must first be checked on consistency. This verification has been performed as certain registers might be manipulated during a hardware-based attack on the chip. After this check is performed, the memory is initialized. In this phase the addresses of the RAM and FLASH memory and other memory sections are set. Furthermore, certain memory regions are cleared in such a way that data from previous executions do not influence the process. Furthermore, the *Memory Management Unit* (MMU) as well as the communication manager are enabled and set up. This initialization routine must be performed before being able to call the converted test project. Additionally to the above described initialization routine the UART of the NXP platform is set up as well. Similar to the version of the converted test project loaded to the STM32 discovery board described in Section 4.2.2 in this step the UART is utilized for outputting the debugging information, as well.

After performing the initialization of the JCOP HAL as well as the UART the converted test project is loaded and executed on the NXP platform. With these few changes to the framework, the execution of the project is possible as the implemented HAL functions described in Section 4.2.2 only utilize the structures defined by the framework. Therefore, no external dependencies are needed. Only the `printf` function introduced in Section 4.2.2 now calls the UART functionality of the NXP platform instead of the one provided by the STM32 discovery board.

Realization of FLASH Simulation

In the next step of the modification process the allocation of the `TIB` structures, the object structures and the global class field variables must be shifted from the `gFlashMemArray` to real FLASH memory. But not only the structures themselves must be put into persistent memory. After a card reset or a power loss the pointers storing the addresses of the individual structures must be retrievable as well. Otherwise, the data of the represented class or object would be preserved in the FLASH memory, but the program would not be able to retrieve them as their addresses would be lost. Therefore, the FLASH memory section is split into several regions. Furthermore, the *Java* class implementing the `main` function must be retrievable. Therefore, its reference must also be preserved in some way. The next paragraphs explain the changes to the memory layout and to the object referencing which enable the realization of the *Java Card* object life cycle.

Memory Sections - Linker Script: JCOP defines a memory region placed in the FLASH memory for storing objects as well as the *Java Card* byte code being interpreted by the JCVM. This region is called `pheap`. As the object representation generated by the JCF needs to be stored persistently (as it is stored by JCOP) and no *Java Card* byte code is loaded to the NXP platform, the `pheap` memory section is perfectly suited for the targeted persistent class and object allocation. With 891KB of space this section is also big enough. Additionally to the `TIB`, object structures and the global class field variables which need to be stored in FLASH, also the pointer variables referencing the individual `TIB` structures and the `Class` objects must be allocated persistently. Therefore, the `pheap` is split into several sections.

The first section of the `pheap` is used for storing the *mother object*. This section is called the `persMemMORef` and is only four bytes big. This is exactly the amount of bytes needed to store a pointer variable. After this first section the `persMemTIBRefs` section is defined. This section is used for storing the pointer referencing the allocated `TIB` structures. It offers 1024 bytes of FLASH memory which allows 256 classes to be referenced. After this `persMemTIBRefs` section comes the `persMemClassRefs` section. Similar to the section before, it is used for preserving addresses of structure. Instead of holding the references to the `TIB` structures, it is used for storing the `__CLASS_` pointers referencing the individual `SimpleClass` objects allocated for each converted *Java* class. Like the `persMemTIBRefs` section it also allows 256 pointers to be stored. The section following the `persMemClassRefs` is used for storing various global variables, used by the underlying framework which must also be stored persistently. This section is called `persMemGlobVars` and takes up 1024 bytes of FLASH. The next section is used for storing the global class variables. They are no longer allocated by the `FLASH_ALLOC` macro and referenced by pointers but generated as normal variables. They are stored in the `persMemStaticFields` section. This section also offers 1024 bytes of memory. How many variables can be stored in there depends on their individual types. The last memory section defined in the former `pheap` region is called `persMemObjectTIB`. It is used for allocating the `TIB` and object structures. This section is the largest one providing 887804 bytes of persistent memory.

These memory sections are defined by adapting the linker script with which the JCOP system is configured. For every section an individual *attribute* is added. This *attribute* is then assigned to global variables in the *C* code to tell the linker in which memory section it must be put. Listing 4.21 depicts an example of how such attributes are used.

Listing 4.21: Example of attribute usage in *C* code

```c
int gExampleIntVariable __attribute__((section("persMemGlobVars")));
```

Adaptation of *C* Code Generation: With the newly defined FLASH memory sections the generating process of the JCF needs further adaptations. The pointers referencing the allocated `TIB` structures are put into the `persMemTIBRefs` section. Therefore, they can only be manipulated by using the `MEM_COPY` macro. This is especially important during the class initialization. Listing 4.22 shows the difference between the old class initialization and its new implementation.

Listing 4.22: Adaptation of `TIB` referencing

```
//-------------------- Old allocation and initialization --------------------//
__TIB_DEFINITION_test_doc_TestClass* __TIB_TestClass = JAVA_NULL;

void __INIT_test_doc_TestClass() {
  if (__TIB_test_doc_TestClass == JAVA_NULL){
    __TIB_test_doc_TestClass = FLASH_ALLOC(sizeof(__TIB__test_doc_TestClass));
    __INIT_IMPL_test_doc_TestClass();
  }
}

//-------------------- New allocation and initialization --------------------//
__TIB_DEFINITION_test_doc_TestClass* __TIB_TestClass
  __attribute__((section("persMemTIBRefs"))) = JAVA_NULL;

void __INIT_test_doc_TestClass() {
  if (__TIB_test_doc_TestClass == JAVA_NULL){
    JAVA_OBJECT pTemp = FLASH_ALLOC(sizeof(__TIB__test_doc_TestClass));
    MEM_COPY(&__TIB_TestClass, &pTemp, sizeof(JAVA_OBJECT));
    __INIT_IMPL_test_doc_TestClass();
  }
}
```

Additionally to this change of the global variables referencing the allocated `TIB` structures, the global pointers to the `SimpleClass` objects representing the individual classes must be modified as well. These modifications are depicted in Listing 4.23. Each `SimpleClass` object pointer is added with the `persMemClassRefs` attribute. Furthermore, the pointers are only manipulated using the `MEM_COPY` macro.

Listing 4.23: Adaptation of `SimpleClass` object referencing

```
JAVA_OBJECT __CLASS_test_doc_TestClass
  __attribute__((section("persMemClassRefs"))) = JAVA_NULL;

JAVA_OBJECT pTemp = XMLVM_CREATE_CLASS_OBJECT(__TIB_test_doc_TestClass);
MEM_COPY(&__CLASS_test_doc_TestClass, &pTemp, sizeof(JAVA_OBJECT));
```

The next change to the *C* code generating process is targeted on the global class variables. In the step in which the FLASH memory is simulated by the `gFlashMemArray`, the global class variables are allocated in the FLASH memory and referenced by global pointers. This change is reverted in the sense of declaring global class variables instead of pointers. However, these variables are put into the FLASH memory by assigning them the `persMemStaticFields` attribute. Therefore, they are no longer allocated using the `FLASH_ALLOC` but instead directly manipulated via the `MEM_COPY` macro. Listing 4.24 shows this modification in detail.

Listing 4.24: Shifting the static field allocation to real FLASH

```
//--------------------- Static field allocation ---------------------//
//old static field allocation
static JAVA_BYTE* _STATIC_test_doc_TestClass_staticField = JAVA_NULL;

void __INIT_IMPL_TestClass(){
...
//allocate static variable and assign default value to it
long long v = 0;
_STATIC_test_doc_TestClass_staticField = FLASH_ALLOC(sizeof(JAVA_BYTE));
MEM_COPY(_STATIC_test_doc_TestClass_staticField, &v, sizeof(JAVA_BYTE));
...
}

//new static field allocation
static JAVA_BYTE _STATIC_test_doc_TestClass_staticField
  __attribute__((section("persMemStaticFields"))) = 0;
```

The last change on the conversion process is made to add the information if a static class field is final or not to the IR. This data is added via a new attribute inside the `vm:field` tag of the `XML` files. Listing 4.25 visualizes how the `public final static byte staticFinalField` is represented.

Listing 4.25: IR of `staticFinalField`

```
<vm:field name="finalField" type="byte" isStatic="true" isPublic="true"
  isFinal="true"/>
```

Based on this new information the *C* generating process in adapted. As `final` static variables are not allowed to be changed during program execution, the value they are assigned to upon declaration is preserved during the whole program's life time. Therefore, in *C* code they can be declared as `const` variables. Furthermore, `const` variables are put into the `TEXT` section of the memory along with the program's native code which is put into the FLASH memory of the NXP platform. Listing 4.26 depicts the difference between the old *C* representation of a `final` class field and the new one.

Listing 4.26: Representation of `final` static field in *C*

```
//--------------------- Static field allocation ---------------------//
//old static field allocation
static JAVA_BYTE _STATIC_test_doc_TestClass_staticFinalField
  __attribute__((section("persMemStaticFields")));

//new static field allocation
const static JAVA_BYTE _STATIC_test_doc_TestClass_staticFinalField = 0;
```

All the modifications described in this paragraph are performed by adapting the `XSL` file. For the last adaptation also the scanning process of the byte code needs to be extended.

Mother Object: As already mentioned before, references to objects and `TIB` structures need to be persistent beyond the program's execution. To find the reference to the object from which to start the execution, a so-called *mother object* is added. In this step the *mother object* stores the address of the class instance holding the `main` function used as entry point to the test project. Furthermore, the `static void main(String[] args)` method is replaced by a virtual `void main()` method. Therefore, the call of this `main` function must be adapted in the generated *C* code. This new call is depicted in Listing 4.27.

Listing 4.27: Creation of *mother object* and call of main function

```
JAVA_OBJECT motherObject __attribute__((section("persMemMORef"))) = JAVA_NULL;
...
if(motherObject == JAVA_NULL){
  JAVA_OBJECT pTemp = __NEW_test_doc_Main();
  MEM_COPY(&motherObject, &pTemp, sizeof(JAVA_OBJECT));
}
test_doc_Main_main__(motherObject);
...
```

Object and TIB structure allocation: The last persistent memory section defined in the linker script, which is not yet explained in more detail, is the `persMemObjectTIB` region. Similar to the `gFlashMemArray` used in the Section 4.2.3 the `persMemObjectTIB` section is used for allocating the `TIB` structures as well as the object structures. To simplify things the `gFlashMemArray` is put into the `persMemObjectTIB` section and its size is adapted to fully cover this section. Furthermore, the `pFreeFlashMem` pointer indicating the beginning of the unallocated memory inside the `gFlashMemArray` is put into the `persMemGlobVars` section by using its attribute. With these adaptations only the manipulation of the `pFreeFlashMem` pointer must be modified. When allocating a new `TIB` structure or object it is changed by using the `MEM_COPY` macro. As the `gFlashMemArray` is still utilized, the persistent allocation does not change at all from the caller's perspective. The `FLASH_ALLOC` macro is still used for this.

Transient Memory: Another feature of *Java Card* is the possibility to create transient arrays. These arrays are not completely stored in FLASH memory as would be the case when creating an array using the `new` keyword. The *header* information of a transient array, such as its length and its type, is put into persistent memory. However, the data of the array is stored in the RAM. JCOP uses a dedicated memory section called the `theap` for saving the data of such transient arrays. This section is also utilized by the JCF to realize this feature. Similar to the `gFlashMemArray` a further array is declared. This `gTransientMemArray` is put into the `theap` by utilizing its memory section attribute. The allocation of space in this memory is performed in the same way as with the `gFlashMemArray` by declaring a `pFreeTransientMem` pointer and setting it to the end of the already allocated space inside the `gTransientMemArray`. This pointer is declared as a global variable and allocated persistently inside the `persMemGlobVars` memory section. Thus, after a card reset or power loss the pointer will still reference the end address of the latest allocated transient memory block in the `gTransientMemArray`. Thus, the already reserved memory for the data sections of the single transient arrays is not overwritten. The `gTransientMemArray` itself is zero initialized. The described functionality is provided by the means of the `xmlvm_mem_manager_TransientMemAlloc` function. The data stored in the `gTransientMemArray` can either be manipulated by using the `=` operator or by utilizing the `MEM_COPY` macro.

To create a transient array a new native function is implemented. This function creates the `SimpleArray` object representing the *Java* array in the `gFlashMemArray` as usual. But, instead of allocating the memory for storing the data in the persistent memory, it is stored by allocating space in the `gTransientMemArray`. Thus, the array object itself outlives the program's execution, but the data is lost as soon as the card is reset or the device loses power.

HAL Function Modification

Because of the changes performed on the `TIB`, object allocation and referencing, the HAL functions need to be adapted as well. First of all, the `MEM_COPY` macro needs to be modified. Until now, a self written `mem_manager_MemCpy` function is utilized. However, manipulating FLASH memory on the NXP platform must be performed by using a dedicated memory copy function. This functionality is provided by the JCOP HAL through the `phScalMem_copy` function. As this function also takes a destination pointer, a source pointer and a length value indicating the amount of bytes to copy as parameters, the `MEM_COPY` macro is used to abstract this function call. Therefore, from a caller's perspective nothing changes. However, as data located in the `gTransientMemArray` does not need to be manipulated using the special `phScalMem_copy` function, the `mem_manager_MemCpy` function performs a check, whether the provided destination address is located inside the `gTransientMemArray` or not. If the destination pointer references an address inside this array, a normal memory copy, as described in Section 4.2.2, is performed. Otherwise, the `phScalMem_copy` function is called. Figure 4.3 visualizes all the modifications performed on the memory layer during this step.

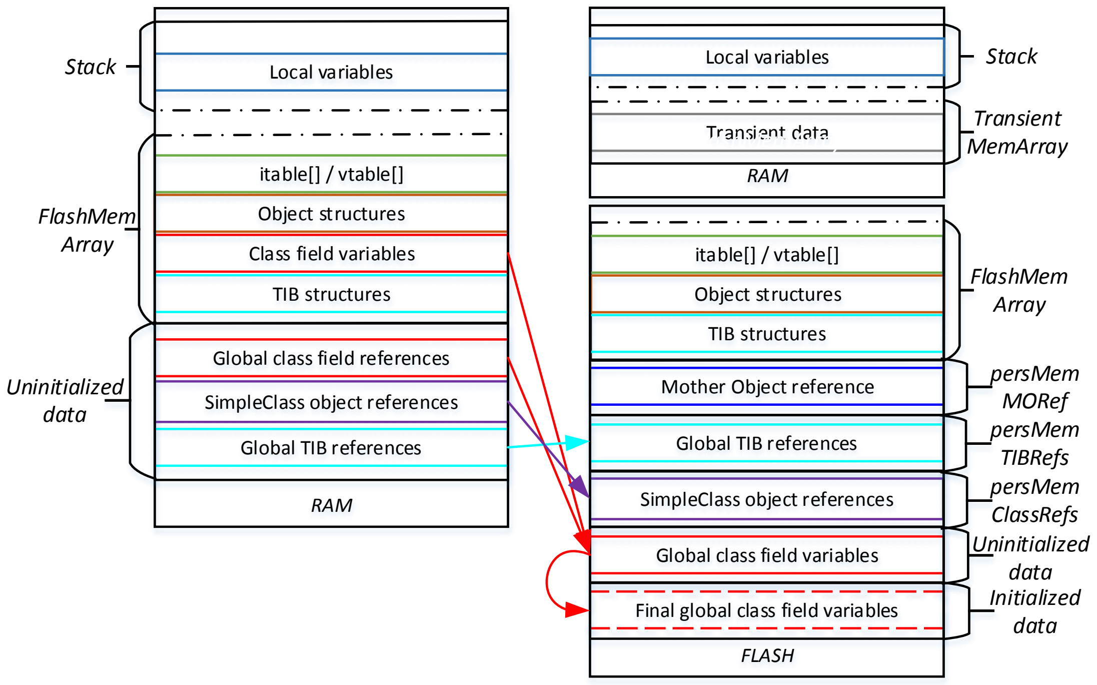


Figure 4.3: Memory layout of *Java Card* object life cycle simulation

Testcase persistent Counter

To test the realization of the *Java Card* object life cycle an additional test case is implemented. This test case consists of a `TestCounter` class which contains one instance field `int counter` which is initialized to 0. In each run of the test project this counter is increased. It is checked at the start up of the test project if the counter is not 0. Thus, it is verified that its value is retrieved from the persistent memory and not lost in between two executions. Another test case is added to check the allocation of transient and persistent arrays. It is checked at the start up of the test project if the data of an already existing transient array is set to 0 and if the data of an already existing persistent array is not reset. This test case verifies that the data stored by a persistent array outlives a card reset but data saved in a transient array, not.

Conclusion

With the adaptations described in this Section, the *Java Card* object life cycle is finally represented by the *C* representation generated by the JCF. This step enables the *Java Card* framework translation as well as converting a *Java Card* applet to native code. This last step is documented in Section 4.2.4.

4.2.4 JCOS and TestApplet

After successfully boarding the test project to the NXP platform and verifying that the changes performed to the memory allocation represents the *Java Card* object life cycle, the next step is to replace the *Java* framework with the *Java Card* framework. Furthermore, the test project launching the single test cases is replaced with a test applet as utilized in *Java Card*. For enabling the interaction with a terminal the receiving, dispatching and sending of APDUs must be implemented. This is accomplished by implementing a *Java Card Operating System* (JCOS).

Java Card Framework

Java Card provides its own framework and API on which the implementation of an applet is built. In the preceding steps the *Java* framework is converted, compiled and linked with the test project to provide it with the necessary classes. As in this step the target is to replace the test project with a test applet, the *Java* framework must be replaced with the *Java Card* framework. Because of the adaptations of the *Java* framework provided by XMLVM performed in Section 4.2.1, the differences to the *Java Card* framework are minor.

In *Java Card* an implementation of an applet is always performed by extending the `Applet` class. This `Applet` class comes with a range of dependencies not needed for the test project with which the test cases are executed up to this step. Furthermore, JCOP also provides its own API which is utilized in such an applet implementation. The *Java Card* framework as well as the JCOP API partly utilize functionality provided by the *Global Platform* (GP) framework. The GP is a public association publishing specifications for secure smart card chips. In order to not overload the converted applet with unnecessary functionality of APIs not needed, certain dependencies were cut by modifying the *Java Card* framework and the JCOP API. The adaptations must be performed in two classes. In the `OsInit` class of the `com.nxp.id.jcop.os` package the calls to the initialization functions of configuration dependent classes are removed. Furthermore, the configuration dependent instantiation of `Exception` classes is omitted. In the `Applet` class of the `javacard.framework` package the *final* `register` method is modified. This method is used to register the applet at the underlying JCVM. In this process the *Applet Identifier* (AID) is retrieved from a received APDU and processed. The method is adapted in such a way that this step is skipped.

Furthermore, the `String` class is removed from the conversion process. This removal has the effect that the debugging information is removed from the single test cases. Instead of printing the `String` information via UART, the result of a triggered test case is encoded in a dedicated byte array. This byte array is sent back to the terminal using the response APDU.

The modified version of the *Java Card* framework and the JCOP API still contain a range of dependencies not yet known to the JCF. These dependencies must be added to the `greenlist`. The following list shows the packages being added to the conversion process.

- Packages completely added to the greenlist:
  - com.nxp.id.jcop.annotations
  - com.nxp.id.jcop.javacard.security
  - com.nxp.id.jcopx.security
  - com.nxp.id.jcop.os
  - javacardx.crypto
  - javacard.framework
  - javacard.security

Furthermore, a range of different classes extending the `Exception` class as well as the `SystemUtils` class located in the `com.nxp.id.jcop.globalplatform.auxiliary` package is added. After adding these classes to the conversion process, a simple test applet is convertible.

TestApplet

The `TestApplet` used for replacing the test project from the previous steps must extend the `javacard.framework.Applet` class. Therefore, it must at least override the `install` and the `process` method. In the install method the `TestApplet` instantiates the single classes of the test cases and calls the `register` method of its base class. In the `process` method the `TestApplet` receives the APDU from the underlying OS layer by calling the `getBuffer` method of the APDU class. It then proceeds to dispatch the received APDU. Depending on the format of the APDU the selected test case is executed and its result is sent back. If the format is erroneous or no test case can be selected by the received APDU, an error code is sent back by the `TestApplet`.

Mother Object Container

As the `TestApplet` is now used as the main class from which the execution is started, the *mother object* must be adapted as well. There is no longer an instance `main` method which triggers the single test cases. Instead, this is performed by the `process` method of the `TestApplet`. To better handle the reference to the instance of the `TestApplet` the *mother object* is extended. In Section 4.2.3 the *mother object* is introduced as a mere reference to the main class instance of the test project. In this step a `MotherObjectContainer` class is implemented. This class contains an `applet` field for referencing an applet as well as an instance field for storing an `APDU` object. Furthermore, it provides a native `nativeInstallApplet` method which calls the `install` method of a predefined applet. It also provides a `dispatchApdu` method which calls the `process` method of the applet assigned to the `applet` field. Furthermore, this `dispatchApdu` method catches the exceptions thrown by the `process` method and creates a response APDU according to the result of this method. The `MotherObjectContainer` class must be added to the `greenlist`.

Native Function Implementation

The implemented `TestApplet` is converted with the framework classes indicated by the `greenlist`. However, the resulting *C* project is not yet compilable. This is due to native methods being declared by the added classes but not yet implemented. Most of these native methods are not even called during the execution of the `TestApplet`. Therefore, these methods are stubbed with *C* functions not containing any meaningful code. However, certain native methods must be implemented. The implementations of these methods are very close to native functions provided by the JCOP OS. These functions are described in the following paragraphs grouped by their classes.

OSUtils: The `short2Object` function receives an index and returns the object reference associated with this index. In this step it is only needed for returning the reference to the `exceptionArray` which stores an instance for each `Exception` class provided by the *Java Card* framework and the `exceptionReasonArray` which holds the reason codes for the single exception instances.

Applet: The `nativeRegister` function takes as a parameter the reference of the applet object to be registered and simply assigns the `applet` field of the `MotherObjectContainer` to this reference.

OsInit: The `setSelectionState` function sets the global `gcSelectonState` variable to the received value. The `getSelectionState` function returns this value to the calling function. The `setProcessingState` function sets the global `gcProcessingState` variable to the received value. The `getProcessingState` function returns this value to the calling function.

APDU: The `nativeGetCurrentAPDU` function checks if the `apdu` field of the `MotherObjectContainer` is already set. If not, it creates a new instance of the `APDU` class and assigns it to the `apdu` field. The `APDU.getBuffer` maps the `APDUBuffer char` array of the internal structure of the `NativeAPDU` provided by the JCOP HAL to a byte array represented by a `SimpleArray` object and returns it. The `setIncomingAndReceive` function sets the state of the `NativeAPDU` structure to `INCOMING` and returns the length of received APDU's data . The `getCurrentState` function simply returns the state field of the `NativeAPDU` structure.

APDUObj: The `getApduLengthType` function returns the `apduLengthType` field of the `NativeAPDU` structure. The `setOutgoingAndSend` function sets the field of the `NativeAPDU` structure according to the provided parameters and sets its state field to `OUTGOING`. The `setState` function sets the state field of the `NativeAPDU` structure to the provided value. The `getLe` function returns the expected length field of the `NativeAPDU` structure.

JCSystem: The `throwIt` function receives as parameters the index of the targeted exception object stored in the `exceptionArray` array and the reason why this exception is thrown encoded with a `short` variable. It checks if the indexed exception exists. If it exists, it retrieves the element of the `exceptionReasonArray` defined by the index parameters and sets it to the provided reason.

SystemUtils: The `setApduSW` function receives a *status word* (SW) to be sent back with the response APDU. Therefore, it sets the `sw` field of the `NativeAPDU` structure to the provided parameter.

JCOS

The JCOS as explained ins Section 3.2.4 must perform the tasks which are normally handled by the JCOP OS layer. It must receive and dispatch APDUs, initialize the necessary data structures, install and execute applet and send back the response APDU.

In more detail the JCOS performs exactly the same initialization of the HAL as done by the `main` function described in Section 4.2.3 on its initial start up. However, instead of calling the test project's `main` method, it immediately jumps into an endless loop. The first action inside this loop is to call the `receiveData` function provided by the HAL which continuously listens for an incoming APDU. On the first received APDU the initialization process of the HAL is finished and the `NativeAPDU` structure is initialized. Afterwards, the APDU's format is checked by utilizing the `apduDispatch` function provided by the JCOP HAL. If the format is correct, the JCOS checks whether the necessary structures for running the converted `TestApplet` are already instantiated.

If not, the JCOS creates a `MotherObjectContainer` object and assigns its address to the *mother object* pointer. Furthermore, it creates the `exceptionArray` as well as the `exceptionReasonArray`. Then it calls the `rom_init` method of the `OsInit` class. This method creates one instance for every exception class provided by the *Java Card* framework and stores them into the `exceptionArray` at their dedicated indexes. Then it initializes the `apdu` field of the `MotherObjectContainer` object by calling the `nativeGetCurrentAPDU` function and assigning its return value to this field.

If the necessary structures are already initialized, the `dispatchApdu` method of the `MotherObjectContainer` is called. In this method it is checked whether an applet is already installed and registered or not. If no applet is installed, the `nativeInstallApplet` is called. This method installs the predefined applet by calling its `install` method. Inside this method the applet calls the native `register` function which assigns the instance of the installed applet to the `applet` field of the `MotherObjectContainer`. If the applet is already installed and registered, the `MotherObjectContainer` checks if an applet is selected.

The `MotherObjectContainer` checks for each APDU, if it is a *select* command. If a *select* command is received the applet is selected by calling the `select()` method of the applet. Afterwards, the `process` method of the applet is called. If an APDU, which is no *select* command, is received and no applet has been selected, the `MotherObjectContainer` sends back a `FILE_NOT_FOUND` status word. The applet selection state is tracked by using a simple state machine. Figure 4.4 depicts the applet selection as a control flow diagram. This state is lost on a power loss or card reset. Therefore, at the beginning of each transaction the applet must be selected. This procedure was implemented, as applets generally require the `select` method to be executed before the `process` method can be run. After a successful applet selection, its `process` method is called with each received APDU. This method is provided the `apdu` field of the `MotherObjectContainer` as parameter.

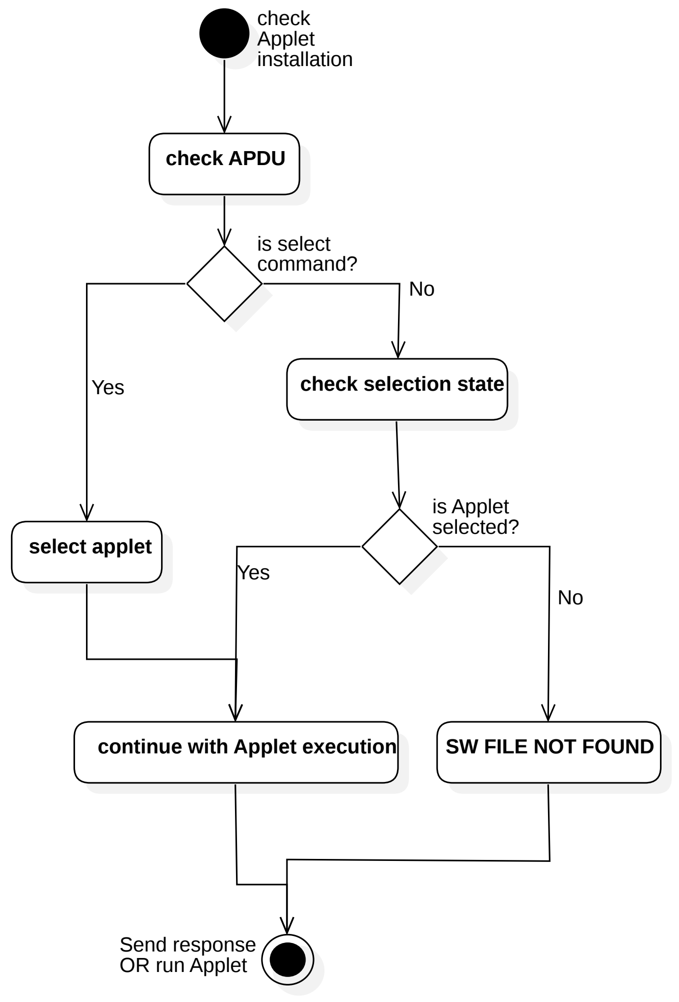


Figure 4.4: Execution Flow Diagram of the *Java Card* OS

After the applet's execution, the `dispatchApdu` method checks if the `process` function returned normally or if an exception was thrown. If an exception was thrown, its reason is set as SW of the APDU. This is accomplished by calling the `getReason` method of the exception object. This method accesses the `exceptionReasonArray` and retrieves the reason value stored at the index assigned to the exception. This reason value is set as SW of the APDU by calling the native `setApduSW` function. If no exception occurred, it is assumed that the response APDU was properly set up during the `process` method.

Finally, the state of the JCOS is set to `NO_PROCESS_IN_PROGRESS` using the native `setProcessingState` and to `NO_SELECTION_IN_PROGRESS` using the native `setSelectionState` function. With these last tasks performed, the `dispatchApdu` method returns and the JCOS continues with its execution.

Before sending back the response APDU set up by the applet or the `MotherObjectContainer`, the value set in the `sw` field of the `NativeAPDU` structure is appended to its `data char` array. Therefore, also the length must be set accordingly. After these final operations the response APDU is sent by calling the `sendData` function provided by the HAL. Thereafter, the execution jumps back to the `receiveData` function and listens for the next APDU sent to the card. Figure 4.5 depicts the complete execution flow of the JCOS. The state *check Applet selection* is a simplified abstraction of Figure 4.4

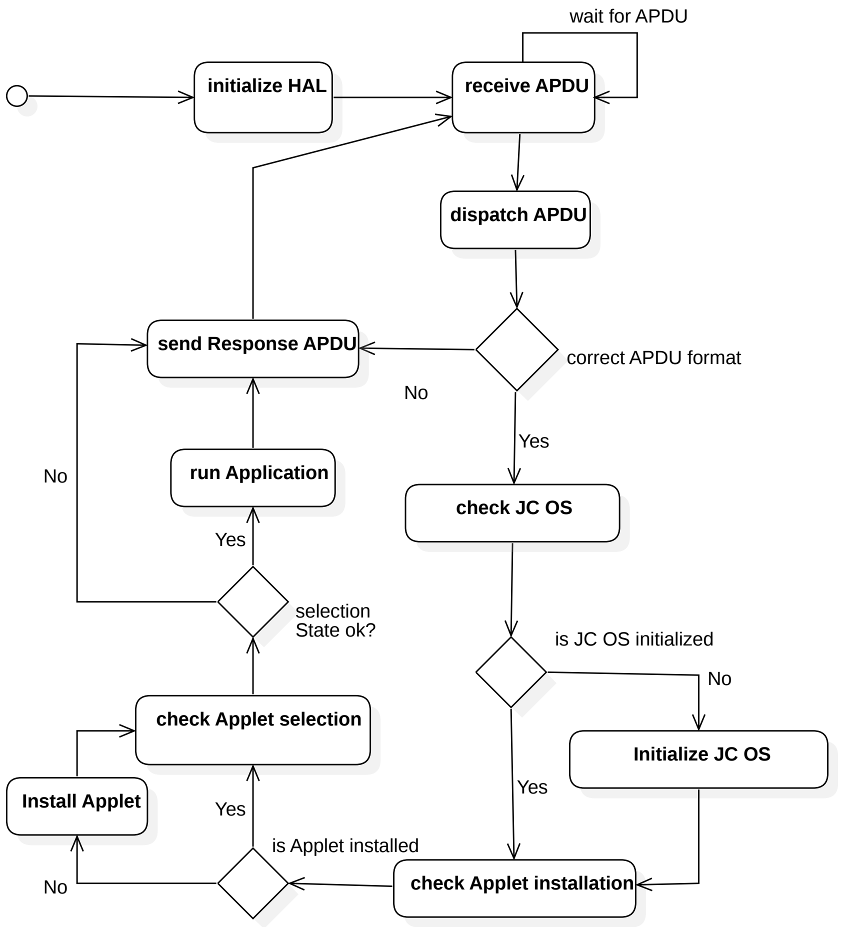


Figure 4.5: Execution Flow Diagram of the *Java Card* OS

Testcase Exception Handling and APDU

The features of the `exceptionArray`, the `exceptionReasonArray` and the APDU handling are tested by two additional testcases. The first test case deliberately throws exceptions using their `throwIt` method and setting reason values. The response APDUs are checked whether they hold the correct reasons as SWs. The second testcase inverts the data section of the received APDU and sends the inverted data back to the terminal.

Conclusion

With this last modification step, it is accomplished to convert a `TestApplet` extending the *Java Card* `Applet` class and using it to trigger the testcases implemented in the predecessor steps. Furthermore, the JCOS allows APDU handling similar to how it is performed by the JCOP OS.

4.3 Resulting Compilation Process

Summarizing, performing the modifications on the XMLVM framework described in this Chapter, a framework is created capable of translating a *Java Card* applet plus the *Java Card* framework classes it depends on to a *C* program. Utilizing a rudimentary *Java Card* OS, the JCOP HAL is initialized and thus the converted applet and the *Java Card* framework classes can make use of the functionality provided by the underlying layers. This *C* program is compiled using a standard *C* compiler. The resulting executable is small enough to fulfill the memory constrains of an NXP platform. Furthermore, the object life cycle is modified in such a way that it maps the requirements of the *Java Card* specification. In Section 1.3.4, a Figure is depicted which shows the layout of the JCOP system. In Figure 4.6,this same layout is visualized and furthermore it is indicated which parts are converted, which are natively implemented and which are omitted completely.

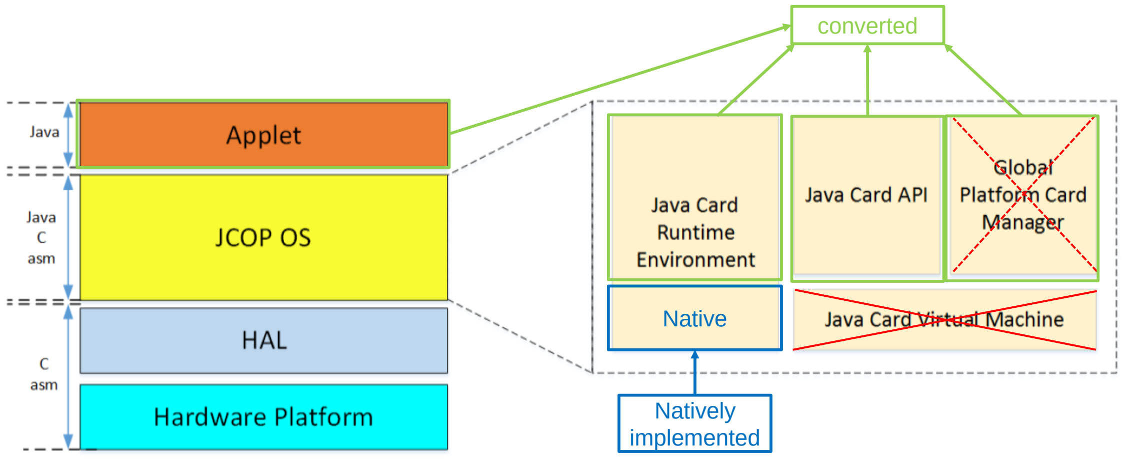


Figure 4.6: Execution Flow Diagram of the *Java Card* OS

The *Java Card* Runtime Environment which is part of the *Java Card* framework is partly converted and partly implemented as native functions. The *Java Card* API is converted using the JCF. The GP Card Manager part is omitted in this version of the JCF but can be converted as well. The JCVM is completely removed.

For additional applets to be converted and natively compiled for an NXP platform, the *Java Card* framework classes must be added to the conversion process and their native function calls must be implemented, if necessary.

Chapter 5

Evaluation

Concluding the thesis, the *Evaluation* chapter describes both benefits and drawbacks of precompiling *Java Card* code using the JCF compared to interpreting it using a VM. The single aspects of the evaluation are explained in the single sections of this chapter. After the implementation of the JCF, the performance of the compiled applets is compared to the performance of the interpretation of their byte code representation using the JCVM. This comparison and a detailed explanation of the tests performed to verify the correct translation of the *Java Card* code can be found in Section 5.1. Furthermore, also the impact of the translation process on the various security mechanisms provided by the JCVM is explained in Section 5.2.

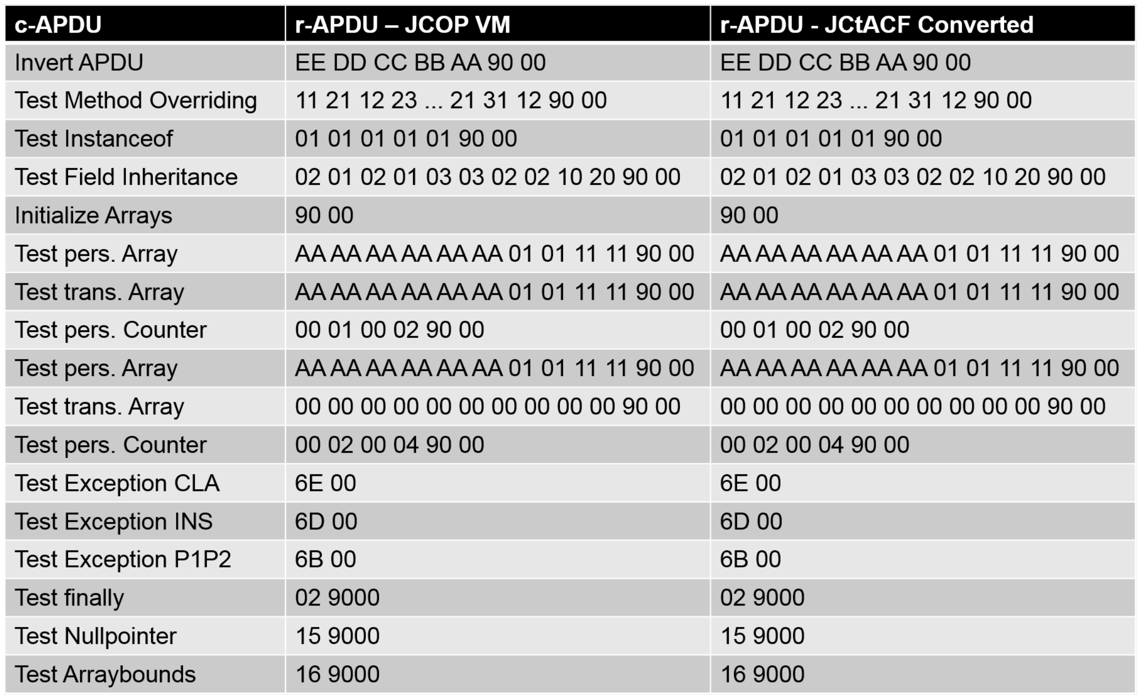
5.1 Performance Evaluation and Testing

In this section, the testing of the conversion process of the JCF is explained in more detail. The tests are basically used to verify that the execution of an applet, which is precompiled by the JCF, leads to the same result as interpreting its byte code using the JCVM. Furthermore, the performance of the precompiled applet is compared to the performance of its interpreted version. In this section, the resulting code size of the compiled applet is collated with the size of its byte code representation as well.

5.1.1 JCF Translation Testing

In this section, the results of the `TestApplet` being interpreted by the JCVM are compared to the results received when executing the natively compiled applet directly on the chip. The table depicted in Figure 5.1 shows the r-APDUs sent back by the interpreted and the compiled `TestApplet` when triggering the single testcases. The *Invert APDU* testcase sends an c-APDU containing some data to the card. The `TestApplet` inverts the data and sends it back in the r-APDU. The *Test Method Overriding* testcase checks, if the methods are correctly overridden by their extending classes. Each method of a class instance in the hierarchy appends its own specially encoded byte to the data of the r-APDU. This data is then sent back to the terminal. The *Test Instanceof* testcase checks if an object is correctly identified being an instance of a certain class/interface or not. The result of this check is set to `0x01`, if it is true and to `0x00`, if it is false. Each result is appended to the data of the r-APDU. At the end, the results are sent back. The *Test Field Inheritance* testcase checks, if the single instance fields are inherited from a base class to its extending class. The values of the single fields of the classes, represented in the class hierarchy, are put into the data section of the r-APDU and sent back. The *Initialize Arrays* command initializes the persistent and transient arrays and creates two instances of the `TestCounter` class. The testcases six, seven and eight are triggered two times in a row. In between a card reset is performed. The *Test persistent Array*testcase returns the data of the persistent array, the *Test transient Array* testcase returns the data of the transient array and the *Test persistent Counter* testcase increases object fields of the two `TestCounter` objects (the first by one, the second by two) and sends back their values. At the first execution both the arrays contain the initialized values. Test counter one is set to one and test counter two is set to two. After the reset the persistent array still contains the data it was initialized with, but the transient array's data is lost. Therefore, only `0x00` is returned. Test counter one is now increased to two and test counter two is set to four. The next three testcases trigger exception handling by causing `ISOException`s that are thrown with different reason codes. Each of the three testcases additionally checks, whether the JCF correctly converts the catching of exceptions thrown by a called method. The *Test Exception CLA* testcase causes a "`CLA` not known" error and expects the `SW_CLA_NOT_SUPPORTED` (`0x6E00`) to be returned. The *Test Exception INS* testcase causes a "`INS` not known" error and expects the `SW_INS_NOT_SUPPORTED` (`0x6D00`) to be returned. The *Test Exception P1P2* testcase causes a "`P1P2` not known" error and expects the `SW_P1P2_NOT_SUPPORTED` (`0x6B00`) to be returned. The *Test finally* testcase checks, whether the `finally` block of a `try - catch` block is correctly entered. The *Test Nullpointer* testcase deliberately accesses an unassigned object and expects an `NullPointerException` to be thrown. The *Test ArrayBounds* testcase tries to access an element within an array using an index that is outside the bounds of the array. The testcase expects an `ArrayIndexOutOfBoundsException` to be thrown. As it can be seen in Figure 5.1, both the interpreted as well as the compiled version of the `TestApplet` return exactly the same r-APDUs when the single testcases get triggered.

Additionally to the `TestApplet`, a *banking applet* is compiled using the JCF. This *banking applet* is used as a reference for benchmarking payment transactions in the industry. This applet is personalized with key material in a first phase and then performs asymmetric cryptographic operations for securing the transaction process. This applet is chosen because of two reasons. Firstly, by being able to compile this *banking applet*, it can be shown that the JCF is fully operational and can be used to compile *Java Card* applets utilized in the industry. Secondly, banking applets need to meet a very demanding timing criteria. Thus, it is not only shown that the JCF's compilation process is fully functional, but also that the precompiled applet's performance is superior to its interpreted version. Therefore, translating such an applet with the JCF can greatly help to meet the given performance goals.



c-APDU	r-APDU – JCOP VM	r-APDU - JCtACF Converted
Invert APDU	EE DD CC BB AA 90 00	EE DD CC BB AA 90 00
Test Method Overriding	11 21 12 23 ... 21 31 12 90 00	11 21 12 23 ... 21 31 12 90 00
Test Instanceof	01 01 01 01 01 90 00	01 01 01 01 01 90 00
Test Field Inheritance	02 01 02 01 03 03 02 02 10 20 90 00	02 01 02 01 03 03 02 02 10 20 90 00
Initialize Arrays	90 00	90 00
Test pers. Array	AA AA AA AA AA AA 01 01 11 11 90 00	AA AA AA AA AA AA 01 01 11 11 90 00
Test trans. Array	AA AA AA AA AA AA 01 01 11 11 90 00	AA AA AA AA AA AA 01 01 11 11 90 00
Test pers. Counter	00 01 00 02 90 00	00 01 00 02 90 00
Test pers. Array	AA AA AA AA AA AA 01 01 11 11 90 00	AA AA AA AA AA AA 01 01 11 11 90 00
Test trans. Array	00 00 00 00 00 00 00 00 00 00 90 00	00 00 00 00 00 00 00 00 00 00 90 00
Test pers. Counter	00 02 00 04 90 00	00 02 00 04 90 00
Test Exception CLA	6E 00	6E 00
Test Exception INS	6D 00	6D 00
Test Exception P1P2	6B 00	6B 00
Test finally	02 9000	02 9000
Test Nullpointer	15 9000	15 9000
Test Arraybounds	16 9000	16 9000

Figure 5.1: Table of r-APDUs of interpreted and compiled `TestApplet`

5.1.2 Performance Measurements and Code Size Comparison

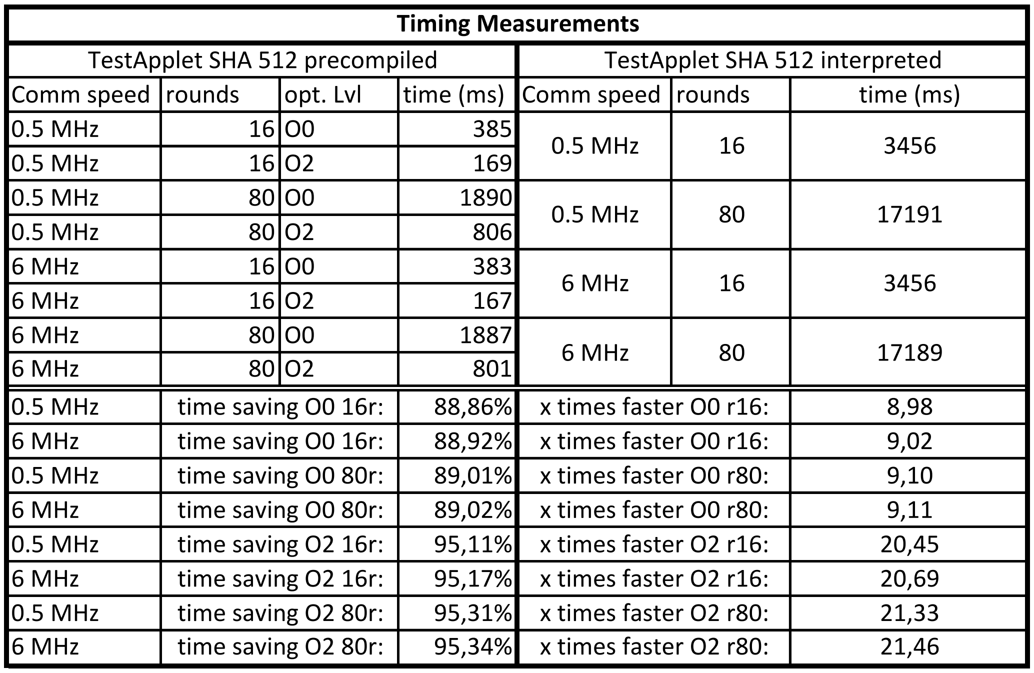
To measure the performance increase achieved by the pre-compilation, two applets are translated using the JCF. Both the execution times of the interpreted applets and the natively compiled applets are then measured and compared to each other. In each time measurement, the execution time of the whole transaction is recorded. This execution time does not include the processing time that is needed by the host. To diminish the distortions, which might occur in a single transaction, each transaction is measured 20 times and the mean value of the aggregated execution times is calculated. This mean value is then used for the comparison.

The execution time is affected by two factors. The first factor which must be taken into account is the communication speed. This speed influences the timing measurement as complete transactions are measured. To evaluate the impact of the communication speed on the transactions the interpreted and natively compiled applets are tested using two different speed adjustments: 0.5 MHz and 6 MHz. The second factor, which only influences the execution speed of the natively compiled applets, is the optimization level of the utilized ARM compiler. The used compiler lets you choose from four different optimization levels. From this range (0 to 3), the optimization levels 0 and 2 are chosen. *Level 0* performs no optimization of the code but only simple source code transformations. It also does not impact the *debug view* when executing the code, which, therefore, implies that every line of source code is compiled. *Level 2* performs high optimization of the source code and is the default optimization level. Additionally to this optimization level also the `Otime` compiler option is enabled. This option lets the compiler perform even more aggressive optimization that also possibly affects the size of the emitted object code.

If the `Otime` is not set, the compiler automatically optimizes for object code size [2], [1]. *Level 2* optimization with `Otime` compiler option is also used for the underlying layers.

TestApplet and SHA512 Hash Algorithm

The first applet,that is converted performs a SHA512 hashing operation on some received data. This algorithm is completely implemented in *Java Card* and was written by *Eric Larchevque*, *Nicolas Bigot*, *Nicolas Dorier* and *Pierre Pollastri*. The source code can be found at their *github* web page [1]. Both the interpreted and the compiled version of the hashing algorithm are executed for 16 and 80 rounds, respectively. The detailed timing measurements are shown in Figure 5.2.



Timing Measurements

TestApplet SHA 512 precompiled				TestApplet SHA 512 interpreted		
Comm speed	rounds	opt. Lvl	time (ms)	Comm speed	rounds	time (ms)
0.5 MHz	16	O0	385	0.5 MHz	16	3456
0.5 MHz	16	O2	169			
0.5 MHz	80	O0	1890	0.5 MHz	80	17191
0.5 MHz	80	O2	806			
6 MHz	16	O0	383	6 MHz	16	3456
6 MHz	16	O2	167			
6 MHz	80	O0	1887	6 MHz	80	17189
6 MHz	80	O2	801			

0.5 MHz	time saving O0 16r:	88,86%	x times faster O0 r16:	8,98
6 MHz	time saving O0 16r:	88,92%	x times faster O0 r16:	9,02
0.5 MHz	time saving O0 80r:	89,01%	x times faster O0 r80:	9,10
6 MHz	time saving O0 80r:	89,02%	x times faster O0 r80:	9,11
0.5 MHz	time saving O2 16r:	95,11%	x times faster O2 r16:	20,45
6 MHz	time saving O2 16r:	95,17%	x times faster O2 r16:	20,69
0.5 MHz	time saving O2 80r:	95,31%	x times faster O2 r80:	21,33
6 MHz	time saving O2 80r:	95,34%	x times faster O2 r80:	21,46

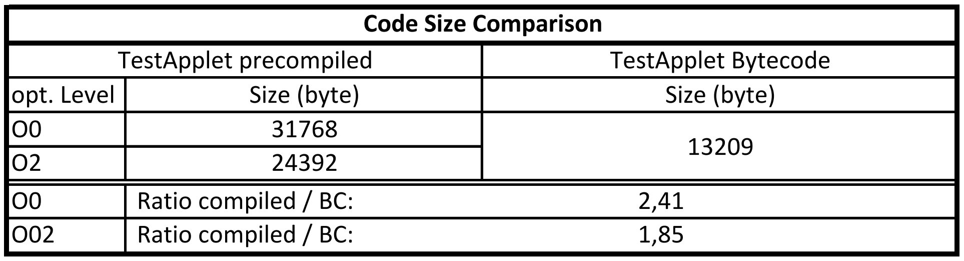
Figure 5.2: Time measurements of SHA512 execution

As it can be seen in the table, the communication speed hardly influences the execution of the tested hash algorithm. The slower communication frequency merely adds an static overhead of two to five milliseconds to the overall execution time of the single transactions. This overhead is independent of how many hash calculations are performed per transaction or the optimization level of the compiler. However, the optimization level greatly influences the execution speed of the precompiled test applet. Using *level 2* optimization the transaction is more than two times faster compared to its *level 0* optimized version. This big impact on the execution speed is largely caused by the fact that the hash algorithm is purely implemented in *Java Card*. Therefore, the whole hash computation, which makes up most of the execution time, is affected by the optimizations applied to

[1]SHA512: `https://github.com/LedgerHQ/ledger-javacard/blob/master/src-preprocessed/com/ledger/wallet/SHA512.javap`

the natively compiled code. Native functions implemented in a lower layer would not be affected by the used optimization level. As the implementation of the hash algorithm does not utilize any native functions for its computations, the precompilation of the *Java Card* code greatly improves the performance of the SHA512 implementation. As depicted in Figure 5.2, the natively compiled applet executes up to 21.46 times faster than the same applet being interpreted by the JCVM. This means that the compiled applet reduces the execution time by up to 95.34% of its interpreted version.

Considering the size of the *Java Card* byte code of the applet and comparing it to the size of the object code of the natively compiled version, the precompiled applet needs 1.85 times more space than the byte code representation. The exact values for both the byte code and the object code can be seen in Figure 5.3. Also the code size is largely affected by the optimization level of the *C* compiler. Optimization *level 2* saves, in this case, 30.24% of size compared to optimization *level 0*.

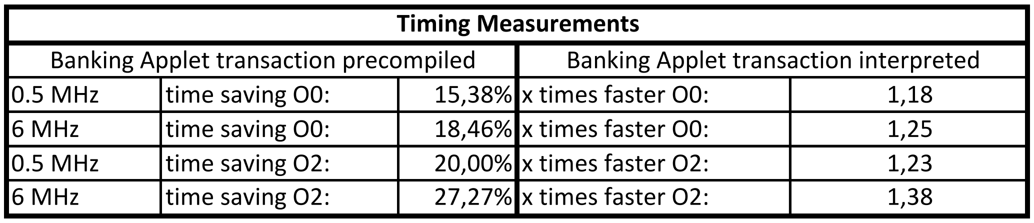


Code Size Comparison		
TestApplet precompiled		TestApplet Bytecode
opt. Level	Size (byte)	Size (byte)
O0	31768	13209
O2	24392	
O0	Ratio compiled / BC:	2,41
O02	Ratio compiled / BC:	1,85

Figure 5.3: Code size considering the TestApplet

Reference *Banking Applet*

The second applet that is used for comparing both code size and execution speed is the reference *banking applet*. As described in Section 5.1.1 this applet is personalized with key material that is then used to perform asymmetric cryptographic operations needed for secure transactions. Thus, the *banking applet* knows a personalization and a transaction phase. The time measurements are only focusing on the transaction phase itself. Figure 5.4 depicts the impact of the precompilation on the transaction's performance. It can be seen that the transaction of the compiled *banking applet* executes 15.38% to 27.27% faster compared to the interpreted version of the applet. Similar to the SHA512 hash algorithm, the *banking applet* benefits from the optimization level of the compiler as well. However, this optimization has a much lower impact on the *banking applet* than on the implementation of the hash algorithm. This is due to the ample utilization of native functions by the *banking applet*. All the time consuming cryptographic operations are already implemented in *C*. Therefore, the native compilation of this applet cannot achieve that much speedup compared to the SHA512 algorithm which is purely written in *Java Card*. As these native functions are provided by the lower layers of the system, the optimization levels applied to the natively compiled *banking applet* have a lower performance impact as well. As the transaction of the *banking applet* itself also takes less time than the execution of the SHA512 with 16 or 80 rounds respectively, the communication speed has a greater impact on the overall performance.

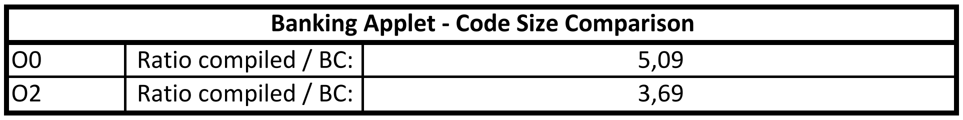


Timing Measurements				
Banking Applet transaction precompiled			Banking Applet transaction interpreted	
0.5 MHz	time saving O0:	15,38%	x times faster O0:	1,18
6 MHz	time saving O0:	18,46%	x times faster O0:	1,25
0.5 MHz	time saving O2:	20,00%	x times faster O2:	1,23
6 MHz	time saving O2:	27,27%	x times faster O2:	1,38

Figure 5.4: Time measurements of the reference *banking applet*

As this time measurement shows, the speed up that can be achieved by precompiling *Java Card* applets is determined by how much runtime is spent in natively implemented functions. The more an applet utilizes these functions the less speed up is achieved, as they are already implemented in *C* and natively compiled for the chip.

Furthermore, also the code size of the natively compiled *banking applet* and its byte code representation is compared. The exact ratios can be seen in Figure 5.5. It can be recognized that also in the case of the *banking applet* the size of the object code is greatly influenced by the optimization level of the compiler. With optimization *level 2* the compiler produces an object code 1.4 times smaller than with *level 0*.



Banking Applet - Code Size Comparison		
O0	Ratio compiled / BC:	5,09
O2	Ratio compiled / BC:	3,69

Figure 5.5: Time measurements of the reference *banking applet*

5.2 Security Evaluation

This section approaches the lack of security mechanisms that is caused by the translation process of the *Java Card* applets. Alternative ideas of how to design and integrate such mechanisms in the generated code are presented.

JCOP implements a range of features ensuring the security of the overall system. These security mechanisms are placed both in software as well as in hardware. As the compilation process proposed in this master thesis only influences the software part, the security of the hardware is not examined any further. Furthermore, as the code generated by the JCF is not bound to the targeted system the underlying hardware and its security mechanisms might change. Considering the software security features the JCOP system provides, it must be mentioned that most of them are placed in the HAL and the OS layer. As the functionality provided by the HAL is not changed, the security mechanisms of this layer are not undermined. However, the security features, which are part of the JCRE and the JCVM are missing in the *C* code generated by the JCF.

The most important security feature implemented by the JCVM is the firewall. This firewall ensures the isolation of installed applets and therefore, is also referred to as the *applet firewall*. A JCVM allows more than just one applet to be uploaded and installed. Because the objects allocated by an applet are put into persistent memory, they might outlive the applet's execution time, if not garbage collected by the runtime environment. Therefore, it must be ensured that each applet is isolated. This means that an applet cannot access or use the class instances allocated by other applets. Hence, an applet can only access the virtual methods and fields of objects which reside in its context. This security mechanism is fundamental. However, there must also exist a way with which certain dedicated objects can be shared between applets, guaranteeing interoperability. This functionality is provided by the JCRE by using a concept known as *shareable interface objects*. The JC application programming interface (JCAPI) provides an interface called `Shareable`. All objects which directly or indirectly implement this interface can pass the *applet firewall* [16]. Furthermore, certain class instances of the JCRE are not restricted by the firewall and, therefore, can be used by all applets [17], [18].

This *applet firewall* is not represented in the current version of the JCF, but must be implemented in case that more than one precompiled applet shall be present on the smart card at a time. Such an isolation of the single applets could be achieved by defining dedicated memory sections for each applet in which its objects are allocated and a single memory section in which the shareable class instances of the JCRE are located. Before accessing any object during applet execution its address would have to be checked. Furthermore, it must be determined if the object is an instance of the `Shareable` interface. This approach, however, is a mere idea of how such a firewall could be added to the JCF and is by no means a design ready for implementation. If the further development of the JCF will be driven into the direction of a complete OS handling the execution of multiple precompiled applets, implementing such a firewall is inevitable.

As the JCF translates *Java Card* classes to *C* code and this translation process is fully controllable the software security mechanisms that are implemented in the native part of the JCOP OS and the HAL can simply be added to the generated *C* code. Therefore, also the code written in *Java Card* could be secured by utilizing these features.

Another security mechanism which is imposed by *Java Card* is the byte code verifier. According to the *Java Card* specification, this verification can be performed either on the targeted embedded system itself after uploading the byte code or before uploading the byte code on an external device. This greatly depends on the available memory space of the targeted system. Considering the JCF the byte code can be verified on an external device before it is being translated and loaded to the embedded system. The correctness of the uploaded native code then depends on security of the connection between the external device and the smart card.

Chapter 6

Conclusion and Future Work

Concluding the thesis, this chapter describes both benefits and drawbacks of precompiling *Java Card* code using the JCF, thus, drawing a conclusion of the work. The first section of this chapter, Section 6.1, outlines the restrictions of the JCF. Building upon these restrictions, the conclusion, which can be found in Section 6.2, describes the more imminent further development of the JCF. Furthermore, the possible future development of the framework is discussed in Section 6.3. The development discussed in this section must be seen as a long term target.

6.1 Restrictions

The version of the JCF described in this thesis is, of course, restricted. As all *Java Card* classes (applet plus runtime environment) are converted to a single *C* program, the *Java Card* applet is bound to the compiled version of the *Java Card* framework. Furthermore, loading another applet to the card, without loading the *Java Card* framework as well, is not possible at this state. The system at this version allows only one applet to be loaded and executed on the smart card. Furthermore, the security mechanisms performed by the JCVM are not yet implemented. These mechanisms are described in more detail in Section 5.2. Furthermore, only partly compiling *Java Card* classes, such as performed by *Mixed Mode AOTCs*, is not possible with the JCF at the current state, as the JCVM is completely removed from the JCOP OS and thus no interpretation of byte code is possible. Hence, also an interaction with a present JCVM, as usually performed by *Mixed Mode AOTCs*, is not possible. Furthermore, the way of representing the *Java Card* classes in the resulting *C* program greatly differs from their representation by the JCVM, which increases the effort that would be needed to create a hybrid system, in which *Java Card* classes are partly precompiled, but interpreted as well.

Another restriction, which must be mentioned, is that it is not possible to translate *Java Card* byte code with the current version of the JCF. The *Java Card* classes, which are compiled by the framework, come in form of *Java* byte code. This actually makes no difference considering the source code itself, as *Java Card* byte code is created from *Java* byte code and is merely a further compressed format [24]. Thus, there is no difference in the program flow when comparing an applet's *Java* byte code to its *Java Card* byte code, as the *Java Card* byte code is a subset of the *Java* byte code. However, the JCF would not be able to correctly convert an applet's `.cap` file at the current state, as *Java Card* simply uses different encodings for the byte code instructions [18]. These would not be understood by the JCF.

6.2 Conclusion

In this master thesis a framework, which is capable of natively compiling *Java Card* applets and runtime classes, is presented. This compilation process is performed by converting the byte code representation to *C* source code and utilizing a standard *C* compiler for the native compilation. The resulting framework is not meant to be a final product. It only shows that the native compilation of *Java Card* applets is possible and a performance improvement can be achieved. However, there are also several drawbacks when AOT-compiling *Java Card* applets. The most significant disadvantage of this method is its additional memory consumption. The JCOP JCVM offers the possibility of storing several applets at the same time. As the memory capacity of smart cards is generally limited, the upload of more than one applet, precompiled by the JCF, would exceed the storage capacity of most of today's smart card chips. Therefore, such an approach is not feasible at the moment. Another drawback is the inflexibility of the standalone AOT-compilation performed by the JCF in its current version. This drawback, which is already outlined in Section 6.1, must be resolved as well, if the precompilation of applets is to be integrated into future products.

Furthermore, Section 6.1 outlines that in the current version of the JCF, the *Java Card* applets are compiled by converting their *Java* byte code representation. External applets are usually represented in *Java Card* byte code. To enable the precompilation of such *Java Card* applets, the JCF would need to be able to convert *Java Card* byte code. This modification would have to be seen as an imminent further development, if the precompilation of applets was targeted.

6.3 Future Work

The further development of the JCF mainly depends on the future improvements of smart cards. Especially the amount of physical memory, available on next generation smart cards, plays a major role. Basically, there are two different ways, in which the JCF's advancement can proceed.

The first approach is targeted to the near future and aims at natively compiling the *Java Card* classes of the JCAPI and the GP API. This approach is interesting as it offers the possibility of precompiling a huge amount of *Java Card* source code, present on every single smart card. This precompilation would not need any change to the process of uploading *Java Card* applets. As the functionality provided by the APIs are fundamental and, therefore, utilized by every professional applet, a considerable performance improvement could be gained by their precompilation. As the APIs are usually preloaded on the smart card, the upload of *Java Card* applets, which is usually performed by external customers, would stay the same. However, to realize this approach, the JCVM would need to be changed in such a way that it is able to interact with the class and object representation of the JCF. Furthermore, the JCF's method representation would need to be changed in such a way that the parameters are not provided in the function header, but instead taken from the JCVM's operand stack. Hence, also the return value would have to be pushed on this stack.

The second approach would not only precompile the JCAPI and the GP API classes, but also partly compile the single applets. This partly compilation would focus on certain methods of an applet, in which an ample amount of execution time is spent. However, this approach would make an adaptation of the upload process necessary. Furthermore, the JCF would need to be changed from a *Standalone AOTC* to a *Mixed Mode AOTC* and the JCVM would have to be changed in such a way that it can call the natively compiled methods of the applet. The JCOP proprietary class and object representation would be utilized in the precompiled code as well. Therefore, the JCF translation process would have to be modified to generate code, which uses this special representation. The partly compilation of the applets would be used to save memory space.

The two approaches include major adaptations of both the JCVM and the JCF. However, the first proposed approach would omit modifications to the applet upload process. Therefore, it would make more sense to track this approach as no adaptations on the customer side would be necessary. Furthermore, the adaptations are of less impact compared to the changes needed to be performed, when following the second approach.

Bibliography

[1] *ARM - MDK-ARM C/C++ Compiler*. URL: `http://www.keil.com/support/man/docs/uv4/uv4%7B%5C_%7Ddg%7B%5C_%7Dadscc.htm` (visited on 11/29/2016).

[2] *ARM - Optimization levels and the debug view*. URL: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.kui0097a/armcc%7B%5C_%7Dcjaieafa.htm` (visited on 11/29/2016).

[3] Per Bothner. *Compiling Java with GCJ*. 2003. URL: `http://www.linuxjournal.com/article/4860` (visited on 07/18/2016).

[4] *Comparison of Dalvik and Java Bytecode*. 2012. URL: `https://forensics.spreitzenbarth.de/2012/08/27/comparison-of-dalvik-and-java-bytecode/` (visited on 10/25/2016).

[5] Seth Cousins. *Android to Include Ahead-Of-Time Compiler*. 2014. URL: `https://www.infoq.com/news/2014/07/ahead-of-time-compiler-os` (visited on 08/17/2016).

[6] *Dalvik bytecode*. URL: `https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html` (visited on 10/26/2016).

[7] Jonathan Day. *What are advantages and disadvantages of using The LLVM Compiler Infrastructure Project as a development infrastructure in comparison to GNU toolset?* 2015. URL: `https://www.quora.com/What-are-advantages-and-disadvantages-of-using-The-LLVM-Compiler-Infrastructure-Project-as-a-development-infrastructure-in-comparison-to-GNU-toolset` (visited on 08/04/2016).

[8] *Excelsior JET*. URL: `http://www.excelsiorjet.com/` (visited on 07/18/2016).

[9] *GCJ Guide*. URL: `https://gcc.gnu.org/java/gcj2.html` (visited on 07/21/2016).

[10] Nicolas Geoffray, Gaël Thomas, and Julia Lawall. “VMKit: a substrate for managed runtime environments”. In: *ACM Sigplan ...* 45.7 (2010), pp. 51–62. ISSN: 0362-1340. DOI: `10.1145/1837854.1736006`. URL: `http://doi.acm.org/10.1145/1837854.1736006%7B%%7D5Cnhttp://dl.acm.org/citation.cfm?id=1736006`.

[11] Lukas Gressl. “Data Management of banking applications on smart cards”. In: (2016).

[12] Sumit Gupta, Nargish Gupta, and Rishabh Gupta. “Objects and Method Calling in Java Virtual Machine”. In: *International Journal of Computer Applications* 86 (2014), pp. 34–36.

[13] John L Hennessy and David a Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* 2006, p. 704. ISBN: 0123704901. DOI: 10.1.1.115.1881. URL: http://portal.acm.org/citation.cfm?id=1200662.

[14] Masahiro Ide. "Study on method-based and trace-based just-in-time compilation for scripting languages". In: (2015).

[15] *ISO/IEC 9899: TC3.* 2007.

[16] *Java Card 3 Platform, Application Programming Interface, Classic Edition.*

[17] *Java Card 3, Platform Runtime Environment Specification, Classic Edition.* September. Oracle, 2011.

[18] *Java Card 3 Platform, Virtual Machine Specification, Classic Edition.* September. Oracle, 2011.

[19] Dong-Heon Jung, Soo-Mook Moon, and Sung-Hwan Bae. "Design and Optimization of a Java Ahead-of-Time Compiler for Embedded Systems". In: *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing* (2008), pp. 169–175. DOI: 10.1109/EUC.2008.80. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4756335.

[20] Peter Klingebiel. *C Standard-Bibliothek.* 1995. URL: http://www2.hs-fulda.de/%7B~%7Dklingebiel/c-stdlib/setjmp.htm (visited on 11/10/2016).

[21] Stefan Krause. *JAVA VS. C BENCHMARK #2: JET, HARMONY AND GCJ.* 2007. URL: http://www.stefankrause.net/wp/?p=6 (visited on 07/18/2016).

[22] Krishan Kumar. *Memory Layout of C Program.* URL: http://cs-fundamentals.com/c-programming/memory-layout-of-c-program-code-data-segments.php (visited on 11/10/2016).

[23] Chris Lattner. *LLVM.* URL: http://www.aosabook.org/en/llvm.html (visited on 08/04/2016).

[24] Andreas Lessiak. "Performance Optimization of the JCOP Java Card Operating System based on HW/SW Co-Design". PhD thesis. Technical University Graz, 2008.

[25] Yeong Kyu Lim et al. "A selective ahead-of-time compiler on android device". In: *2012 International Conference on Information Science and Applications, ICISA 2012* (2012), pp. 1–6. DOI: 10.1109/ICISA.2012.6220938.

[26] Tim Lindholm et al. "The Java® Virtual Machine Specification". In: (2014), pp. 1–626. URL: http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.

[27] *LLVM Java Frontend Documentation.* URL: https://llvm.org/svn/llvm-project/java/trunk/docs (visited on 07/19/2016).

[28] *[LLVMdev] VMKit is retired (but you can help if you want!)* URL: https://groups.google.com/forum/%7B%5C#%7D!msg/llvm-dev/1FvBAURb4q0/dwl7Mli3wAgJ (visited on 07/20/2016).

[29] Activity Manager. *Anatomy of Android.* 2015. URL: https://anatomyofandroid.com/2013/10/15/zygote/ (visited on 06/21/2016).

[30] Torben Ægidius Mogensen. *Basics of Compiler Design*. Vol. 3. 2. 2009, pp. 1–23. ISBN: 9788799315406. URL: `http://www.diku.dk/hjemmesider/ansatte/torbenm/Basics/basics%7B%5C_%7Dlulu2.pdf`.

[31] Hyeong-Seok Oh, Ji Hwan Yeo, and Soo-Mook Moon. "Bytecode-to-C Ahead-of-time Compilation for Android Dalvik Virtual Machine". In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition* (2015), pp. 1048–1053. ISSN: 15301591. URL: `http://dl.acm.org/citation.cfm?id=2757012.2757057`.

[32] TA Proebsting et al. "Toba: Java for applications: A way ahead of time (WAT) compiler". In: *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS'97)* (1997), pp. 41–54.

[33] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*. Third Edit. John Wiley & Sons Ltd, Baffins Lane, Chichester West Sussex, PO19 1UD, England, 2003. ISBN: 0470856688.

[34] *RoboVM*. URL: `http://docs.robovm.com/index.html` (visited on 07/20/2016).

[35] *STM32F407VG*. URL: `http://www.st.com/content/st%7B%5C_%7Dcom/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f4-series/stm32f407-417/stm32f407vg.html` (visited on 08/27/2016).

[36] G C C Team. *The GNU Compiler for the Java Programming Language*. 2004. URL: `http://gcc.gnu.org/java/` (visited on 07/14/2016).

[37] *The LLVM Compiler Infrastructure*. URL: `http://llvm.org/` (visited on 08/04/2016).

[38] *Turboj home page*. URL: `http://www.opengroup.org/openitsol/turboj.` (visited on 07/13/2016).

[39] Ankush Varma and Shuvra S. Bhattacharyya. "Java-through-C compilation: An enabling technology for Java in embedded systems". In: *Proceedings -Design, Automation and Test in Europe* 3 (2004), pp. 161–166. ISSN: 15301591. DOI: `10.1109/DATE.2004.1269224`.

[40] *w3c -XSL*. URL: `https://www.w3.org/Style/XSL/` (visited on 10/22/2016).

[41] *w3schools.com - XML Tutorial*. URL: `http://www.w3schools.com/xml/` (visited on 10/22/2016).

[42] Chih-sheng Wang et al. "A Method-Based Ahead-of-Time Compiler for Android Applications". In: (2011), pp. 15–24.

[43] Michael Weiss et al. "TurboJ a Java Bytecode-to-Native Compiler". In: *LCTES '98 Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (1998), pp. 119–130.

[44] Mark Wielaard. *A look at GCJ 4.1*. 2006. URL: `http://lwn.net/Articles/171139/` (visited on 07/18/2016).

[45] *XMLVM Documentation*. URL: `http://xmlvm.org/documentation/` (visited on 07/20/2016).

[46] "XMLVM User Manual". In: (2009).