Master's Thesis

Performance Estimation for Smartcard Applications ported onto Java Cards

Harald Schlatte-Schatte

Institute for Technical Informatics
Graz University of Technology

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

Advisor: Ass. Prof. Dipl.-Ing. Dr. techn. Christian Steger
Advisor: Dipl.-Ing. Johannes Loinig

Graz, March 2012

Kurzfassung

Applikationen auf Smartcards sind in der heutigen modernen Gesellschaft allgegenwärtig. Die bekanntesten Anwendungsgebiete sind neben Banking und Zutrittskontrolle auch öffentlicher Verkehr und mehr denn je mobile Geräte. Solche Applikationen können auf verschiedene Arten auf Smartcards implementiert werden. Einerseits existiert die konventionelle Smartcard, auf der in Programmiersprachen wie C und Assembler entwickelt wird. Andererseits gibt es Java Cards, die auf dynamisch nachladbare Applets setzen, welche in Java implementiert werden. Java Cards bieten Vorteile hinsichtlich Entwicklungszeit und Wartbarkeit und weisen eine große Interopabilität durch Einsatz der Programmiersprache Java auf. Da die Ausführung von Java Programmen jedoch einen Zwischenschritt beinhaltet, die so genannte virtuelle Maschine, verringert sich die Performance der Applikation um genau diesen Overhead. Sollen nun also nativ implementierte Applikationen auf eine Java Card portiert werden, muss mit einem Performanceverlust gerechnet werden.
Diese Diplomarbeit befasst sich mit der Abschätzung des Performanceverlusts bei der Portierung von nativen Smartcardapplikationen auf eine Java Card. Die Ergebnisse sollen als Entscheidungshilfe dienen, ob es Sinn macht, Projekte auf Java Cards durchzuführen.

Abstract

Applications for smartcards are an important part of the modern information culture. Well known application areas include not only payment processing and access control, but also public transportation systems and increasingly mobile communication. More specifically, payment with mobile devices is becoming a really important and interesting area of research. Applications can be implemented in different kinds of programming languages. There are either smartcard applications, which are implemented in programming languages, such as C or assembler, or there are Java Cards, which allow implementing applications in the higher Java language. Java Cards also offer the functionality to load applets dynamically. However, the execution of such applications contains an intermediate step: the execution of byte codes on the Java virtual machine. This step causes an overhead, which has impact on the performance of the application. Thus, if native implemented applications are ported to Java Cards, performance is likely to deteriorate. This diploma thesis deals, in detail, with estimating performance loss while porting native applications onto Java Cards. The results are meant to serve as a basis for deciding whether it's worth it to port specific projects onto Java Cards.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ...
(Signature)

Acknowledgement

This master's thesis was created at the Institute for Technical Informatics, Graz University of Technology in cooperation with NXP Semiconductors Austria GmbH in Gratkorn.

First of all, I would like to thank my advisor Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger for the professional and organizational support during the creation of this master's thesis. Thanks to him, the Institute for Technical Informatics and their good connections to business partners this thesis was made possible.

I owe my deepest gratitude to my advisor Dipl.-Ing. Johannes Loinig, who supported me in a great way in all details concerning this master's thesis. His amazing help concerning technical details about the practical work, but also during the creation of the theoretical part was contributing a lot to the success of this work.

Further, I want to thank NXP Semiconductors in Gratkorn for the opportunity to do the masters's thesis in their company. I am grateful I could do this thesis in such a professional environment with excellent facilities and great colleagues. This thesis would not have been possible without support from my team members, who never hesitated to help me with any kind of upcoming questions.

Last but not least, I want to thank my family for the great support not just during the work for my diploma thesis, but for enabling my studies in general. Also, I would like to thank my girlfriend for her support in these sometimes hard and stressful times.

Graz, March 2012 Harald Schlatte-Schatte

Contents

1 Introduction **11**
- 1.1 Motivation . . . 12
- 1.2 Outline . . . 13

2 General Overview **14**
- 2.1 Card Types . . . 14
  - 2.1.1 Embossed Cards . . . 15
  - 2.1.2 Magnetic Stripe Cards . . . 16
  - 2.1.3 Smartcards . . . 16
- 2.2 Smartcard software architecture . . . 17
- 2.3 Java Cards . . . 18
  - 2.3.1 The Java Virtual Machine on a Java Card . . . 20
  - 2.3.2 Applets . . . 22
  - 2.3.3 Java Card communication principles . . . 23

3 Related Work **25**
- 3.1 „Mesure“ tool to benchmark Java Cards . . . 25
  - 3.1.1 Results . . . 28
- 3.2 Benchmarking Java Cards . . . 29
- 3.3 Analyzing Control Flow in Java Bytecode . . . 30
- 3.4 Comparison of Java Cards and native smartcards . . . 32
- 3.5 Embedded software execution time estimation at different abstraction levels 34
  - 3.5.1 Techniques . . . 34
  - 3.5.2 Basic Block Approach . . . 35
  - 3.5.3 Source Code Analysis . . . 36
  - 3.5.4 Results . . . 37
- 3.6 A SW performance estimation framework for early System-Level-Design using finegrained instrumentation . . . 38

4 Design and Concept of the Estimation Method **40**
- 4.1 Problem definition . . . 40
- 4.2 Overview . . . 42
  - 4.2.1 Design for the application block model . . . 43
  - 4.2.2 Modules design . . . 43
- 4.3 Design of setup of estimation rules . . . 44
  - 4.3.1 Logical view of estimation rules . . . 45

4.3.2 Tools view for creating estimation rules 45
4.3.3 Process view for creating estimation rules 47
4.4 Design of performance estimation 48
4.4.1 Tools for estimation of an application 49
4.4.2 Logical view for estimation of a native application 50
4.4.3 Process view for estimating a native application 50
4.5 Approaches to get accurate timing information 51
4.5.1 Using the PC clock 52
4.5.2 Using an oscilloscope 53
4.5.3 Using a hardware spy 53
4.5.4 Using a simulation on a hardware model 54
4.5.5 Used method in this work 54

5 Implementation of the Estimation Method **55**
5.1 Block representation 55
5.2 Setup of estimation rules 57
5.2.1 Analyzing sample application performance 58
5.2.2 Sample application block model 60
5.2.3 Creating sample applications 61
5.2.4 Deducing performance for single blocks 64
5.2.5 Deducing TimedBlockDB 65
5.3 Estimation of native applications 67
5.3.1 Parsing of application block model 69
5.3.2 Parsing TimedBlockDB 70
5.3.3 Combining application block model and TimedBlockDB 70
5.4 Extending the estimation process with new GenericBlocks 71

6 Results **72**
6.1 Setup of estimation rules (TimedBlockDB) 72
6.1.1 Identifying blocks 72
6.1.2 Sample applications 75
6.1.3 Joining sample applications and deducing TimedBlockDB 78
6.2 Estimation of a security module 82
6.3 Summary 83

7 Conclusion and Future Work **86**
7.1 Future Work 87

A Glossaries **89**
A.1 Acronyms 89
A.2 Symbols 91

Bibliography **92**

List of Figures

2.1 Smartcard types 15
2.2 Java Card layout 19
2.3 Different kinds to compile and execute a Java program [RE08] 21
2.4 Oncard and offcard part of the JVM [Sch04] 22
2.5 Typical applet development process [RE08] 22
2.6 Different cases of APDU transactions [Che00] 24

3.1 Architecture of Mesure benchmark environment [PCB09] 26
3.2 Distribution of timing measurement of the *sadd* bytecode [PCB09] 29
3.3 Impact of different measurement systems on timing [Erd04] 30
3.4 Basic block characterization and estimation [GASE] 36
3.5 Timing estimation based on source code analysis [GASE] 37
3.6 Software estimation techniques [KKW^{+}06] 39

4.1 Overview of problem definition 43
4.2 Design of an application block model 44
4.3 Performance estimation module overview 45
4.4 Logical view for creation of TimedBlockDB 46
4.5 Tools involved in creating TimedBlockDB 46
4.6 Process of getting performance information for unknown blocks 48
4.7 Tools involved in estimating an application 49
4.8 Logical view for estimation of a program 50
4.9 Process view for estimating programs 51
4.10 Time consumed on different levels of smartcard communication 52
4.11 Measuring processing time with the oscilloscope 53

5.1 System model for getting instruction accurate performance information . . 58
5.2 Steps for analyzing performance of a sample application 59
5.3 Identifying blocks of the source file 61
5.4 Deducing TimedBlockDB 66
5.5 Communication between C# and Matlab 67
5.6 Operational flow for estimating an application 68
5.7 Detailed view of the single steps of the Performance Estimator 68
5.8 General view of interpreter pattern 69
5.9 Class diagram of the Parser 69
5.10 Class diagram of TimedBlockDB 70

6.1 Application to calculate estimation rules 79
6.2 Overview, how operations are distributed on above mentioned operation classes in different test cases 84

List of Tables

2.1 Structure of a command-APDU . . . 23
2.2 Structure of a response-APDU . . . 23

3.1 Speedup and error in different test cases compared to ISS based results . . 37

4.1 Native source code, which can not easily be mapped to Java . . . 42

5.1 Examples for different generic blocks . . . 56
5.2 APDU for measuring overhead, first two bytes describe the loopcount . . . 64

6.1 A subset of possible blocks used for creation of TimedBlockDB . . . 73
6.2 Advanced Java Card specific composite blocks, which are implemented on a low level. Those are analyzed as a whole and are not split up in their low level components . . . 74
6.3 Result of block analysis: blocks and their timing in cycles . . . 81
6.4 Results for estimation of a security module . . . 83
6.5 Results for estimation of a security module, cryptographic operations are discarded . . . 84

Listings

3.1 Example for evaluating the *smul* bytecode 27
3.2 Bytecode sequence for above multiplication 27
3.3 Java sourcecode and corresponding Java bytecode 31
5.1 DTD for block representation 56
5.2 Sample source code to set and reset the SFR 59
5.3 Sample timing log file for executing an addition on a Java Card 60
5.4 Sample application block model for the application stated in Listing 5.2 . . 60
5.5 Application to analyze the performance of sample applications 62
5.6 Sample application with main execution loop 63
6.1 Sample application to measure overhead 75
6.2 Sample application block model to measure overhead 75
6.3 Sample application to analyze If and Assignment 76
6.4 Sample application block model to analyze If and Assignment 76
6.5 Sample application block model to analyze an Assignment block 77
6.6 Sample application block model to analyze Add 77
6.7 Sample application block model to analyze Decrement 78

Chapter 1

Introduction

In recent years smartcards have been used more and more in a wide variety of applications. As mentioned in [PCB07] payment processing has encountered a huge growth in popularity. However, other areas like access control systems have also made the smartcard really popular. With the boom in mobile communication smartcards have acquired another important application domain. Used as SIM cards, smartcards serve as an identification module within communication networks. Furthermore, the significance of mobile payment processing has been vastly expanding in turn becoming another application for smartcards. With increasing potential usage, especially for high security applications such as payment processing, access control, and similar applications, it is important to provide a sufficient amount of security features on a smartcard. Therefore, it is essential for smartcards to become more and more intelligent.

At the beginning smartcards only contained memory and simple logic to control access to that memory. In the meantime modern smartcards have improved to contain several parts of a personal computer. There are obviously no input/output systems contained in smartcards, since cards can only communicate with card terminals, but concerning the internal structures everything else is in place. The main parts of a smartcard include: a smartcard processor, memory, a special purpose co-processor (to ensure a high level of security), and buses (in order to facilitate communication between various smartcard components). Smartcards and reader devices communicate either in contact or contact-less manner. Smartcards requiring contact communication are frequently used in banking as well as in mobile communication devices. Smartcards, which communicate in a contact-less manner, on the other hand, have many applications for access control and identification purposes, especially in outdoor environments where external factors, such as weather can influence their performance.

As mentioned before, security is probably the most important aspect of smartcard design. Since, smartcards are often used for finance purposes, there will always be a threat of somebody wanting to crack the system. Thus, a lot of effort is put into developing countermeasures against possible attacks. Unfortunately, as a smartcard developer one can never be sure that these security features suffice. Consequently, maybe it is necessary to be able to alter an application after having released it. It is advantageous for a smartcard to allow loading application code dynamically, that is, manufacturing the smartcard with

a fixed operating system and allowing future/later application upload by the customer.

As a matter of fact, smartcard applications can be developed using various technologies. For example, there are native applications, which are written in languages like C or assembler. These applications require time-consuming development cycles. Since native code can become quite complex, it is possible that it will soon become too difficult to maintain, especially if code quality is low or the original developer is not available any more. Applications can also be developed using high-level languages like Java. In that case, a smartcard must provide an execution platform for that specific programming language. One example of such smartcards are Java Cards. Those cards provide a Java Virtual Machine, which executes previously generated Java applications. The great benefit of this approach is the interoperability of Java applications. Specifically, no matter what hardware is used, the applications always remain the same. In comparison to native code, Java code is easier to develop and to maintain. However, because Java applications are executed on a Java Virtual Machine, which consumes lots of resources, they lack in terms of performance.

This paper describes the performance of both types of smartcards, Java Cards and native platform smartcards. It further explores a method of estimating application performance (i.e. speed of execution of an application on a smartcard), when a native application is re-implemented (i.e. ported) onto a Java Card. Using this method, there is no need to actually re-implement the native application; rather a specification or a source code is sufficient in order to estimate the impact of re-implementation on application performance.

1.1 Motivation

As mentioned before, Java Cards offer the opportunity to run easy and fast to develop Java applications (so called applets) on a smartcard. This is managed by a strict separation between operating system and applications. During the manufacturing process the operating system is located in the read only memory (ROM) of the card and cannot be changed any longer. Other than that, customer applications are put on the card later on and are stored in electrically erasable programmable read only memory (EEPROM). The card operating system provides a Java virtual machine (JVM), on which customer programs are executed. If, in case of application adaptions or security issues, the application code has to be changed, just the modified applet has to be updated on the Java Card. Further, the higher programming language Java allows pretty fast development cycles. Also, Java Card applets enable a high grade of interopability, which means the applet can be executed on every Java Card, no matter which manufacturer or how the underlying hardware is designed. However, the penalty for these capabilities can be found in the performance of Java Card applications. Since there is a virtual machine, which also consumes processing time and memory resources, the performance is going down.

Today a lot of applications for smartcards are implemented in native programming languages. There are some considerations to port these application to a Java Card to benefit from the easy and fast development and maintainability respectively the interopability.

Before doing that, one wants to know how big the performance impact will be. Therefore a method for estimating the performance loss in case of porting this application to a Java Card shall be developed. Based on this results a decision can be taken if this performance loss is acceptable for more flexibility and a faster development process.

1.2 Outline

This master's thesis is divided in following chapters.

Chapter 2 gives general information about smartcards, which kinds are available and their different specifications. Furthermore, the fundamental features of a Java Card are mentioned, their basic properties and specialties which make a Java Card so unique are described.

Chapter 3 deals with papers which concern similar topics as this thesis does. Some of them served as general impressions which were taken in a refined form to solve this master's thesis' task.

Chapter 4 covers the research part of the diploma thesis. Here a method for an accurate estimation of the performance impact shall be found. Before one can do that, it is necessary to find a way for exact timing measurements on smartcards. The system is divided into two parts there; one for setting up estimation rules and another for the actual estimation. The estimation system itself is presented from different points of view, to get a good idea which components are existing and how these are interacting together.

Chapter 5 considers the implementation details of the system. It describes how the designed components are implemented and which development environment has been used.

In Chapter 6 the results from the previous chapter are discussed and verified. Therefore an existing native version of an application was taken and implemented on a Java Card so that these two versions are behaving equally. With a sample transaction, the timing behavior of both cards is measured. These results are compared with the estimation.

The last chapter gives a final summary and a discussion about the found results. Also possible improvement steps and weaknesses in the existing solution are discussed.

Chapter 2

General Overview

General information about smartcards has been taken from [RE08]. First smartcards were already used in the early 1950s, but these were rather just *cards*, no point of calling them „smart“. They were used for payment, however this kind of money transaction was reserved for the upper class. These cards had a few security features and were quite easy to fake. With the increasing abuse of cashless money transactions, some security features had to be implemented. The first attempt were magnetic stripes which allowed the storage of data on the card. Since a magnetic stripe is easy to read, delete and rewrite, it cannot be used as a strong security feature. Therefore an additional personal identification number (PIN) was implemented. These classical bank cards are still widely spread in financial transactions.

The development of microelectronics opened a lot of new opportunities in the smartcard area. This evolving technology enabled the first microprocessor card where memory could not be accessed in an uncontrolled way any more. Furthermore, the implementation of modern cryptographic algorithms on smartcards provided a quite high grade of security.

Another important step in smartcard history was the completion of the EMV specification as stated in [EMV11], which contained general description of cards and card terminals. Also the electronic fund transfer in whole Europe was unified. EMV stands for the three companies Europay International (in the meanwhile Mastercard Europe), Mastercard and Visa.

But smartcards are not only present in payment applications. In the meanwhile they are used in a lot of applications in daily life for example mobile phones, public transport, access control systems respectively generally in secured application areas like passports.

2.1 Card Types

Cards can be distinguished either because of their format or their offered features, as discussed in [RE08]. In the smartcard area, the ID-1 format has succeeded which can be found in all Credit-, ATM- and other cards for fund transfers. Since features are developed in an evolving process, most cards provide a combination out of more features. This

is necessary to ensure an appropriate backwards compatibility. Additionally, every new feature increases security on smartcards.

On the other hand, cards can be classified as memory cards and microprocessor cards. Memory cards have the task to store a value and allow terminal machines to change or in most cases to decrease this value. Telephone cards with magnetic stripes could provide that functionality, but it was rather easy to reset them to the initial value. To counterfeit this problem, memory cards are provided with a security logic, which ensures that each memory cell can just be written once. This kind of card is used for all kinds of transactions where services or goods are paid without the use of cash. Popular examples are public transport, vending machines, car parking machines and so on. Microprocessor cards are able to store private keys and make use of modern cryptographic algorithms. What a microprocessor card actually does, is freely decided by the smartcard developer, the only restrictions are memory and processing capacity. Different applications depending on the capacity of memory and processing are shown in Figure 2.1. Information about different card types are stated in [RE08].

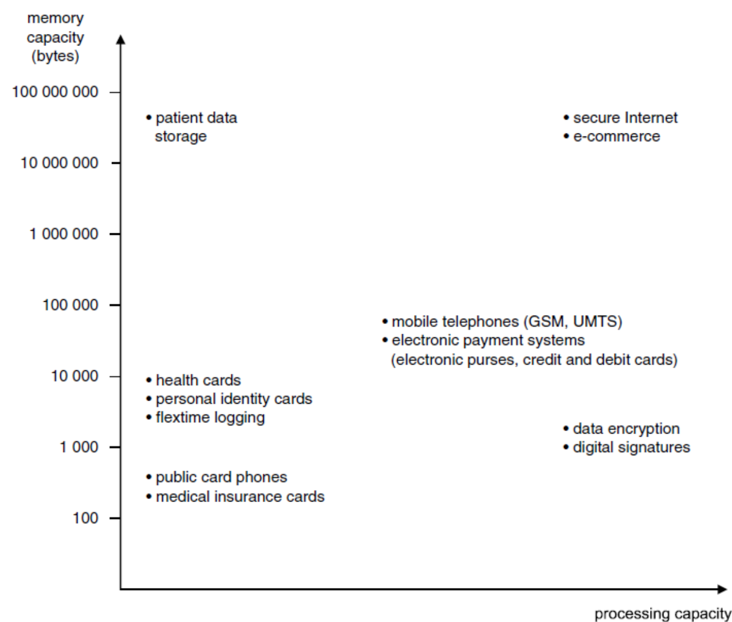

Figure 2.1: Typical smartcard application areas [RE08]

2.1.1 Embossed Cards

This is the easiest form of an identification card, everyone can read out the stored information. Although such a card is easy to read, modifications can't be done without special equipment. However, if modifications are done, they can be discovered quite easily by a skilled person. Additionally, a great advantage is that fund transfers can be performed with simple and inexpensive equipment. That's why embossed cards became really popular all over the world. However, one drawback is that this kind of finance transaction

produces a large amount of paper. To make this process easier, cards needed to become machine-readable.

2.1.2 Magnetic Stripe Cards

The requirement, to be readable by machines is totally fulfilled by magnetic stripe cards. Now information can be read out and manipulated by terminal machines. However, a big disadvantage is that everyone can modify data with a read/write device. Additionally, which makes this a really bad drawback, it's hard to detect changes which have been done abusive.

2.1.3 Smartcards

With the integration of a chip on identification cards, the smartcard as we know it today, was born. Combined with other components, security features like safe memory access and encryption algorithms can be implemented. Communication can be done either in contact mode or within an electromagnetic field, which is known as contactless mode. Moreover the available memory is way larger than the one on magnetic stripe cards. Probably the biggest advantage is the ability, to restrict access to memory. This makes it possible to store security relevant data on the card, like private encryption keys, account numbers and so on. That is also a reason why the smartcard became the most popular security module, it's portable and it is able to store private data in a secure way. As mentioned before, smartcards can be distinguished in two types:

Memory Cards

These cards contain memory elements (mostly EEPROMs) and a simple logic which controls the access to the memory. In the simplest case it is just a write protection for the memory. Advanced memory cards have a more complex logic which allows writing card data only after a correct PIN input via the terminal.

Microprocessor Cards

These cards contain a dedicated microprocessor, which allows the implementation of complex security features on a smartcard. Further, there are numeric co-processors for a fast calculation of cryptographic operations included. Such a kind of smartcards have a widespread application area, because the final purpose is totally determined by the program that is running on the smartcard. It is also possible, to make applications dynamically loadable by the card issuer, so only the operating system is integrated in the card. Of course, such smartcards need a huge amount of security features, to prevent the card holder from accessing or in worst case changing the operating system code. So the operating system has to make sure, that a user program can only access its own memory.

Today, most of these cards are able to load user content dynamically. However, for security reasons, this feature is kept secret intentionally by the card manufacturers. They try to detain any abusive use of that mechanism as good as possible.

Memory Architecture

Commonly, smartcards contain three kinds of memory. Non volatile non writable memory, non volatile writable memory and volatile writable memory. These types are also known as:

- ROM (Read Only Memory) is only readable and non writable memory. Typically most parts of the operating system and other never changing data are stored here. The ROM layout is fully defined at card's manufacturing process and can not be changed afterwards.
- EEPROM (Electrically Erasable Programmable Read Only Memory) keeps its content also after a loss of power, so it can be used for keeping persistent data which is changing during a card's lifecyle. It is writable by applications but one has to keep in mind, that EEPROM access is fairly slow. Compared to write operations in RAM, EEPROM needs about 1000 times longer. So EEPROM write operations should be used rarely in time-critical applications. Reading EEPROM takes about the same time as reading RAM.
- RAM (Random Access Memory) is volatile memory which is used for keeping data during a session. When power is gone, values in RAM are lost. Concerning the amount of memory, RAM has the smallest size of all memories on a smartcard. Accessing RAM is quite fast, so all performance critical data should be stored in RAM.

2.2 Smartcard software architecture

The development of smartcard software architecture is analyzed here according to information from [DDJ03]. At the time when smartcards were invented, the entire software on a smartcard was a single monolithic block and was burned on the card by the semiconductor manufacturer. That means there was no difference between operating system and user code, it was just one big module that managed everything from hardware access until application behavior. Companies quickly noticed, that smartcard software consists of more and more similar parts which were simply copied during the development process for another application. So software was divided into following layers:

- Hardware management modules to access hardware
- Application level modules, which are reusable modules that are needed in common applications like PIN code management etc.
- Application specific source code to provide features which are not covered by previously defined application modules

These *second generation* smartcards benefit of reusable application modules but still have the drawback, that every feature is burned on the card. So card behavior could not be changed after the card manufacturing process.

In the *third generation*, the application's development process moved from the semiconductor manufacturer to smartcard manufacturers and further to smartcard issuers and service providers. The semiconductor manufacturer was just responsible for developing a standard platform while the smartcard manufacturer added some special components called filters. These filters contained generic modules which were not covered by the standard platform. Smartcard issuers finally add specific data to the card before they give it to the customer (card holder).

In the *fourth generation* service providers wanted to keep the time-to-market as short as possible. The earlier they bring a product on the market, the more customers can be acquired, so a solution for fast smartcard development shall be implemented. They want to avoid to create a new card every time they change something in their software behavior. To meet these new requirements, the smartcard software design needed to be totally revised. A way to change user applications after card issuance (so called *post issuance*) became necessary. So smartcards are just an execution platform for later loaded applications. They provide a framework of many application programming interfaces (APIs) which contain all the needed features for smartcard development. Some of those smartcards use a different programming language and an according virtual machine for their user application. This provides a unified execution platform and user applications are totally independent of the operating system. This makes applications portable and also serves as a security feature since intermediate codes usually can be checked better for security threats.

Some facts could be seen during evolving smartcard software architecture. While in the late eighties, the development process was entirely done by the semiconductor manufacturer, in the meanwhile the operating system and application are separated more and more. So application development can be done widely independent of the operating system. This offered new opportunities for smartcard providers because they could manage the software lifecycle theirselves. On the other hand, companies could focus just on operating system development which opened up new business segments.

Although this separation is great concerning flexibility and modularity, with the increasing number of abstraction layers, the performance suffers from that. Hence, operating systems still must be extensible to the target applications needs.

2.3 Java Cards

One example for such a smartcard are Java Cards. At its manufacturing time, the card primarily consists of its operating system and the Java Virtual Machine (JVM, see specification in [LY99]). Since interopability is a great goal in the Java Card terminology, applications designed for Java Cards can be run on every Java Card, no matter which hardware it is based on. Since most of its applications are somehow money related, security is also an important feature. It ensures that no unauthorized access neither to the operating system nor to the user application is possible. A further feature is, that more user applications can exist on a card whereas mutual influence needs to be prevented. In the Java virtual machine, every application has its own sandbox and cannot access other

memory than in its sandbox.

Logical layers on a Java Card are shown in Figure 2.2. Smartcard hardware is located at the lowest level, it consists of the smartcard processor including its co processors for example for cryptographic operations. The next layer can be seen as the hardware abstraction layer, it controls how to access hardware and offers interfaces to native functions. On the next layer, the Java virtual machine is located. It is executing bytecodes on the lower lying layers. Those two layers can be seen as the operating system. The next layer is already independent of the hardware. The Java Card class library contains general descriptions, how an applet for a Java Card has to look like and offers general methods. On top, the applet layer is located. The actual user applications are located here. Applications are loaded on a Java Card during the personalization process. This allows a great flexibility concerning the possible fields of operation. The card issuer has full control about the offered functionality of this applet. Further, software updates can be performed quite easily, since new features can be added to the applet in a simple way. The use of the programming language Java makes the development process easier as well.

A Java environment on a PC needs a lot of memory and processing resources. Since smartcards have a quite restricted amount of memory and also not so powerful processors, some restrictions in the Java Card environment are necessary:

- no garbage collector
- no threads
- no dynamic class loading
- data types long, float, double and char are not accessible

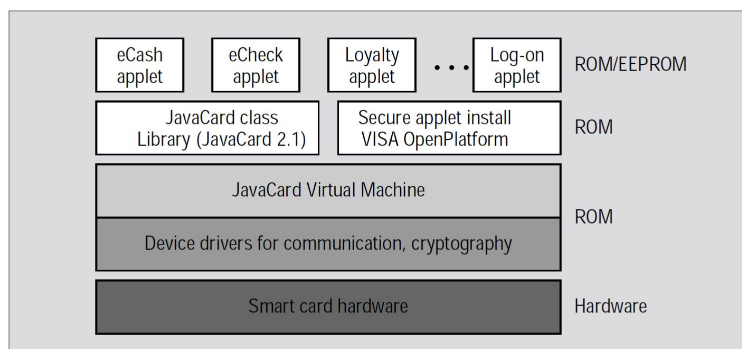

Figure 2.2: The logical layers on a Java Card [BBE+99]

The Java Card technology fulfills following specifications [LY99]:

- Java Card Runtime Environment (JCRE): specifies which behavior and functionality should be provided by implementations of the Java Card technology.

- Java Card Virtual Machine Specification: defines the instruction subset of the Java Card virtual machine. Also file formats for installing applets and libraries on a Java Card are specified.
- APIs for the Java Card Platform describe the application programming interface of the Java Card technology. It also contains the class definitions for virtual machine and runtime environment.

Java applications are translated into Java bytecode by the compiler. This is kind of a hardware independent object code and can be interpreted as machine code for a Java processor. The Java processor itself is not existing physically, it is an interpreter which is simulated by a real processor. The Java Virtual Machine (JVM) is such a simulation of a Java processor according to its specification in [LY99]. That means, the JVM is the only interface to the hardware. So if one wants to execute Java on a new platform, only the Java Virtual Machine has to be ported to that new target platform. Additionally the JVM ensures security when accessing objects and other data structures. In fact, every program gets its own *sandbox*, which is an isolated memory area just for this dedicated program. It is not allowed to access memory areas of any other application. With the big advantage of hardware independence however, the execution speed is decreasing significantly. The development of a JIT-compiler was the first step to handle this problem. Here Java bytecode is translated directly into machine code during the first execution. This needs some more time during the first execution, but benefits in all coming executions. Such a JIT-compiler is a rather complex construction, it needs a lot of memory and processing resources. Since these resources are rarely available on a smartcard, a JIT-compiler is also not available on Java Cards. Nevertheless, there is an approach for a JIT compiler from [Sha02]. It shows a method for a JIT implementation on an ARM processor which manages to work with these restricted resources. Another approach to improve the performance is the direct compilation in machine code. But the non uniform architectures of Java Cards make this step quite difficult, as well. An overview of the different kinds of compiling and executing a Java program is shown in Figure 2.3.

Nevertheless, there are still opportunities to improve performance on Java Cards. One of it is to implement performance critical functions on a very low level. This needs a high level of planning, to design the functions generic to cover a wide area of applications. The usage of such native functions results in a restriction of flexibility, since these functions have to be implemented for every platform to make sure the APIs are unique. See also in [Ora11].

2.3.1 The Java Virtual Machine on a Java Card

The Java Virtual Machine is a simulation of the Java processor to execute Java bytecode. It contains all necessary components of a processor like instructions, register, program counter and accumulator. Architecture, command set and data types are clearly defined by the specification [LY99]. Extensions to the Java language do not necessarily require a changing of the JVM since the new Java commands just need to be modeled with existing bytecodes which is a task for the compiler. The virtual machine is executing those bytecodes, no matter out of which language they had been generated.

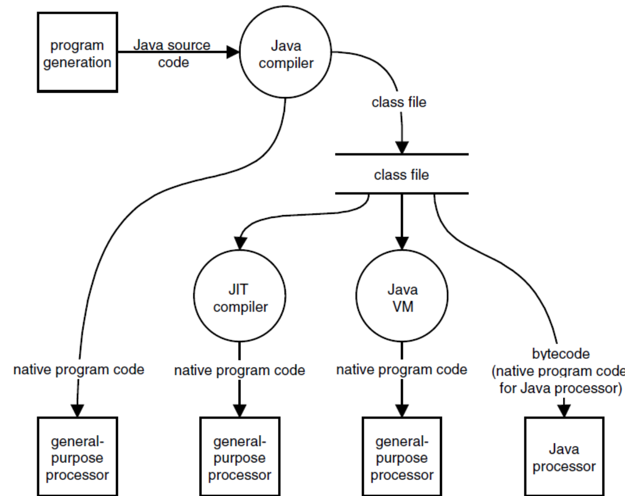

Figure 2.3: Different kinds to compile and execute a Java program [RE08]

Since the Java Virtual Machine is modeled as a stack architecture, commands do not have any source or destination register. So all necessary operands for the operation are fetched from the stack and the results are pushed back on the stack. However, most processors nowadays are designed as register architecture, so a mapping between commands on register level and stack commands has to be done. Among others, the virtual machine contains the classloader which loads all required classes. During that, it ensures that access policies for classes, methods and variables are redeemed to secure access among the involved classes.

Further, in another pre-execution step, the byte code is checked by the Byte Code Verifier for typical programmer errors. It prevents wrong casts, array index overflows and ensures data to be always initialized and type-safe. Additionally, it ensures that the stack size moves within an allowed range.

To execute this virtual machine on a Java Card, some reductions were necessary. Among others, the number of instructions was reduced from 150 to nearly 80. This made the virtual machine itself significantly smaller.

Additionally, the virtual machine is split into two parts, on-card and off-card as shown in Figure 2.4. All static operations and calculations are performed already at the time of development, so these operations do not generate workload on the cards processor. The on-card part is responsible for executing programs on a Java Card, where data is changing every session.

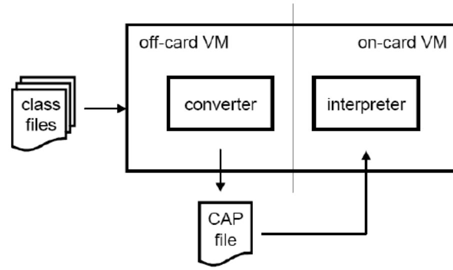

Figure 2.4: Oncard and offcard part of the JVM [Sch04]

2.3.2 Applets

Applications for Java Cards are implemented as so called applets. Those can be loaded to a card dynamically through a loading mechanisms and then be executed by the virtual machine. The transfer of such an applet can only be done in form of so called cap-files. These files are defined in the Java Card standard and contain a converted and shortened version of the class file. Here linking information like names of fields, methods and classes are deleted and some optimizations are applied. After that, referenced classes are bound. Uploading means just placing the cap-file in the non-volatile memory. After that, an installation step is required to tell the Java Card about the new piece of software.

On the Java Card, the cardmanager (which is also a special kind of applet itself) takes care about those uploading and installation steps. Further, the cardmanager also offers functionality to select, deselect and delete an applet. It also gives control to the applet in case it gets selected. After selection of an applet, all received commands are forwarded directly to the applet and processed there. Another possibility is to place applets in the ROM. In that case, applets can be accessed already at the very beginning of the applet lifecycle.

A typical applet development flow is shown in Figure 2.5.

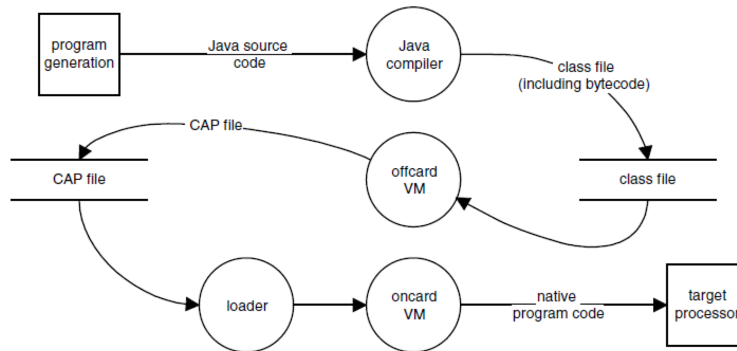

Figure 2.5: Typical applet development process [RE08]

Mandatory Header				Optional Body		
CLA	INS	P1	P2	Lc	Data field	Le

Table 2.1: Structure of a command-APDU

Optional Body	Mandatory Trailer	
Data field	SW1	SW2

Table 2.2: Structure of a response-APDU

2.3.3 Java Card communication principles

Communication between card and terminal is done in a half-duplexed way, that means data can either be sent or received at the same time. Card and terminal are communicating via packets formed according to a protocol, in this case the packages are called application protocol data units (APDUs). APDUs can be used to send a command or a response.

Communication is always initiated by the terminal, so a Java Card plays the slave-role in a card-terminal connection. The card waits for a command, executes it and sends back the response. The APDU protocol is specified in ISO/IEC 7816-4 [Car11] and it distinguishes between command APDUs (C-APDUs) and response APDUs (R-APDUs). Tables 2.1 and 2.2 show the structure of these two packages.

In a C-APDU *CLA* stands for class byte, which indicates the type of command, for example proprietary or interindustry. *INS* contains the specific instruction, for example „read data“. *P1* and *P2* are parameters for the given command and can be chosen freely. Optionally one can send data of variable length with the command. This length is specified in *Lc*, which is followed by the data itself.

If the command expects data to be returned, the length of this data must be specified in *Le*. There can also be the case that no data is sent, but data is expected to be returned. In that case, *data* and *Lc* are not existing in the APDU. A response APDU consists optionally of the data which is returned and it mandatory contains two status bytes, which tell the caller whether the processing was successful. For example, „90 00“ stands for a successful execution.

Since both APDU types contain optional parts, there are four different cases of APDUs which are shown in Figure 2.6 [Che00].

- In the first case no data is sent to the terminal, and no data is expected to be returned. So the C-APDU just consists out of the header and the R-APDU contains just the trailer.
- The second case sends no data to the terminal, but expects data (specified in *Le*) to

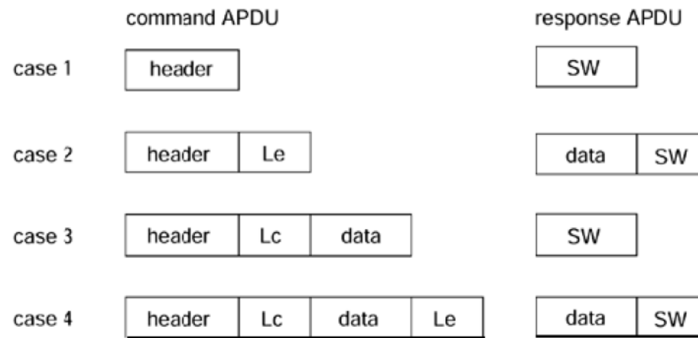

Figure 2.6: Different cases of APDU transactions [Che00]

be returned. The R-APDU contains both, data and trailer.

- Case 3 shows that data is sent to the terminal, so the data field and the *Lc* field exist in the C-APDU. Since *Le* is not existing, no data is expected and the R-APDU just contains the mandatory trailer consisting of the two status words.
- In the fourth case the terminal sends data and expects to get data back from the card. So all optional fields are filled in the APDUs.

Chapter 3

Related Work

This chapter contains a brief description of some literature which covers similar topics as mentioned in this work. Additionally some impressions from these papers have been taken and refined to fit to our problem. First, some approaches for benchmarking Java Cards are presented. There are also some ideas, how to measure the performance respectively execution time on a Java Card. Furthermore, Java Card bytecodes are concerned. Since bytecodes are the smallest unit on the Java layer there is an approach to use them for performance estimations.

3.1 „Mesure“ tool to benchmark Java Cards

This work is done under the MESURE project which was funded by the French administration (Agence Nationale de Recherche) and wanted to find new possibilities to benchmark the performance of Java Card platforms. Information for that topic is stated according to [PCB09] and [PCB07]. They are emphasizing only on the usage phase of a Java Card, not installation or personalization. Performance benchmarking is done at three levels:

- VM level: evaluate the execution time of basic virtual machine instructions.
- API level: evaluate methods which are provided by the API, their implementation itself is done in lower layers to increase performance.
- JCRE (Java Card Runtime Execution) level: for measuring the „overhead“ of applications like function calls, etc.

This paper gives interesting input, how performance measurement can be done for specific parts of applets, and which modules are necessary to achieve good results. Further, the results are interpreted concerning their statistical distribution.

The performance benchmark does not take care about reader or protocol dependent information since every reader behaves differently and one can never be sure how much overhead the communication between reader and PC produces. Another very important condition is, that the tests should be available for all cards on every reader. To be consistent, all these overhead and noise content needs to be cleared out. This can be done by calculating an average over all measurements. Statistical calculations were used to find

the trustworthy results of measurements.

Further, a repeated execution of test items is necessary since their execution time is far beyond 1 second, quite short comparing to protocol and other overhead. So it would be quite difficult to filter this information out of the whole signal. With several execution times, the part of the needed information in the whole timing profile increases. But not just every test item is executed several times, also the operation to measure is executed very often. This can be adjusted by the P2 value of the APDU which is sent to the on-card application.

The benchmarking environment itself consists out of some functional modules. It starts with configuring and calibrating the benchmark, continues with execution and finally filtering and post processing of the obtained results. An overview of the architecture is shown in Figure 3.1.

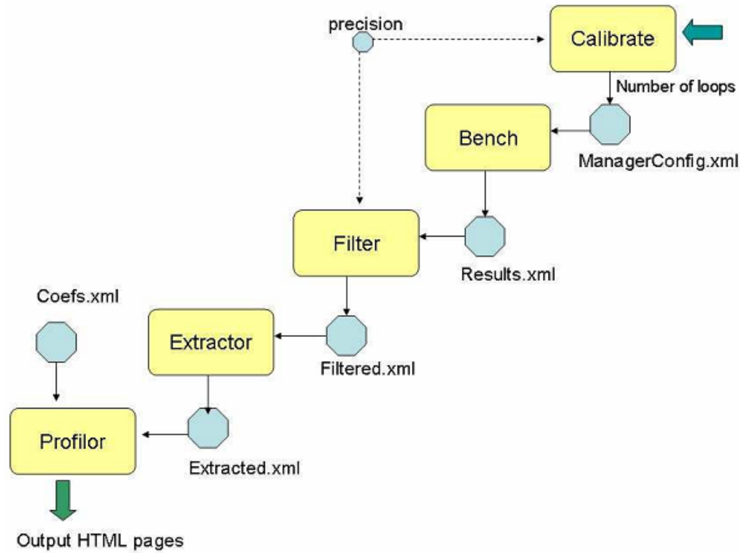

Figure 3.1: Architecture of Mesure benchmark environment [PCB09]

The calibrate module makes sure that the right number of executions for the desired operation is taken. This is done by evaluating results for different numbers of executions. Mean value and standard deviation are analyzed and the configuration with the smallest standard deviation is used for further tests.

The bench module is executing the desired operation itself, execution count and internal loop count are set according to the results from the previous module.

In the filter module, noise is eliminated to get only the meaningful measurements. Since measurements are normal Gaussian distributed, a confidence interval helps to identify noisy measurements.

In the extractor, timing information for relevant bytecodes is isolated out of the whole measurement data. It is not always possible to execute just one bytecode alone, since there are depending bytecodes which are executed additionally during a specific operation. For example the bytecode *smul* always needs two *sload* commands in advance to load parameters on the stack. The benchmark, however, just gives information about the execution time for the whole multiply operation. So, the timing for each dependent bytecode needs to be known for calculating the time for the bytecode of interest. Also, performance of the empty benchmarking loop needs to be measured, to clear that overhead out of the results. So, the test method looks like the following:

```
1  process() {
2      i=0;
3      while(i<L) {
4          runTest();
5          i++;
6      }
7  }
8
9  runTest() {
10     p1 = p1 * p2;
11 }
```

Listing 3.1: Example for evaluating the *smul* bytecode

In the test environment above, $runTest()$ contains the bytecode to evaluate itself. L describes the execution count for the desired test, as found in the calibration step.

As mentioned above, this is not always just one single bytecode but a sequence of more depending bytecodes. In case of the *smul* operation, the bytecode sequence could look like the following, here an assignment has been added to store the value back in a variable:

```
1 sload p1
2 sload p2
3 smul
4 sstore p1
```

Listing 3.2: Bytecode sequence for above multiplication

The only possible time to measure is, however, the overall execution time for that single test, which means all loop iterations including overhead for APDU transmission and similar. This overhead needs to be cleared out to get the isolated execution time for the desired bytecode. So, the execution time for an empty iteration needs to be measured, which is achieved by leaving the runTest body empty. With this information the runtime for one iteration of *runTest()* can be calculated along equation 3.1.

$$M_{smul+} = \frac{m_{smul} - m_{EmptyLoop}}{L} \tag{3.1}$$

Unfortunately, the value M_{smul+} contains not just the execution of *smul*, but also the time for *sload* and *sstore* operations. To isolate the execution time just for *smul*, all the other operations have to be cleared out along the next equation:

$$M_{smul} = M_{smul+} - \sum_{i=0}^{n} op_i \tag{3.2}$$

Here op_i stands for the execution times of depending operations, which are subtracted from the overall execution time of the desired bytecode.

The profiling step establishes a connection to the future application area of the desired Java Card. Following three application domains are concerned:

- Banking applications,
- Transport applications and
- Identity applications.

For each application domain, a set of characteristic test applets is defined. To find an average application domain specific performance mark, the influence of each characteristic applet to the entire application domain needs to be computed. For a final performance analysis of unknown Java Cards, weighted means of the measured performance are used. With this feature, cards can quickly be benchmarked for specific application domains.

3.1.1 Results

The high number of executions per specific bytecode allowed an expectation, that the timing measurement results would be normally distributed. Unfortunately none of the time performances showed a normal distribution. There were similarities for the same operation on different cards. Actually a change of the test environment from Windows to Linux showed differences. Obviously, the implementation of drivers and other software related parts has a great influence on the test results. The normality of results was also checked by a Shapiro-Wilk test, which uses some peaks of distribution. It calculates a value W which is an indicator for normality if it is close to 1. With a result of 0.884, this test showed, that the results are not normally distributed. A graphical view of the measurement for *sadd* is shown in Figure 3.2.

To validate the results from the measurement with conventional card access devices (CADs), tests were repeated on a high precision Micropross MP 300 reader. This reader has the big advantage to measure timing independently of the host system (see also 4.5.3) and can give most accurate results. It measures the time between the end of transmitting an APDU to the card and the start of getting the response. The measurement results did not contain so much noise and were normally distributed according to a Shapiro-Wilk test with $W \geq 0.96$.

Nevertheless, comparing the results found by a regular reader and those from a high precision reader shows that the mean value is a quite good approximation. For example, the mean value for a Linux environment contains only 3.42 % error compared to the more accurate result, whereas the Windows environment caused an error of 7.42 %. Both are values which are acceptable, especially because no special equipment is needed in those cases.

Except noise, the framework gives quite precise and accurate results. It does not need costly readers for an evaluation of the performance of a smartcard. Finally, this paper provides good information about how to measure performance of specific parts of applets.

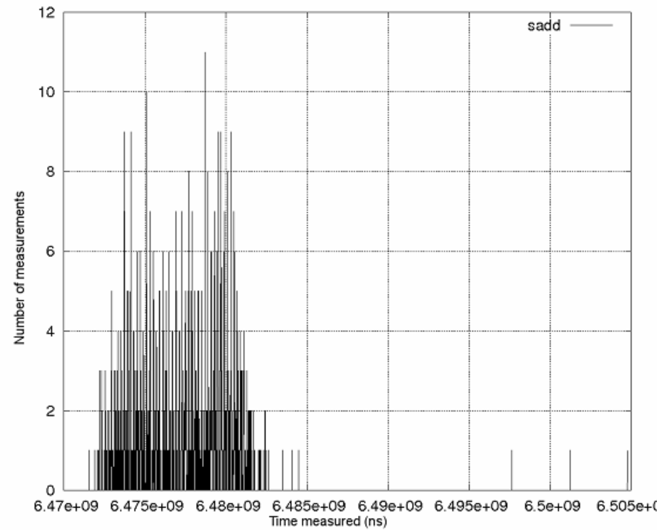

Figure 3.2: Distribution of timing measurement of the *sadd* bytecode [PCB09]

Although the results were not normally distributed in the non-precise measurements, the mean value seems to be a quite good approximation for timing values. It also showed an attempt, how to benchmark Java Cards for specific application domains.

3.2 Benchmarking Java Cards

In this diploma thesis, [Erd04] shows a method to benchmark Java Cards. Again, timing information for specific operations needs to be determined. This was done by quite the same process as mentioned before in [PCB09]. First measuring sample applications, then measuring the communication overhead to deduce the performance for the desired operation. Moreover, interesting approaches for a benchmark system are presented and also the impact of different test systems is discussed. Further, operations were split up in operation classes and Java Cards were benchmarked according to this classes, which are shown in the following:

- operators
- read- and write-access
- applet-methods and object-methods
- arrays
- Java Card language-elements
- Java Card system
- security
- Visa Open Platform

Further, application classes like banking, GSM, authentication and others have been defined. Each and every application class needs a defined amount of operations out of above mentioned operation classes. So, a formula for each application class was defined, which specifies the contribution of each operation class to the entire application.

Every smartcard in the test was evaluated concerning these operation classes. This allows a good statement how good smartcards suit to an application range. For example smartcards for banking should have good results in the security and Visa Open Platform values. The results are presented in a spiderweb view.

Additionally, Erdmann inspects the impact of different readers and test environments on the test result. Three measurement systems with different readers and different software equipment were implemented. In best case, one single card should get the same test results, no matter which measurement system is used. Erdmann devotes attention to this topic by performing the same test on the same card on different measurement systems. The maximum deviation from one system to another was about 20 percent when running a read/write EEPROM test as shown in Figure 3.3.

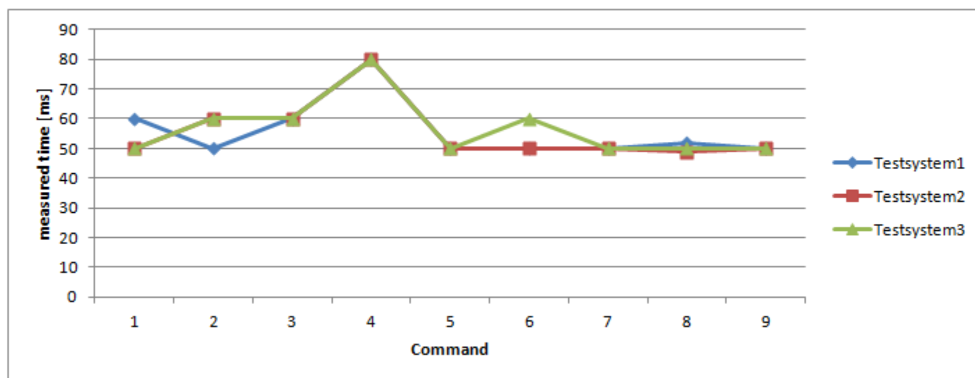

Figure 3.3: Impact of different measurement systems on timing [Erd04]

To test performance of security features, an applet with DES encryption and decryption has been developed. This applet contains key generation, encryption and decryption of differently sized data. First, the results for the overall execution time of the applets are shown and further detailed timing information for each cryptographic operation within the applet is presented. The detailed information allows interesting statements concerning the percentage of specific operations. There is no general relation between the percentage of the applet parts which is valid for all cards. Actually the deviation is quite big concerning different cards and measurement systems. The variation of the time difference is somewhere between factor 2 and 3 as shown in [Erd04].

3.3 Analyzing Control Flow in Java Bytecode

This work from [Zha] provides a good overview about how programs are constructed in Java. Since for our task, it is also necessary to understand how Java programs could be

split up, respectively how to find corresponding Java blocks for given native blocks, control flow graphs give an approach for this topic.

Since performance analysis is always depending on the internal control flow of a program, it is important to understand how Java applications are represented internally. This paper gives a good overview how applications can be modeled as control flow graphs. The key element are so called bytecodes, the smallest unit in the Java world. Bytecodes are generated out of the sourcecode by the compiler. Here, one statement in the sourcecode results in a bunch of Java bytecodes. This bytecode sequence is loaded into a Java interpreter the so called Java virtual machine. In the JVM bytecodes are executed one after another. Listing 3.3 shows how Java source is translated into bytecode.

```
1  Java Sourcecode:
2
3        short i=0;
4        while (i < 100)
5        {
6             array[i]= (byte)i;
7        }
8
9  Java Bytecode after processing by javac:
10
11       P:0134   03          sconst_0
12       P:0135   29 05       sstore            i
13       P:0137   70 0a       goto              +10 (P:0141)
14       P:0139   ad 45       getfield_a_this
15       P:013b   16 05       sload             i (S)
16       P:013d   16 05       sload             i (S)
17       P:013f   5b          s2b
18       P:0140   38          bastore
19       P:0141   16 05       sload             i (S)
20       P:0143   10 64       bspush            100 (0x64)
21       P:0145   6c f4       if_scmplt         -12 (P:0139)
```

Listing 3.3: Java sourcecode and corresponding Java bytecode

To generate a control flow graph, the bytecode sequence needs to be separated in so called *Basic Blocks:*
„A basic block is a sequence of instructions with a single entry point and single exit point: execution of a basic block can start only at its entry point, and control can only leave a basic block at its exit point. Thus, if control enters a basic block, each instruction in that block will be executed“ [Zha].

So a Java program will be split up into basic blocks and special blocks, so called leaders, with additional roles. Leaders are necessary when the program flow can change, like after conditional or unconditional branches, method calls and callbacks. Also, every statement which occurs immediately after a branch is a leader. After every leader, all instructions until the next leader are put into a basic block. Method invocations are put into a separate basic block theirselves.

To construct a graph, the found basic blocks need to be connected somehow. There are some rules, how to connect basic blocks. First, block u and v need to be connected if v follows the bytecode u except u ends in an unconditional branch. If it ends in an unconditional branch, then a connection to the successing basic block needs to be established. Subroutine calls need two connections, one for the jump into the subroutine and the other one for the return jump to the calling block.

A quite complex topic is exception handling, since nearly every instruction can throw an exception, which is an abnormal program flow. This topic is not concerned deeper here because for performance estimations such a program flow can be neglected. See also in [Zha].

3.4 Comparison of Java Cards and native smartcards

In this paper by [Fis06], native applications and corresponding Java Card applications are set in relation. The author tries to find a rule, how the execution time of those applications are connected. This is actually a similar problem as stated in this diploma thesis, but it showed that it is not possible to find a simple relationship between the two applications without knowing further details about the program flow. Further, some interesting approaches for measuring processing time on Java Cards are presented.

This work was done during an internship at a leading semiconductor company in Germany. It deals with the question how, respectively if a relation between the performance on a Java Card and the performance on a native card can be described. The initial situation are some pairs of smartcards (one native and one Java Card) where both cards of the pair execute the same application. Further, both cards have the same underlying hardware specification. The target is now to estimate the performance for an unknown pair of cards, where just the Java Card part is available.

Therefore Fischer deduces the performance relation between a Java Card version and a native version of some known pairs of cards. If this relation is roughly constant for all pairs, the found factor could be used to estimate the native performance for a known Java Card. To get performance information on both sides, benchmark code has been generated, on one hand on the Java side and then again in a native language. It is important, that the two code pieces behave as similar as possible to make the comparison work. The test was run as follows:

- Measure runtime relation of first pair of cards, $x_1 = t_{J1}/t_{N1}$.
- Measure runtime relation of some other pairs x_n and check if results have roughly the same value.
- If the factors are constant, measure Java runtime of the unknown card and estimate native runtime: $t_{N3} = t_{J3}/x_1$

During performing those runtime analysis it showed up that this approach doesn't work as expected. The relation between measured runtimes was varying quite a lot, depending strongly on the used card type.

The results were varying from factor 22 to factor 14. Possible reasons for that behavior are probably different versions of the Java virtual machine. Another reason is the T=1 protocol which uses so called waiting time extension (WTX) commands while executing for a longer time to let the terminal know the card is still working. This could also cause some performance impact.

Additionally this approach implies to have exactly the same applications on each card pair. The relation would just be valid for exactly that kind of application. In the following, a short summary of the results is presented:

- The relation between different cards varys quite a lot
- Different JVM versions make a performance comparison quite difficult
- It is not possible to estimate unknown cards performance with this approach
- The T=1 protocol probably causes performance impact as well

In the second part of his work, Fischer presents some approaches how to measure timing information on smartcards. Having accurate timing information is totally necessary for making reasonable statements. The easiest approach is of course using the PC clock for measuring execution times. Just compare the time when sending a command to the time when receiving the result. But there are several things which have to be kept in mind:

- PCs don't guarantee hard real-time, the scheduling mechanism is maybe doing a lot of other stuff between processing the request of interest
- Drivers can be interrupted by other hardware interrupts
- USB drivers need some protocol time
- Overhead is not constant, so it can not be cleared out by executing several times (jitter effect)

Because of all those problems, measuring time with a PC clock is more or less guessing. The next approach is using a transparent reader. This reader is put between card and terminal, logs the traffic on the I/O port and sends it over a level changer to the serial interface. Since there is a lot of processing hardware in such a transparent reader, its ability to send information in real time needs to be analyzed. By checking that on an oscilloscope, it was proofed that the transparent reader works without jitter and delay. So such a transparent reader allows to get accurate timing information, but it just gives raw data, so its not possible (with acceptable effort) to see detailed information about the data which has been sent.

Although the transparent reader itself gives timing accurate information, data still has to be tracked on a PC, which generates delays and jitter like mentioned above. To face

this problem, data needs to be recorded by a real time system.

Fischer uses in his work some real time hardware called JNut-Box. This box is sniffing the traffic between card and terminal and captures all information in real time. Additionally it contains a high-precision clock to analyze the timing behavior of the commands. The JNut-box needs to be parameterized with some start- and stop-pattern. If this pattern is recognized in the traffic, the timer is started resp. stopped.

Finally, this work gave some impressions how timing measurement can be done in an accurate way, which is totally necessary for a performance estimation. It also showed that it doesn't make sense to use just simple relations between Java Cards and native cards for such an estimation. Here, the link to the program flow itself has to be established. For further information, see [Fis06].

3.5 Embedded software execution time estimation at different abstraction levels

Portable devices are booming more than ever and since memory and processing capabilities are restricted, great effort is put in analysis and optimization of portable applications. Also the software development process has become faster and faster to release products in a very short time. To ensure this, it is necessary to have early performance estimations already at design time. The results from such an execution time estimation can lead to decisions, which hardware will be taken for that specific application.

This work gives interesting input on how application performance can be estimated. Although this work concerns estimations for applications written in C, the principal approach can be used for estimations on Java Cards as well.

There are different approaches to estimate timing behavior on different abstraction levels. One always has to make a compromise on how much time is needed for the estimation and the accuracy. The estimation on the lowest level, which is register transfer layer, needs by far the most time, but it is really accurate. Against that, a sourcecode analysis can be performed really quickly, therefore the result is less precise.

3.5.1 Techniques

The techniques for execution time estimation can be divided in following groups:

- *Register Transfer Modeling:* This is the lowest level for simulation. A model for every logical element exists which allows very accurate results, but the simulation time is quite slow. It focuses on the hardware-specific details of every component and allows to analyze the effects of changes on a very low level.

- *Instruction Level Modeling:* Here timing information for every microprocessor instruction is necessary, which causes quite a big effort. Additionally always pairs of instructions need to be measured with respect to the internal pipeline of the processor, which is not trivial any more, since about 100 000 pairs of instructions need to be measured. However, after these complex measurement is done, and the results are found for a specific hardware, the simulation is about 100 times faster than a simulation on register transfer level.

- *Black-box macro modeling:* It sees an application as a sequence of pre-defined software portions which perform defined tasks, so called black boxes. An application is modeled as a flow of such black boxes. Here it matters significantly, how these blocks are generated. They should be generic and reusable, so they have to be parameterizable. In C, most functions are concerned as basic blocks, whereras in assembler instructions which are usually executed together form those basic blocks. Since the performance of a black box also depends on the input (parameters) it is necessary to regard those parameters in the performance estimation. For example an encryption according to some kind of algorithm can be seen as basic block. For a performance estimation, however, the length of the key and data influence the performance.

- *Source Code Analysis:* This approach uses elements of the source code as a base for estimation. For every language element like branches, loops, memory-operations etc. performance information is needed. Of course, this performance is dependent on the platform it is running on. That also means, that the platform specification can be exchanged easily to compare performance with each other platform.

The previously mentioned techniques were basic approaches how to estimate performance on different levels. Now, those methods are tested concerning their accuracy and effort. First an instruction set simulator is used to generate a reference result. All other methods are finally compared to that. For this work, the basic block approach and a source code analysis are relevant. Those two methods are shown in the following two sections.

3.5.2 Basic Block Approach

The basic block approach was done in assembler layer, once by defining assembler blocks and secondly just by counting assembler instructions. The process is split into two parts. First a characterization of the application and after that the estimation itself. In the characterization phase, the algorithm identifies basic blocks and provides them with an execution time. The identifying is done by adding assembler instructions into the C source code to be able to separate the basic blocks in the later assembler file. After compilation, these boundaries are still available and serve as basic block separator. This code is analyzed so the number of assembler instructions per basic block can be evaluated. An overview of this process is shown in Figure 3.4.

The second phase contains the estimation itself. Additionally to the instruction count from the previous phase, also information about cache misses is considered in the estimation. So cache misses need to be counted per basic block and also timing for a cache miss

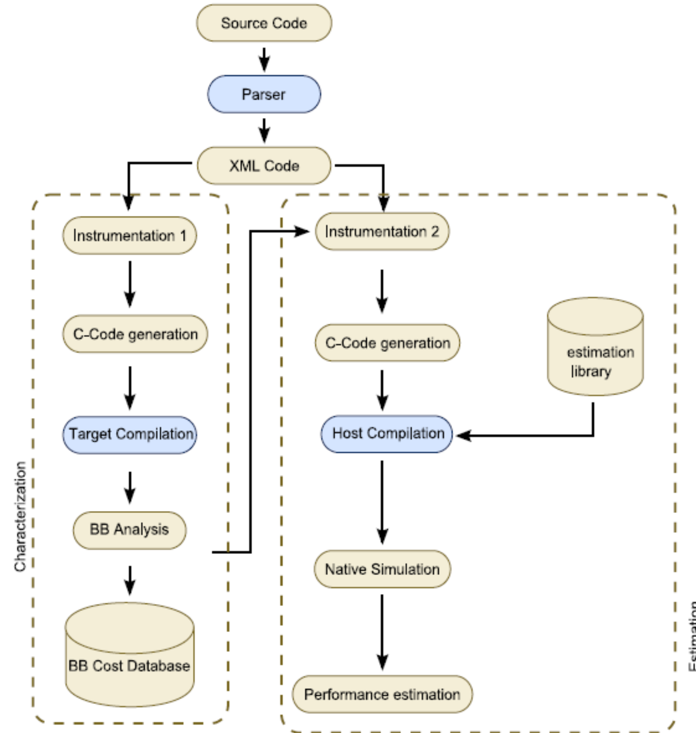

Figure 3.4: Basic block characterization and estimation [GASE]

is necessary.

A surprising fact is that the execution of an assembler instruction doesn't take a constant time. Reasons for that can be found in pipelining or buffer stalls. Unfortunately this impact is dependent on the application input. Since this information is not known at compilation time, a mean time needs to be added, which results in less accuracy.

3.5.3 Source Code Analysis

As second approach, source code analysis was concerned. Here the source code is split into elements of the C/C++ grammar. For each element, a constant execution time is assigned. This is of course platform dependent since C operators do not consist of the same machine instructions on every single platform. So the overall execution time can be written as

$$T = \sum_{n=0}^{N} T_n \tag{3.3}$$

where T_n contains the time for one operation. Down on assembler layer, every operation can be written as a number of instructions.

$$T = \sum_{n=0}^{N} \sum_{m=0}^{M} T_{mn} \tag{3.4}$$

T_{mn} stands for the execution time of instruction M in the context of operation n. As mentioned previously execution times don't necessarily have to be constant, but for this high level estimation a mean value $\bar{\mathrm{T}}$ is totally sufficient. I_n in the following equation stands for the number of assembler instructions for the operation n.

$$T = \sum_{n=0}^{N} I_n \bar{T} \tag{3.5}$$

A graphical overview, how operation based source code analysis and following time estimation works, is showed in Figure 3.5.

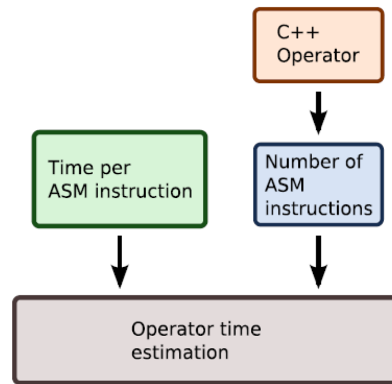

Figure 3.5: Timing estimation based on source code analysis [GASE]

3.5.4 Results

To see how good these estimations work, above mentioned methods have been tested on some popular examples in computer science. As reference model, an instruction set simulation was taken. Following table shows the accuracy and the speedup of other methods.

	Assembler blocks		C operations	
	Speedup	**Error[%]**	**Speedup**	**Error[%]**
Bubble	156	0.26	187	5.24
Factorial	34	0.36	68	20.55
Queens	163	0.72	489	2.54
Hanoi	34	0.74	69	0.51

Table 3.1: Speedup and error in different test cases compared to ISS based results

It shows, that an estimation based on basic blocks in assembler has a very low error. Additionally the speedup comparing to the instruction set simulation is also quite good.

Analyzing C source code leads to a bigger error, but also gains a high speedup concerning the simulation time which can be an advantage if fast results are needed. For further information see [GASE].

3.6 A SW performance estimation framework for early System-Level-Design using finegrained instrumentation

This work primarily concerns an estimation approach for applications in an early level of design. Since some applications require special features from the underlying hardware, this method allows to adjust the hardware to the applications needs, especially in meanings of performance.

At first sight, this doesn't have to do anything with an estimation from C to Java, but it contains quite interesting approaches for analyzing dynamic C application flows. This is the reason for mentioning this work here.

The framework should provide an opportunity to evaluate and analyze different system designs for a single application. Moreover, system parameters shall be changeable dynamically to see how those changes affect the performance. There are several ways to analyze performance for system level design. Again, they reach from fast and inaccurate to very accurate but slow. On one hand lies a statistically estimated representation of task timing, which is very fast but not very accurate. The other extreme is an instruction accurate analysis of the task, which delivers accurate results, but the simulation speed is very low. So obviously, the reasonable methods are located somewhere in the center. As shown in Figure 3.6 such an analysis will be done by a so called source code instrumentation. Here, the source code will be provided with function calls, which allow to give a statement about the operation's runtime (cycle count).

Source code instrumentation in a manual way produces a lot of manual effort, since a line has to be added to the sourcecode for every operation that should be tracked. Therefore it is necessary to model the timings for each task manually.

The central idea in this work is to have a process for a highly accurate, fine-grained source code instrumentation. Moreover, this source code instrumentation shall be done in an automatic way, so no user interaction is necessary. For that fine grained software instrumentation, a *Three Address Code Intermediate Representation* has been chosen. It shows all C operators and the majority of all memory accesses. Further, optimization steps had been performed already at that level, so just lines which are actually executed, are shown. So, the user just has to provide costs for each operation.

This automated instrumentation is done by the instrumentation engine of microprofiler [Pro11]. The tool provides runtime statistics like usage profile of C operators and slight timing estimations.

Further, the performance of memory accesses is concerned. Local variables are mostly held in registers, so an access is pretty fast and can be neglected. More time consuming

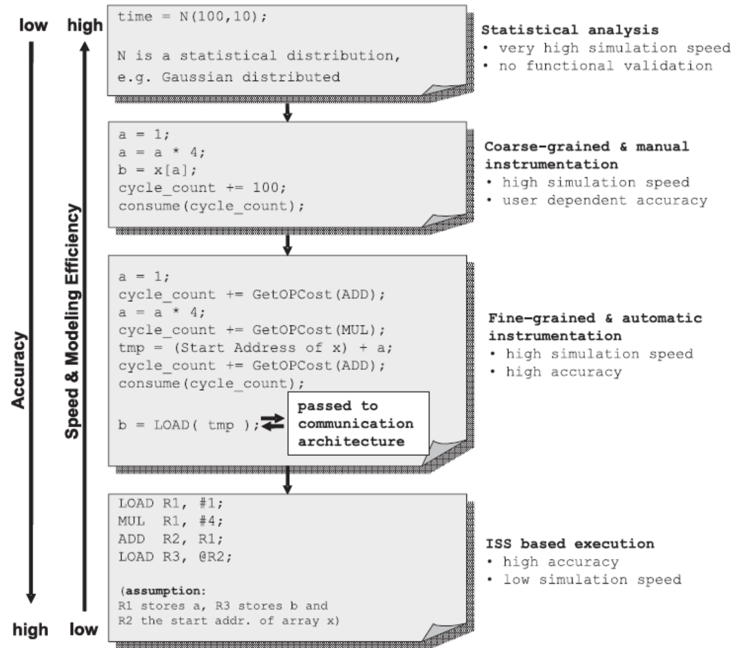

Figure 3.6: Software estimation techniques [KKW+06]

is the access to variables which are lying somewhere else, like on the heap. Such array operations are split up into more parts. First of all, the address needs to be calculated (which is an add operation itself) and after that the access operation itself is given to the communication architecture which loads the variable finally. There are different kinds of memory residing in the simulation which have different address ranges. The framework offers a method to change the memory area for every variable by so called memory maps. Each variable's address can be edited in that memory map so that it points to a different address range. This makes it easy to test the influence of different types of memory on the application.

The results are just presented concerning their accuracy of execution cycles compared to an ISS-based analysis. As first test case, the blowfish algorithm was taken and it showed that the execution cycles could be estimated with an error of about 8 %. Another testcase, the G.279 speech compression algorithm got an error of 7.5 % which is also quite good. The error concerning memory access count is a bit higher, with 14 resp. 20 %. Concerning the simulation speed the results showed a speedup of about factor 10 compared to the ISS based result according to [KKW+06].

Chapter 4

Design and Concept of the Estimation Method

This section describes the detailed design for an estimation of the performance impact when porting a native written application to a Java Card. First, the problem itself is described concerning its particular technical challenges. Further, some existing approaches are discussed regarding their qualification for solving this problem. After that, the performance estimation system is described, starting with a general overview and getting into a more detailed description of the single components. Finally, a process for verification of the results is presented, which serves as indicator how good the initial problem could be solved.

4.1 Problem definition

A lot of applications for Java Cards have been developed using so called native programming languages. With an increasing number of features, those applications got bigger and even more complex. Additionally, native implemented applications are platform dependent. Those applications can be ported to Java Cards to benefit from features like interopability and flexibility (for further description of Java Cards see Section 2.3). However, these advantages have their tax. In this case, those benefits cost performance because application execution on a Java Card takes more time than on a native platform. The reasons for this, and how this performance loss could be estimated, are presented in the following.

Applications for native platforms are developed in low-level programming languages like C or assembler. Although C is concerned as a higher programming language, in meanings of smartcard development, the low-level capabilities of C are used very intensely and C can be called low-level in this case. These programs are written and compiled for exactly one dedicated hardware platform. This makes the applications tailor-made to their underlying hardware, which means that changes in the hardware require a change in the application as well. Especially for assembler developed applications this means a lot of effort. On the other hand, developing on a very low level means a good capability of controlling hardware specific features, which results in a high performance for applications.

Also instructions are executed directly on the target processor.

Applications on Java Cards use a totally different paradigm, their sourcecode is written independently of the underlying hardware. In other words, an additional layer of abstraction has been inserted. Java Card applications are executed on the Java Card virtual machine (JCVM), a virtual processor for Java Cards. The virtual machine has to be implemented for every hardware Java should run on. This allows a great amount of flexibility for applets, since they are running on every JCVM. But this intermediate step in the virtual machine needs time, so the performance of applications for Java Cards is lower compared to the native version.

This work deals with the question, how this performance impact could be estimated. Further, this should be done without an implementation of the desired application on Java Cards, only the native application and its specification is available. First of all, the native application needs to be analyzed. This analysis is the base for concluding to a Java application which has the same behavior. This is not a trivial task since the architectures of those two platforms are different in a lot of respects:

- The native side is designed as *register architecture* while the JVM is a *stack architecture.*
- Instructions are executed directly on the native side's target processor while Java Card application contains bytecodes which are executed on the JCVM. Further, the JCVM executes bytecodes on the real processor.
- Very basic operations on the native side vs. powerful Java operations.
- Memory can be accessed directly in native code, while Java performs several memory access checks.

All these points make it quite difficult to find a universal mapping between native developed applications and Java Card applets. A simple mapping from instructions to bytecodes is not possible since instruction sequences do not appear in a regular way because of compilation effects. So the application will be analyzed on a higher level like source code analysis. Here one has to face the problem of different language constructions. For example it is not possible to access memory directly on the Java side. Some source code examples for not easily mappable elements are shown in Table 4.1. This means an automatic creation of an according Java applet model will not be possible, so this task will remain a manual step.

There was no related work found which deals with performance estimations between different platforms like native platforms and Java Cards. So, for the transfer step between native and Java something completely new had to be developed. Concerning benchmarking techniques the „Mesure Project“ (see 3.1, [PCB09]) gives some interesting input how to rate Java Cards by deducing and comparing the runtime of each single bytecode.

Further, [GASE] concerns performance estimation on embedded platforms. Java Cards theirselves are not mentioned in that work, cause it focuses more on the estimation of na-

Native source	**Java**	**Issue**
memcpy(a,b,10)	not available	Since Java does not allow direct access to memory, no copy operation of single memory areas can be performed. Instead, this has to be realized by copying whole objects or arrays (which results in method calls).
for(i=0;i¡10;i++) array[i]=0;	Util.ArrayFill(...)	Array initializations are done more efficiently by calling appropriate methods in Java.
*(p++)=10;	p[..]=10	Pointer arithmetics must be realized by array accesses.

Table 4.1: Native source code, which can not easily be mapped to Java

tive applications but the approaches can be used in a similar way.

Moreover, the Java model's performance needs to be analyzed. Therefore the Java model elements need to be provided with performance indicators. Thus, another task is to identify all those elements accordingly and analyze them concerning their runtime.

To verify this process, actual native applications are estimated according to the found process. After that, the same application is implemented as a Java applet, and the performance is measured. The estimation result is finally compared to the real result to get an indicator for the quality of the solution.

4.2 Overview

For a given native application on a smartcard, the performance impact when porting this application to a Java Card should be estimated. Following section will give a little overview how the problems from the previous section are solved, how the estimation system is going to work, and which steps are necessary to make reasonable estimations. Figure 4.1 shows the necessary components of the estimation system. The input is a *native application* (written in C or assembler), specified by a running example and a functional description (mostly this will be the source code). It is analyzed by the *performance estimator* (see Section 4.4) concerning its program flow and its dynamic runtime behavior). Dynamic runtime analysis is important for the performance estimation to have additional information to the static program flow. After this analysis, the estimation process has a model of the application. This model's Java performance needs to be estimated according to *estimation rules* (see Section 4.3). Those rules define the performance for each possible element in the model. With estimation rules and an application block model, the estimator can determine the *estimation result.*

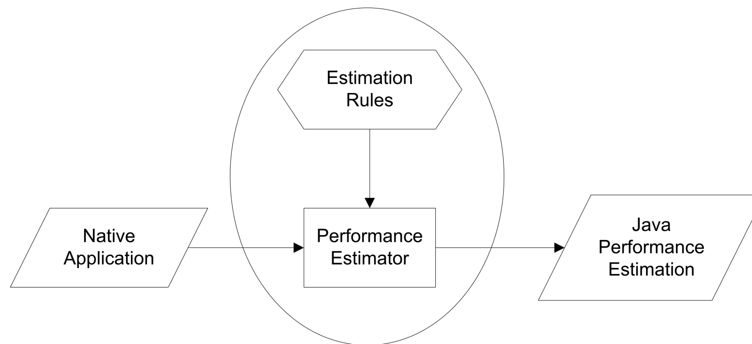

Figure 4.1: Overview of problem definition

4.2.1 Design for the application block model

The design in the previous section includes a model that represents an application's functional behavior. In this section a more detailed description how this model works, follows. An application is split up into blocks which represent defined operations in the program flow. These blocks are generic and independent of any programming language, so every possible application can be modeled with them. They can be very basic structures like elements which influence the program flow as well as complex blocks which cover a whole sequence of operations (i. e. encryption or decryption of data). Figure 4.2 shows an example for an application block model. It is a hierarchical sequence of generic blocks. The figure is already an actual example for an application block model, how it is going to be used in the future.

Worth mentioning are those blocks, which influence the program flow like loop- and if-blocks. Those need to be provided with special parameters like a loopcount for loops or a jump probability for if blocks. For loops, all subordinated blocks are executed according to that loopcount, while the condition evaluation has to be executed once more. This has to be considered in the overall calculation. Furtheron, if-blocks execute their subordinated blocks according to the given jump probability. If there is an else clause as well, another jump operation is executed which has to be considered in the calculation. That information is necessary for finally calculating the overall execution count for each single block in the specific application.

4.2.2 Modules design

The entire process of estimating the performance impact is split up in two parts. As mentioned before, first it is necessary to find some estimation rules which picture general performance characteristics of the platform applications should be estimated for. More detailed, this means they contain performance information about all possible blocks of the block model. On the other hand, there is the estimation process itself, which uses those rules to estimate the desired native application.

Relevant modules for these two parts are shown in Figure 4.3. The creation of estimation rules is based on an analysis of sample sourcefiles. So a *Set of Sourcefiles* is

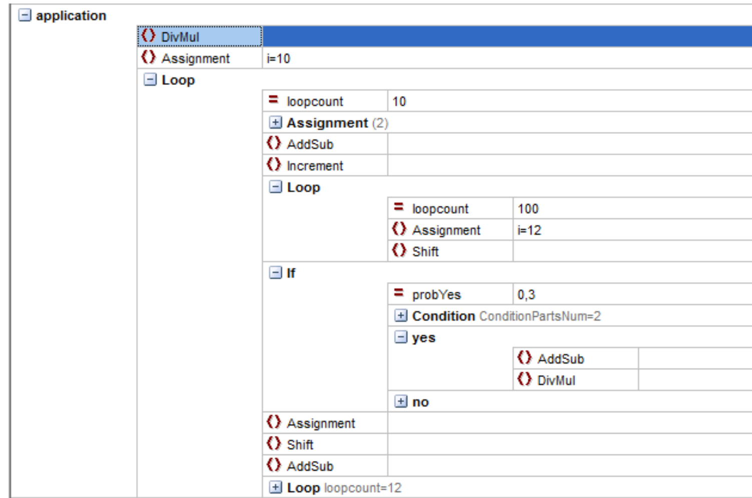

Figure 4.2: Design of an application block model

put into an analysis process which analyzes the performance according to the *Platform Specification*. For each sourcefile, runtime information is set in relation to its contained blocks. So the outcome are estimation rules, which provide performance information for each specific block, stored in a database, further on called TimedBlockDB (for a more detailed explanation of TimedBlockDB see the logical view in 4.3.1).

Estimating native applications starts with having an *Application Specification*. This can be either sourcecode or a functional description of the application. For an estimation, this application needs to be modeled according to Section 4.2.1, which results in an *Application Block Model*. The *Performance Estimator* gets the previously generated Application Block Model and uses estimation rules to estimate the performance of the modeled application. The estimation result contains a performance estimation for the desired application.

4.3 Design of setup of estimation rules

The creation of estimation rules is necessary to get performance knowledge about the target platform. In detail this means that every possible block needs to be provided with performance information.

To get performance data about each possible block, a sufficient amount of sample programs need to be analyzed. That's done by simulating those sample programs on a SystemC model. The model allows to get a detailed output which instructions have been executed. So a certain number of instruction cycles can be assigned to a sample program. A sample program itself consists a certain number of defined blocks. With a sufficient number of training applications, every block gets a performance attribute.

In most cases, no basic block can be measured straightforward, but there are always

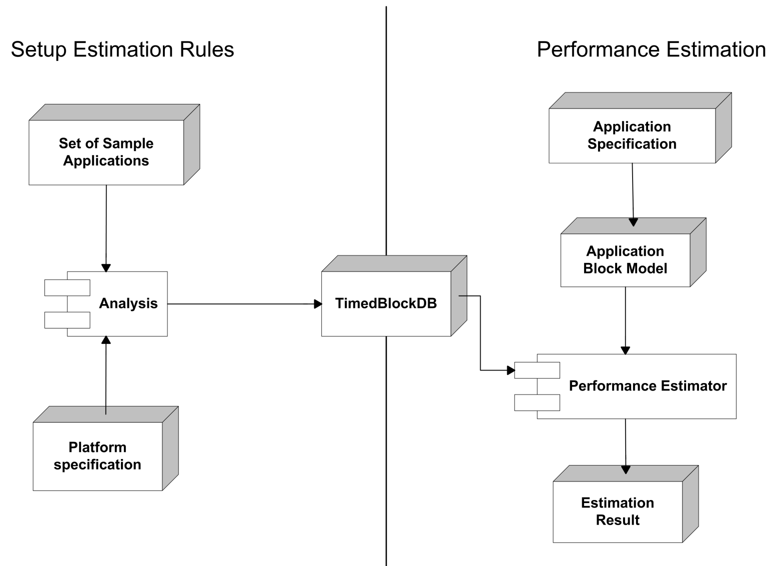

Figure 4.3: Performance estimation module overview

environmental effects like the overhead which has to be measured and cleared out. This overhead has its reason in communication time which is needed by different protocols, transmitting time and similar stuff. Secondly in most cases each sample program will consist of more than one different block, so the measured cycles are always a linear combination of more blocks. In the end it is going to be a linear system of equations where at least one equation per block is necessary to be able to solve the system of equations.

In the following, the setup of estimation rules is shown from different viewpoints to get a better overall idea of the system.

4.3.1 Logical view of estimation rules

Figure 4.4 shows a logical view for the setup of estimation rules. A sample application is modeled as a defined sequence of *GenericBlocks*. This is stored as a sample application block model. After simulation of the application, logfiles are written which contain the corresponding timing information to the previously defined block model. Those two are passed to an analysis process which gives specific timing information for each block. The result is a list of TimedBlocks, which are stored in TimedBlockDB. The reason for two kinds of blocks (GenericBlock and TimedBlock) lies in platform dependency. A GenericBlock is valid for all platforms and can be used to model an application without any knowledge of the platform. Against that, a TimedBlock contains timing information about one block on a specific platform.

4.3.2 Tools view for creating estimation rules

This view shows necessary tools for creating the TimedBlockDB and how those tools interact together. An overall illustration is shown in Figure 4.5.

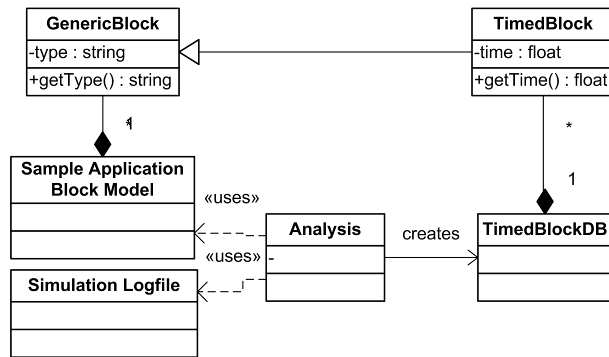

Figure 4.4: Logical view for creation of TimedBlockDB

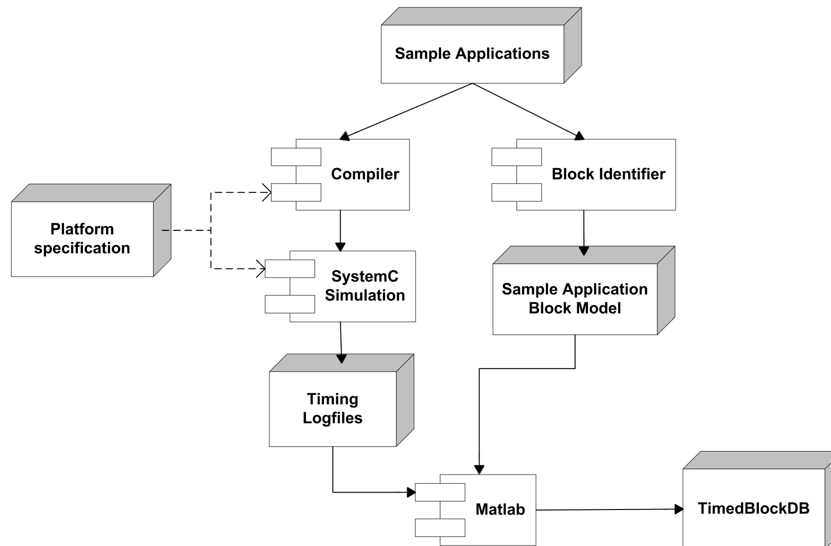

Figure 4.5: Tools involved in creating TimedBlockDB

As mentioned before, one needs a bunch of sample applications to create an appropriate model for the timing behavior of certain blocks. These files are compiled and run on a SystemC simulation (or another reasonable hardware model) of a Java Card. The platform specification in this design model defines properties of the underlying hardware. It determines which compiler has to be used and defines details of the SystemC simulation. The simulation offers a mechanism that allows the user to set a flag, so the simulator starts tracking the operation codes that are executed until the flag is unset again. Further, a unique number is passed to the simulation to make the sections (time range) identifiable. With this, the operation codes for certain sections in sample applications can be logged. As a result, the SystemC model writes timing logfiles, which have to be processed and split up in certain sections. Those sections are the base for following analysis.

Further, the sample application is analyzed by a block identifier. This tool models the sample application as a hierarchical structure of blocks, a so called block representation. However, it is also necessary to know the execution count for each block. To ensure this, control structures like loops have to get a loopcount whereas branches need jump probabilities for at least one path. With that information one can calculate exactly the execution count of each and every block in the sample application.

Now, the sample application's block representation can be set in relation to its runtime. The previous steps are executed for every single sample application. Their results are finally passed to a *Matlab* procedure, which puts data together as a linear system of equations. To solve that, at least as many sample applications as generic blocks are necessary. Additionally these samples must produce linear independent block sets, otherwise no information can be acquired. As an output each and every block gets its execution time for the previously defined platform. This information is stored in TimedBlockDB.

4.3.3 Process view for creating estimation rules

In the process view, dynamic aspects of the system are concerned. It shows, which processes are necessary and how they are interacting together. Additionally parallel processes can be identified and treated accordingly.

The aim is to get exact performance information for all possible blocks on a desired platform. Therefore, sample applications which cover all of those blocks have to be generated. These must contain certain code which enables performance measurement and tells the SystemC model the relevant program parts. These applications have to be compiled for a specific platform and are executed on a SystemC simulation. During execution, the simulation writes timing logfiles, which show the executed instructions for the whole application. These logfiles need to be analyzed first to filter only the necessary sections which can later be associated with certain application parts.

Parallel to that, the sample program has to be modeled as a sequence of GenericBlocks according to 4.2.1 in the „identify blocks“ step. Further, runtime information needs to be added to this model to determine the execution count of each block („analyze runtime

behavior“). Results from the two previous steps allow to model the application as *Sample Application Block Model*.

The simulation and block modeling is performed for every sample application, so that sufficient data for each block is available. All those results are finally combined and a cycle count for each block can be calculated. An overall view of this process is shown in Figure 4.6.

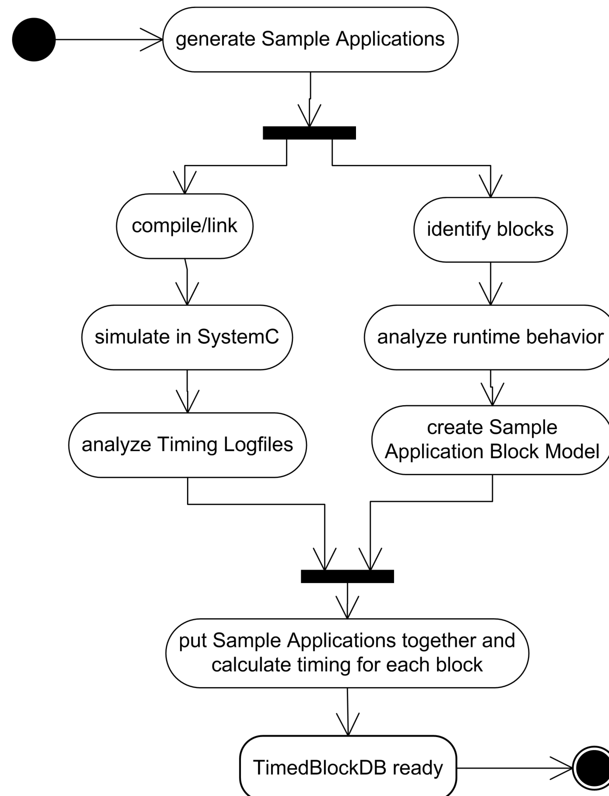

Figure 4.6: Process of getting performance information for unknown blocks

4.4 Design of performance estimation

In this section, the design for an estimation of native applications is focused. It describes which steps are necessary and how they interact with each other. Different views consider the topic from individual angles. Beginning with a logical view, which gives an overview about the functionality offered to the user, further an outline of all used tools for the estimation is given. In the process view, especially the chronological sequence of the steps is concerned.

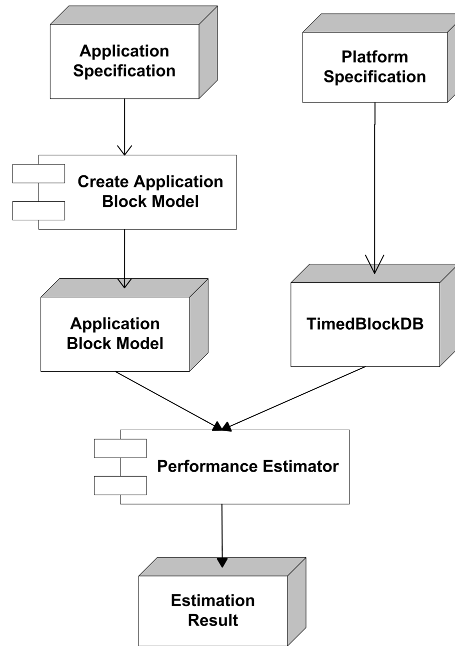

Figure 4.7: Tools involved in estimating an application

4.4.1 Tools for estimation of an application

Base for estimation is a well specified application. This can be either a functional description or the application's sourcecode. To estimate it, the user needs to know about the program flow and which operations are needed (i. e. cryptographic operations). With that information the „Create Application Block Model" step is creating an Application Block Model, that means the whole functionality of the desired application has to be modeled with previously defined GenericBlocks. Additionally, a previously developed TimedBlockDB (see 4.3 for the desired platform (specified by Platform Specification) is necessary. The TimedBlockDB contains timing information for all GenericBlocks for a specific platform.

Design of the Performance Estimator

The Performance Estimator is the central tool, which calculates the performance estimation for the desired application. First step in the estimation is to calculate an overall execution count for all GenericBlocks used in the application. Since the Application Block Model is a hierarchical structure of GenericBlocks, the overall counting of every single block needs to be performed accordingly. For example, a loop block contains various other blocks; those blocks are executed as often as indicated in the loopcount attribute of the loop block. The process is similar for branches (if blocks): Here all sub-blocks need to be counted according to the jump probability. In general, for every block that could contain sub-blocks, a certain way of treating it during the estimation needs to be designed.

After the Application Block model has been analyzed and the execution count of GenericBlocks has been calculated, the time for an execution on a specific platform can

be estimated. Therefore the TimedBlockDB for the desired platform is needed. Further, timing information from TimedBlocks is applied to the appropriate blocks from the model which finally allows to calculate the overall performance for the application.

Those considerations were kept quite general, which means this approach can be used for estimating every application for every platform. In our specific case, the estimation platform is a Java Card.

4.4.2 Logical view for estimation of a native application

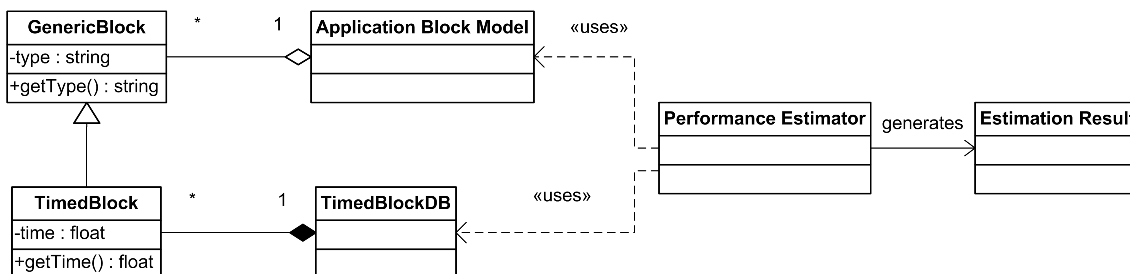

Figure 4.8: Logical view for estimation of a program

The logical view for estimation of an application is shown in Figure 4.8. Estimating a specified application first requires a well defined Application Block Model, which consists out of GenericBlocks. TimedBlockDB is a list of TimedBlocks which contains performance information about every block on the specific platform. It inherits from GenericBlock and has an additional attribute containing the timing information for the specified platform. The estimator analyzes the Application Block Model to determine the overall execution count (see Performance Estimator in Section 4.4.1), of all blocks in that specific application. Additionally, the generated TimedBlockDB from the setup step is evaluated and combined with the application block model by the Estimator which calculates the overall timing for the specific application.

4.4.3 Process view for estimating a native application

The process view shows which processes are necessary for the estimation of a native application and concerns their chronological order as well. Figure 4.9 gives an overview of the necessary steps. As already mentioned in the overview, it is not possible to estimate native applications automatically right away. There are manual steps in between, especially the creation of the Application Block Model. It starts with an application specification, a description what the application does and how this is implemented at the moment. From that, information about needed blocks is extracted and the program flow is modeled (*identify blocks*).

After block identification, every found block has to be examined concerning its execution count. For an accurate estimation it is totally necessary to get its exact execution count. This can be achieved with the help of so called source code instrumentation (see 3.6). Hereby, certain lines are added to the sourcecode which count the execution of

certain parts of the application. Since the newly defined blocks do not necessarily need to appear in the old application due to platform specific differences, this step needs to be done manually. The instrumented application is finally run and runtime information like loopcounts and jump probabilities for branches can be extracted (*analyze runtime behavior*).

With information from the two previous steps, a fully defined Application Block Model can be generated (*create Application Block Model*).

As final step, the Application Block Model is estimated using the Performance Estimator explained in Section 4.4.1 (*estimate Application*). The result is an estimation for the given native application if it was running on a Java Card.

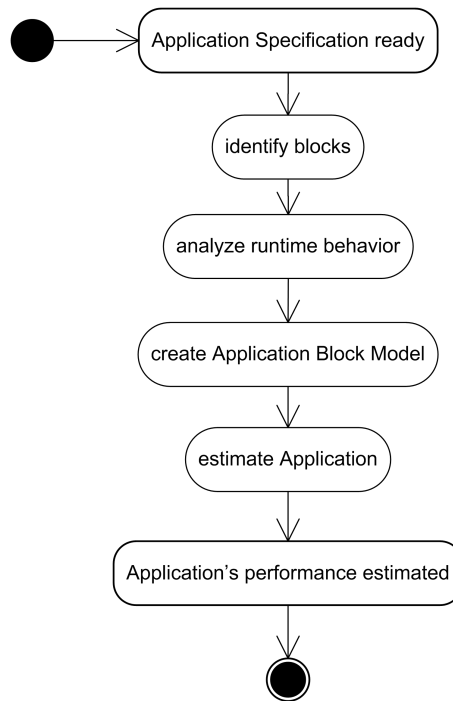

Figure 4.9: Process view for estimating programs

4.5 Approaches to get accurate timing information

This section contains some information about the approaches to measure processing time on smartcards in an accurate way. Since those times are in a range of micro- to nanoseconds, typical straightforward methods like using a clock on the calling device are failing because of their lack of accuracy. Thus, more sophisticated ways of measuring time have to be established.

Figure 4.10 shows the different communication layers when interacting with a smartcard. The processing happens in the smartcard itself, so the most accurate way would

be to measure performance directly in the card. There are approaches how to do that with an internal clock on the card, but the timing is depending on hardware clock, which is not always a fixed value, so the values are varying. Further, special functions need to be provided by the card's operating system to measure it that way. Not all smartcard operating systems support this, so a different solution needs to be found.

Since measurement can not easily be done directly on the card, time has to be measured somewhere between the calling application and the processing smartcard, in Figure 4.10 the top and bottom layer. The nearer at the software layer this is done, the more overhead is measured and needs to be cleared out somehow. There is overhead when transmitting the call to the PC/SC interface of the smartcard reader. Also, the PC/SC driver itself generates overhead while communicating with the reader hardware. The reader itself also contains some processing which consumes time. APDU processing also needs some time until the desired operation is finally executed. Of course, this overhead is produced twice, once when transmitting the command to the smartcard and once when getting the result back. To top it all, this overhead can not be considered as constant during several executions, so a clearing task is everything else than trivial. Since a PC contains a scheduling mechanism, you can never be sure what else is done during processing. When one thinks of those many layers between software and finally the operation processing step on the card, it becomes clear that the measurement must be done as near at the card as possible.

In the following sections, some approaches for getting accurate timing information are shown.

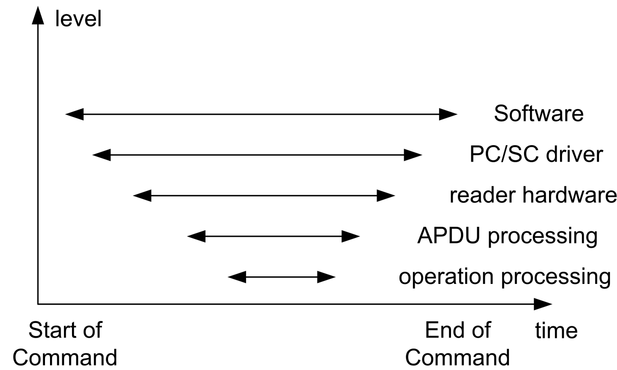

Figure 4.10: Time consumed on different levels of smartcard communication

4.5.1 Using the PC clock

As mentioned before there are a lot of layers an APDU needs to traverse until it gets executed on the card. There are USB stacks, drivers, hardware protocols just to mention a few and each of those steps takes time. To make it even more complex, the time needed in those layers does not necessarily have to be constant. A reason for this is a scheduling operating system which distributes resources of course in a reasonable way, but unfortunately not in the same way every time an operation is executed. So it is possible,

that the overhead is varying a lot from one measurement to another measurement. This makes it difficult, or nearly even impossible to clear out this overhead. Additionally the pc clock's resolution is far beyond an acceptable level for measurement of processing times in the microsecond area.

4.5.2 Using an oscilloscope

A real accurate way to measure processing time is to probe the input/output (I/O) pin of the smartcard with an oscilloscope. During communication, one can see the voltage trace of the sent commands and also the sent responses. So the pauses between command and response are the time needed for processing. This time can be measured quite easily with an appropriate oscilloscope. A communication trace is shown in Figure 4.11. In fact, there are two traces in this image, the yellow one shows a native application while the blue one states a communication trace of an application on a Java Card. The solid vertical bars are an indicator for active communication, the pauses in between are processing time.

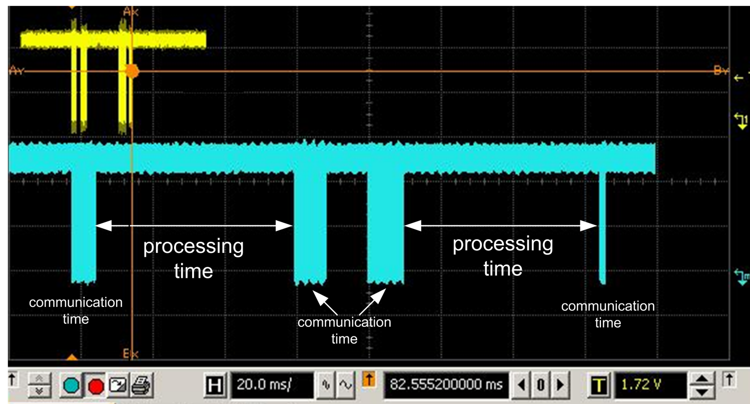

Figure 4.11: Measuring processing time with the oscilloscope

Still, measurement and the analysis of a voltage trace is a manual task, and for the small amount of measurements in this case it is not necessary to put much effort in developing an automated analysis process. Of course, if a test series with a huge amount of measurements shall be performed, an automatic analysis process for those voltage traces needs to be implemented. For now, oscilloscopes offer sufficient support to measure time between rising and falling edges with an automatic cursor.

This kind of approach is of course only applicable for cards which operate in contact mode. In contactless mode, the process of identifying commands and responses is much more difficult due to protocol reasons. For contactless cards, the following measurement method with a spy would be a quite good alternative.

4.5.3 Using a hardware spy

A spy is special hardware which is able to monitor the traffic between card and card access device in detail. It also allows to measure physical parameters like the magnetic field strength, impedances and so on. There are two types of spys: Either they are just

monitoring devices which are put between card and reader (so called transparent reader) or they are full qualified readers theirselves. The first option is used when the low-level communication between card and CAD is unknown, for example when the behavior of an unknown application with an unknown reader shall be examined. On the other side, there are spys which have all functionalities of a reader and can be used as readers as well. But there are further features which allow to measure communication properties very accurately. As a side effect of monitoring commands and responses, it can also show the time in between, which is the actual processing time we are interested in. In fact, it uses a similar approach as the oscilloscope but in a more sophisticated way. Further, it is also applicable for contactless cards since the processing is done automatically. Such a spy is for example the Micropross MP 300 as stated in [Mic11].

4.5.4 Using a simulation on a hardware model

This method is different to the previously mentioned ones since the application is not executed on real hardware, it's done in a simulation. That's the most accurate way of measurement since it allows to get an instruction accurate result cause there is no overhead which needs to be cleared out. This information can be used for evaluating which instruction is executed how often. Also processor cycles can be logged, which gives highly timing accurate results since execution time is directly dependent on the number of performed processor cycles. To enable meaningful measurement, the application to measure needs some special code in it which tells the simulation when to start and stop acquisition, a so called source code instrumentation. The manner of doing this instrumentation is simulation dependent, however, this means, an instrumented application can only be executed on that one dedicated simulation.

The interface to such a simulation is implemented as a PC/SC driver. Instead of accessing a smartcard reader, the driver transmits the commands to the smartcard simulation. So, it is pretty easy for the calling application to change between simulation and actual card.

After execution of the simulation one needs to analyze the generated logfiles to get instruction/cycle information for specific pieces of code. This method's main drawback is mainly the performance of the simulation itself. Depending on the simulation, it will be way slower than on real hardware, and additionally the deceleration factor grows with higher execution times. So a lot of time is needed for simulating programs on a cycle accurate (or even instruction accurate) simulator. Also, a change of the application's source code is necessary to provide start and stop points for the measurements.

4.5.5 Used method in this work

In this work, measurement is done with the oscilloscope on real hardware as well as a hardware simulation in SystemC is used. The results of the simulation have been used for creating the estimation rules. For verification, both methods were used and compared to each other concerning their accuracy.

Chapter 5

Implementation of the Estimation Method

The implementation of the performance estimation process in this thesis is split into two parts according to the design chapter.

First, in the setup step, timing data for a specific platform is generated by analyzing sample applications. This step is done by simulating user generated sample applications. Also, the block model of these samples has to be created. This block model combined with the simulation data allows to get the performance information for every single block. The outcome of this step is a complete database of blocks provided with timing information for a specific platform.

Secondly, a tool to estimate the performance for a native application has to be implemented. As an input for the estimation, a block model of the application is needed. Further, the timing database for the desired platform has to be known. The combination of these two allows finally to estimate the performance of the application.

But first of all, a reasonable way to describe those block models need to be found. The following section shows, how an application block model as designed in Section 4.2.1 can be implemented.

5.1 Block representation

Applications in general can be seen as a sequence of commands which is specified by the underlying program flow. The program flow itself has a hierarchical structure, which means that there are some structures which can contain other structures. Exactly that has to be modeled by an application block model. Thus, a method to model a hierarchical structure of blocks has to be found. For that purpose, an application block model is represented as an extended markup language (XML) file.

The application block model consists out of some elementary blocks, which can't contain subordinated blocks for basic operations like binary operations, shifts and similar

Basic Blocks	Control Blocks
Add	If/Branch
Sub	Loop
Multiply	Condition
Divide	
Increment	
Decrement	
Shift	
Assignment	
Binary operation	
ArrayAccess	
Cryptographic operation	

Table 5.1: Examples for different generic blocks

things. Further, it contains control structures like branches, loops and comparable which necessarily have to contain other blocks. For control structures, further information is required which tells about how often the subordinated elements are executed. This is applicable for loops where the loopcount has to be specified, whereas branches need information about execution probability of each path. All of this information needs to be tracked in the block model of the application to estimate. Some generic blocks to model applications are shown in Table 5.1.

Since not every block can be followed by any arbitrary block, the structure of the XML file has to be restricted. This is done by a document type definition (DTD) which controls the possible sequences of blocks. In following Listing 5.1 a part of the DTD is shown.

```
1  <!ELEMENT Assignment (#PCDATA)>
2
3  <!ELEMENT Add (#PCDATA)>
4  <!ELEMENT DivMul (#PCDATA)>
5  <!ELEMENT Shift (#PCDATA)>
6  <!ELEMENT BinaryOp (#PCDATA)>
7
8  <!ELEMENT Loop (Assignment | Loop | Add | Sub | Mul | Shift | BinaryOp ) >
9  <!ATTLIST Loop
10           loopcount CDATA #REQUIRED>
11
12 <!ELEMENT yes (Assignment | Loop | Add |Sub | Mul | Shift | BinaryOp ) >
13
14 <!ELEMENT no (Assignment | Loop | Add | Sub | Mul | Shift | BinaryOp ) >
15
16 <!ELEMENT Condition (Add | Sub | Mul | ArrayAccess |
```

```
17     BinaryOp |Increment )*>
18
19 <!ELEMENT If (Condition, yes, no?)>
20 <!ATTLIST If probYes CDATA #REQUIRED>
```

Listing 5.1: DTD for block representation

As shown, there are several special blocks which are not elementary, for example *Loop-* and *If-blocks*. They also need to get special treatment in the estimation process:

- **Loop:** A loop can contain all other possible blocks, as well as loops theirselves. It needs an execution count, which tells the system how often the containing blocks are executed. This information has to be determined through a dynamic analysis of the application during runtime.
- **If/Branch:** An *If-block* contains a *Condition-block* and two blocks for the two possible paths, *Yes* or *No*. It splits the program flow at a certain point and executes either one path or the other. For a detailed estimation it is necessary to know the probability that the true-path is executed which is stored in *probYes*. In case there is a no-path existing, it is executed with a probability of $1 - probYes$. With this information is it possible to determine the execution count of subordinated blocks.
- **Condition:** A condition is a necessary element for a *Branch-block*. Since conditions can contain quite complex calculations, it is necessary to know which operations are executed within that part. Further the number of comparisons in the *Condition-block* needs to be known. Internally, every comparison is executed as a jump operation so their number is affecting the runtime. Moreover, these conditions can contain every kind of operation, even assignments are possible.

The above listing explains some details about those control blocks which affect the application's program flow. A detailed description of the stated basic blocks is not necessary here, since those are basic operations in a programming language which need no further explanation. The block „Cryptographic operation“ stands as an example for all kinds of encryptions and decryptions using any cryptographic algorithm. It will be a user's task to define the needed cryptographic blocks for his certain application.

5.2 Setup of estimation rules

In the setup step, timing characteristics for all possible blocks need to be determined. This is done by creating sample applications and analyzing them. All possible blocks are defined in a central point which is the DTD file. The flow splits up in two parallel parts at this point. First, the runtime analysis for a specific sample program is performed, which gives performance information for the entire program. This can either be done by simple timing analysis, or an instruction accurate mechanism. Secondly, the sample program needs to be modeled as block representation. These two results have to be put in relation to each other to find the performance for each block.

5.2.1 Analyzing sample application performance

As mentioned in the design chapter, there are more alternatives to get performance information about a smartcard application. For advantages and disadvantages of the single approaches see Section 4.5. To get timing information for sample applications, we decided to use the most accurate method, which is done by simulation on a system model. We used a SystemC model for a smartcard microprocessor with a running Java Card operating system. That means, the hardware layer is substituted by a software simulation. All other layers are the same as on a real smartcard, as shown in Figure 5.1. The sample application accesses methods from the Java Card class library, which are finally translated into bytecodes. Those bytecodes are executed on the JCVM, which accesses the Java Card operating system.

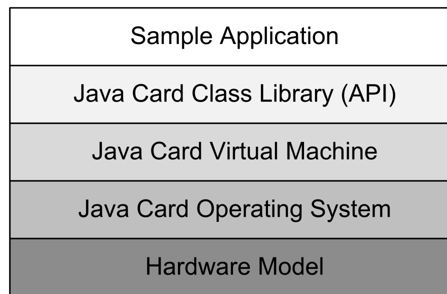

Figure 5.1: System model for getting instruction accurate performance information

Sample applications are designed as applets for Java Cards which contain the desired operations. The applet is compiled for the specific platform and loaded to the Java Card simulation. After installing, the sample application can be executed on the simulation.

In general this method is independent of the underlying platform, so this approach is applicable for every other platform as well, not just Java Cards. Only the simulation and the compiler need to be exchanged if the performance shall be analyzed for another platform. An overview of this flow is shown in Figure 5.2.

This SystemC model offers a functionality to track all operations which are executed. To track just the operations which are associated to the desired operation in the sample application, the simulation needs to get information when the operation starts and ends. This is done by so called *SFR*s (special function registers) in the simulation. In detail, this means that a defined value in that SFR causes the system to track all the executed instructions and assign them to the corresponding section. There need to be a mechanism to tell the simulation the value of the SFR. Therefore, an API function has been added to the operating system, which is able to set and reset this value. This allows to set and reset the SFR also from the Java layer. An example for this can be seen in Listing 5.2.

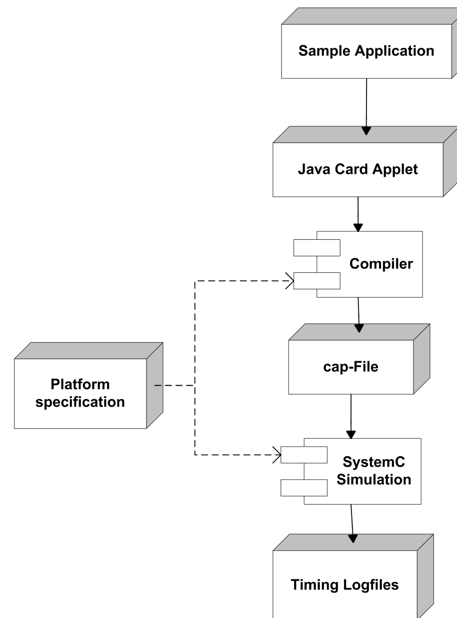

Figure 5.2: Steps for analyzing performance of a sample application

```
1 private void pTest() {
2     setSection(10);
3     x=2+x;
4     setSection(0);
5 }
```

Listing 5.2: Sample source code to set and reset the SFR

This example causes the simulation to start tracking instructions before the add operation and stops tracking after it. The outcome of this process is a timing log file which contains instruction information for all sections which had been set during the applet execution. Actually the system just tracks starttime and endtime of the operation and which operation is executed at what time. After that, it merges instructions with corresponding sections over time and gets the instructions per section. The measurement system is topic of another diploma thesis, [Pöl11], who implemented a detailed analysis of the executed operations. Therefore, every possible operation for that processor was assigned to one of the following operation classes:

- Arithmetic operations
- Logical operations
- Data transfer
- Boolean variable manipulation
- Program control

The outcome of this step is a timing log file which contains the central processing unit (CPU) cycle count per section, split up into the previously mentioned operation classes.

The tool for analyzing the CPU cycles already performs a detailed analysis concerning operation classes. However, for a performance estimation it is sufficient to have the overall cycles for each section, which are deduced by summing up all operation class specific cycles. Splitting up the entire simulation in sections makes it possible to analyze several sample applications during one simulation, simply the SFR that contains the section needs to be set accordingly. These files look similar to following Listing 5.3.

```
1 Section10:
2 25   Cycles for Arithmetic operations
3 86   Cycles for Logic operations
4 122  Cycles for Data transfer
5 9    Cycles for Boolean variable manipulation
```

Listing 5.3: Sample timing log file for executing an addition on a Java Card

The overall CPU cycle count finally needs to be combined with the sample application block model to determine the cycles per block.

5.2.2 Sample application block model

After simulation of a sample application, it has to be modeled as a sample application block model to know which generic blocks are executed in the application. Since the block representation is done via XML-files, this can be done by any XML Editor, the only restriction is, that its structure has to be according to the DTD. The corresponding block model for our sample application in Listing 5.2 is shown in Listing 5.4.

```
1 <application>
2 <Assignment></Assignment>
3 <Add></Add>
4 </application>
```

Listing 5.4: Sample application block model for the application stated in Listing 5.2

The task of identifying blocks is shown in Figure 5.3. The found sequence of blocks stands for a general functional description of the native application. Especially the modeling of C applications in Java brings up some difficulties. It is no trivial task, since language constructions in C do not necessarily have a corresponding counterpart on the Java side. One just needs to think about pointers and all related features, those are not available on the Java side. Thus, it is not easy to find a block representation for a native application automatically just by source code analysis. Some reasons for that are shown in the following:

- no direct memory access via pointer
- memory management has differences
- different memory areas on the Java side
- object-oriented programming on the Java side

The previously mentioned facts lead the block identifier to be a manual process. This means, one needs to analyze the application's source code and find a way to model this behavior with generic blocks. It also might happen, that there is no corresponding block

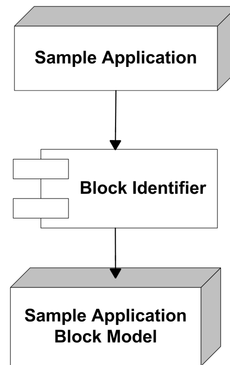

Figure 5.3: Identifying blocks of the source file

for certain statements, for example when direct memory access or similar operations occur. In this case, one has to think how the same behavior could be implemented with existing blocks. If this is not possible, a new block which performs the needed operations needs to be created and inserted into the model. Further, the dynamic aspects of the application needs to be analyzed and also put into the model. Those give information how often each block is executed. For sample applications, this step is quite easy, since they won't contain complex block constructions.

5.2.3 Creating sample applications

Thus, a set of sample applications has to be created that covers all blocks. Most of the operations have a really fast runtime behavior (in the microsecond area), which makes it quite difficult to measure their execution time. To make the measuring process easier operations are executed several times in a row to make the time period longer. A further advantage of the repeated execution is the minimization of the effect of possible statistical outliers during the execution.

For Java Cards, following test environment has been implemented. Sample applications are part of a Java Card applet, where the desired operations are executed. The execution of every application on a Java Card has to be triggered by APDUs. In case of simulation on a hardware model, all needed operations can be put into one application since the association between sample application and corresponding section can be set in the application itself. If performance shall be measured by an oscilloscope, just the processing time of the entire APDU command can be measured (as shown in Figure 4.11). So every operation needs to be put in a separate command in that case. To enable the sample application to be measured both ways, every single sample application gets its own APDU.

First of all, the process method checks, if it has been called by a select command. The application then contains a splitting according to the APDU and calls an appropriate method where the desired operation is executed. The *process* method is called every time an APDU is sent to the smartcard. It distinguishes according to the value of the INS byte, which method will be called. So the INS byte is an indicator for the operation to execute.

```
1 public void process(APDU apdu) {
2
3       if (selectingApplet()) {
4          return;
5       }
6
7       byte[] buf = apdu.getBuffer();
8       switch (buf[ISO7816.OFFSET_INS]) {
9       case (byte) 0x00: testOverhead(apdu);
10         break;
11      case (byte) 0x01: testCase1(apdu);
12         break;
13      case (byte) 0x02: testCase2(apdu);
14         break;
15      case (byte) 0x03: testCase3(apdu);
16         break;
17      case (byte) 0x04: testCase4(apdu);
18      break;
19      case (byte) 0x05: testCase5(apdu);
20         break;
21      case (byte) 0x06: testCase6(apdu);
22         break;
23      case (byte) 0x07: testCase7(apdu);
24         break;
25     default:
26         ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
27      }
28   }
```

Listing 5.5: Application to analyze the performance of sample applications

As mentioned before, desired operations are executed several times within the sample application. This is achieved by executing the operation within a loop, whose loopcount is passed to the application via a parameter in the APDU. Thus, the sample application is stated in Listing 5.6 and will be explained further in the following.

```
1 public void testOverhead(APDU apdu) {
2       byte[] buffer = apdu.getBuffer();
3
4       short p1 = Util.getShort(buffer, ISO7816.OFFSET_CDATA);
5       short p2 = Util.getShort(buffer, (short)(ISO7816.OFFSET_CDATA+2));
6
7       setSection(1);
8       while(p1 > 0)
9       {
10         // OPERATION TO ANALYZE
11         p1--;
12      }
13      setSection(0);
14
15      apdu.setOutgoingAndSend((short)0, (short)2);
16   }
```

Listing 5.6: Sample application with main execution loop

First, the APDU buffer is read out. After that, the two parameters (two bytes for each) given in the APDU are being read out to get the loopcount, which is stored in the first two bytes. The *setSection(1)* call in line 7 causes the underlying hardware simulation to start tracking processor cycles and associating them with section number 1. Now the actual execution loop is being performed; it contains the desired operation which shall be analyzed. Thus, the operation is executed exactly as often as specified in the APDU. After the main loop, *setSection(0)* stops the cycle tracking mechanism again to prevent tracking of following commands.

The example shown in Listing 5.6 shows an empty sample application, which is used for determining the overhead which is caused by:

- The main execution loop in case of simulation on a hardware model. Loop condition checking and variable decrementation needs processing cycles as well.
- The overall processing time in case of time measurement. The total time from starting the processing on the card until sending the response back is tracked.

This overhead can either be cleared out by hand, or it has to be considered in the sample application block model accordingly. It is considered in the application block model in this case to make the process of block model creation more straightforward.

Sample applications need to be compiled for Java Cards. After that, the generated CAP-file is uploaded and installed on the Java Card simulation (or real Java Card). It turned out, in case of simulation, the uploading and install step consumes much more time than the simulation of sample application itself. So, the implementation of sample applications should be well-considered, cause every change requires a new compilation and upload/install step on the simulation, which takes time.

When the applet is prepared on the simulation, single sample applications are executed by sending the appropriate APDU to the simulation. For example, for executing the sample

application to measure overhead, following APDU, described in Table 5.2 needs to be sent.

CLA	INS	P1	P2	Lc	Data
80	00	00	00	04	00FF00FF

Table 5.2: APDU for measuring overhead, first two bytes describe the loopcount

This command causes the Java Card to execute the sample application specified by the INS byte. The first two bytes in the data segment contain the loopcount for execution as shown in Listing 5.6.

5.2.4 Deducing performance for single blocks

A sufficient number of sample applications need to be simulated and analyzed to find out each block's performance. This means, at least one performance measurement has to be executed per block. In the easiest case, of course, each sample applications would just contain exactly one block so the timing information could be read out directly from the simulation time. But it is not always possible to execute just one designated block because of the following reasons:

- Calculation operations always contain an assignment block
- Loops always contain conditions (which can be more or less complex)
- Loops contain increment or decrement operations
- Branches have conditions

So one sample application always contains a combination out of more blocks. To find out the time for exactly one block, some more information about that block is needed. This is done by having another sample application which contains the same blocks, but in a different combination. In a mathematical view, this can be seen as a linear system of equations. So the timing for each sample application can be written as:

$$T_1 = t_1 c_{11} + t_2 c_{12} + ... + t_n c_{1n} \tag{5.1}$$

where t_i is the execution time for block i and c_{ij} is the execution count of block j in sample application i. T_i stands for the entire execution time of sample program i. This can also be written as

$$T_1 = \sum_{n=0}^{N} t_n c_{1n} \tag{5.2}$$

Since our goal is to find the values for t_i more equations of this form are necessary which results in a linear system of equations. To get a convenient representation, a matrix

view on the topic is used. Therefore t and T can be seen as column vectors, C is the matrix for the linear coefficients.

$$\begin{pmatrix} c_{1,1} & c_{1,2} & ... & c_{1,n} \\ c_{2,1} & c_{2,2} & ... & c_{2,n} \\ .. & .. & .. & .. \\ c_{m,1} & c_{m,2} & .. & c_{m,n} \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \\ .. \\ t_n \end{pmatrix} = \begin{pmatrix} T_1 \\ T_2 \\ ... \\ T_m \end{pmatrix} \quad (5.3)$$

which can be rewritten as the well known form for a linear system of equations

$$Ct = T \quad (5.4)$$

Since we want to know the execution time of each single block (which is vector t) this linear system needs to be solved. This is done by calculating the inverse of C and combining it with the result vector T.

$$t = C^{-1}T \quad (5.5)$$

This method to solve a system of equations is an exact way. Of course, there will be situations, where single equations are not linear independent. That means further, that the matrix C gets singular and can not be inverted any more. Another point is, if the number of test cases or sample applications is higher than the number of possible blocks, the system is overdefined and the matrix C is not square any more. In both cases, no unique solution of the system is existing. That's why a statistical estimator, called least square method is used for solving the linear system. It uses the pseudo-inverse matrix of C for the calculation of t, which gives the following result:

$$t = (C^T \times C)^{-1} \times C^T \times T \quad (5.6)$$

This gives a final result for the vector t which provides execution times for all generic blocks. Those execution times are further used in the estimation process for unknown applications.

5.2.5 Deducing TimedBlockDB

The two steps of analyzing a sample application's performance and to find its sample application block model are the prerequisites for determining a TimedBlockDB for the desired platform. It has to be ensured that a sufficient number of sample applications is existing to be able to calculate the consumed time for every single block, so at least one sample application per block has to be found.

Therefore a C# application has been developed, which analyzes timing log file and sample application block model for all sample cases. The timing logfiles provide runtime information about the sample application whereas the block model shows the corresponding blocks which are causing that runtime. Block model and timing logfiles are joined based on their underlying section, which is used in the sample application and finally shown in the timing log file. Therefore, timing log file, section id and corresponding XML

block representation have to be provided. The routine searches the timing log file for the certain section and extracts timing for the specified application. Moreover, it analyzes the sample application block model to find out the total execution count of each block. Since the sample application block model is a hierarchical structure of blocks, it needs to be parsed hierarchically as well. Since sample applications are rather simple examples, the step for parsing the block model is not described in further detail here, this is done in Section 5.3.1 when the estimation process itself is concerned. Still, the output of the parser is a structure which contains an execution count for each block in the sample application block model.

A C# application is responsible for joining together the execution count for each block and the overall timing. To establish a link to Section 5.2.4, this means the values for c and T in Equation 5.1 are provided. With a sufficient number of samples, the matrix C and the vector T according to Section 5.2.4 can be set up. The execution count for all possible blocks form one row in the matrix C whereas the time value (the overall execution time for that sample application) is stored in the corresponding row of vector T.

With every sample application, the matrix C and the vector T get one more row. This needs to be repeated at least until the matrix C is square, in order to be able to get a unique timing value for each single block. To be more exact, not even a square matrix can guarantee a unique solution for the linear system of equations. If the execution counts for blocks are linear dependent in some sample applications, C gets singular, and the system is underdefined. In such a case, more sample application have to be developed to make the system solvable. Another special case can occur if the system gets overdefined. This happens, if one and the same block has different execution times in every execution. In this case, the least square algorithm calculates a mean value for the certain block.

This analysis and matrix building step is done in a C# application, whereas the calculation as mentioned in Equation 5.6 is done in Matlab, as shown in Figure 5.4.

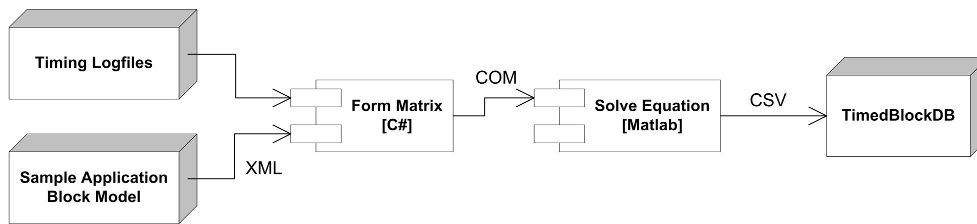

Figure 5.4: Deducing TimedBlockDB

Thus, the Matlab script needs to be provided with the matrix C and the result vector T. Additionally, also a list of GenericBlocks is passed to the Matlab script to make it possible to join the single blocks according to their names. With this data, Matlab can solve the system of equations according to the least-square algorithm described in Equation 5.6. The generated output is a *CSV*-file, which contains all blocks defined by the generic block model (taken from the DTD) provided with timing information for each block. This file is further used by the *performance estimator* to evaluate unknown applications.

Interaction between C# and Matlab

The interaction between C#, or in general between the .net layer and Matlab is done over a component object model (COM) interface called Matlab Automation Service. COM is a feature of the operating system Windows which allows communication between different kinds of software developing platforms. It also enables the access to objects across different programming languages. The communication over COM follows a client/server principle and uses generally defined interfaces. These interfaces are abstract classes which have to be implemented by the target platform. A principal overview of the communication between the setup application and Matlab can be seen in Figure 5.5. The C# application contains a COM client which communicates with the COM server. The server passes all requests further to the Matlab application where the calculation is executed. Results of the calculation are passed back to the calling application through those layers finally. [Mat11]

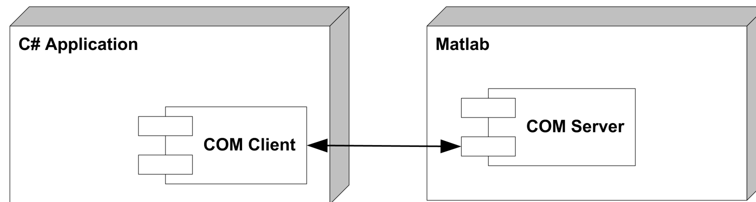

Figure 5.5: Communication between C# and Matlab

5.3 Estimation of native applications

After having performance characteristics for each block, this information is used to evaluate new applications. Therefore the desired native application itself has to be modeled with GenericBlocks. Since an application block model is represented as XML-file, the modeling can be done with any reasonable XML-editor. Preferably a graphical editor is used to generate the application block model with respect to the DTD. The editor then takes care of the correct structure of the block model. This block model of a new application and the TimedBlockDB from the previous step are the input for the estimation process. The outcome is a performance estimation for the desired application. The general flow can be seen in Figure 5.6.

In following Figure 5.7 the implementation of the performance estimator in detail is described; it contains following steps:

- Parsing of the application block model XML file. The structure of the application block model is transferred to an object model and the XML file is analyzed in order to obtain the execution count for every block.
- Parsing of the TimedBlockDB.
- Combining application block model and TimedBlockDB in order to calculate the estimated execution time.

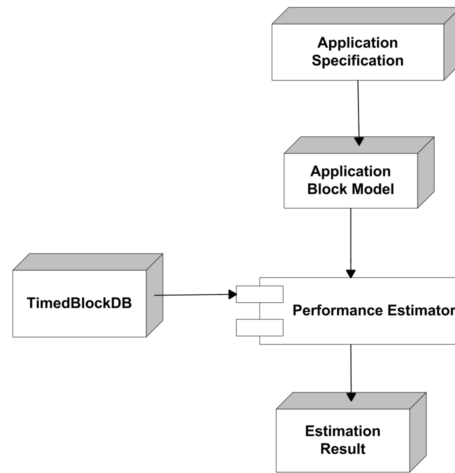

Figure 5.6: Operational flow for estimating an application

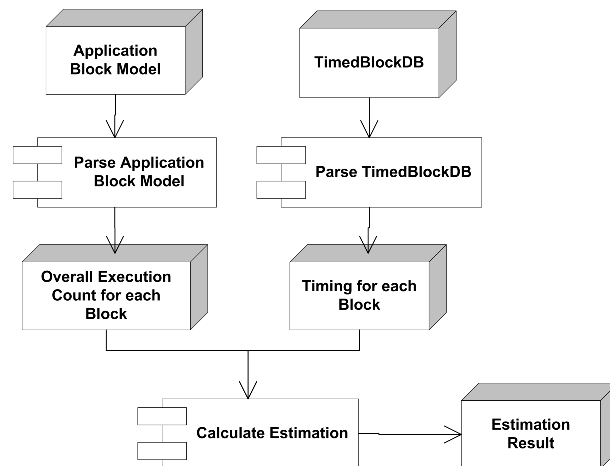

Figure 5.7: Detailed view of the single steps of the Performance Estimator

5.3.1 Parsing of application block model

The goal of this step is to analyze the XML description and calculate an execution count for every single block used in the application. Respect needs to put especially to special blocks for control structures like branches and loops, since they influence the execution count of other blocks.
So, the parser must distinguish between those „standard“ blocks and blocks which require a special treatment during the estimation process. To perform this in an object-oriented way, an interpreter pattern ([GHJV94]) is used, which is shown in a general form in Figure 5.8. It describes a language as a sequence of terminal and non-terminal expressions. Non-terminal expressions can also contain any other expressions theirselves. The context stands for an actual instance of the model.

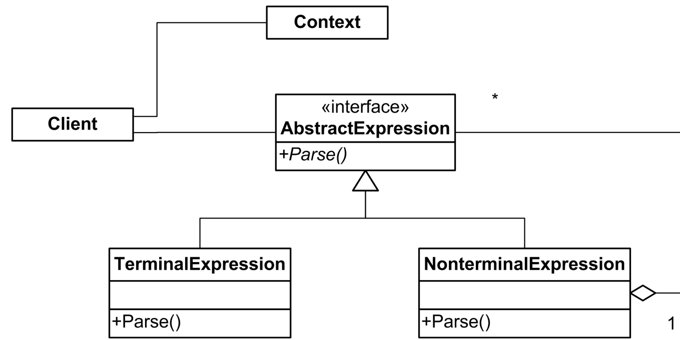

Figure 5.8: General view of interpreter pattern

This general structure in Figure 5.8 is similar to the XML representation of the block model. Every dedicated block has a corresponding class which implements the interface *GenericBlock*. It contains a *Parse* method, which describes, how the specific block will be parsed. While the GenericBlock hierarchy describes the grammar of the block model, data itself is stored in the context. So the context is a sequence of instances of those GenericBlocks. For this special case, the class diagram looks like the following (see Figure 5.9).

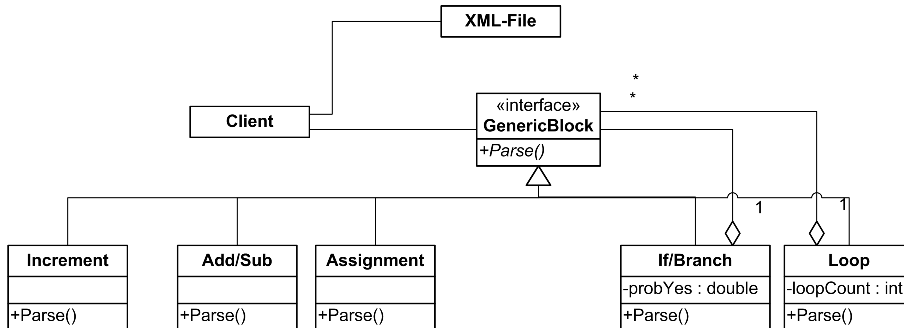

Figure 5.9: Class diagram of the Parser

It shows, that those nonterminal-blocks have to contain further information about their execution behavior. *Loopcount* gives information how often this loop is being executed while *probYes* indicates the probability for executing the „yes“ path of the branch.

The *Parse* method of each *GenericBlock* tells which blocks are executed how often in this block. This is trivial for blocks which do not contain any other blocks since just one execution for itself is counted. A more difficult case are nonterminal-blocks, which can contain any other sequence of blocks theirselves. Therefore it is necessary to investigate all its sub blocks for their execution count. This procedure needs to be continued recursively until the lowest level of hierarchy.

As a data structure to track the execution count for all possible blocks, a *Hashtable* is used. A Hashtable stores key/value pairs, where the key is the block and the value is the execution count which can be incremented according to the contained blocks. This Hashtable is returned from the *Parse* method and contains the execution count which is generated by all subordinated blocks. So the caller has to do an element-wise addition and gets the overall execution count for all blocks in the finally returned Hashtable.

5.3.2 Parsing TimedBlockDB

The TimedBlockDB is represented as a CSV-file which contains the results from the setup step. Every block is provided with the number of cycles its execution needs. In the estimation application this CSV file is transferred into an object model as stated in Figure 5.10.

A TimedBlockDB contains a list of TimedBlocks and the name of the platform it was made for. It further contains a method which transforms a given comma separated values (CSV) file into this object model. Every row in the CSV file is transferred into a TimedBlock object. Name and time in the TimedBlock are set accordingly and get added to the list in TimedBlockDB. In detail, TimedBlocks are stored in a *Hashtable*, one of the C# collection classes. Blockname serves as the key whereas the value is the TimedBlock object itself. The TimedBlockDB provides an object model for further use in the performance estimation application.

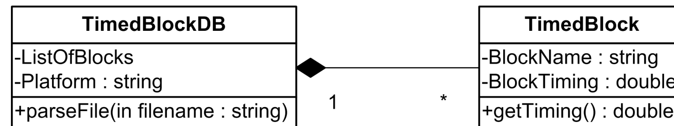

Figure 5.10: Class diagram of TimedBlockDB

5.3.3 Combining application block model and TimedBlockDB

After having an application block model and TimedBlockDB loaded, those two need to be combined in order to get the total execution time estimation for the application. The application block model parser gives a list of all blocks and their execution count while the TimedBlockDB parser gives a list of all blocks with their execution time. So a multiplication of execution count and timing gives the execution time for one specific block. This step is done for each block, the result is summed up to get the overall execution count for the application, which is the result of the performance estimation.

5.4 Extending the estimation process with new GenericBlocks

In case, the amount of existing blocks is not sufficient to model a certain application, new blocks have to be defined. The GenericBlockDB is defined in the DTD file, so this is the first point to add new blocks. New blocks have to be integrated in the structure of DTD, so it is possible to use them on the right place later on.

After being integrated in the DTD, sample applications which represent that blocks in a good way need to be found. Their runtime needs to be analyzed according to Section 5.2.1 and the application needs to be modeled as described in Section 5.2.2. At this point, either all previously generated sample applications can also be regarded in the TimedBlockDB deduction according to Section 5.2.5 or only the newly defined blocks can be put into the deduction system. In the first case, TimedBlockDB contains all blocks right away. Whereas in the second case, TimedBlockDB only contains timing information for the freshly analyzed blocks. Thus, to get a complete TimedBlockDB, the newly created has to be merged with the previously created one by inserting new rows.

Chapter 6

Results

This section describes how the previously found knowledge can be used in practical examples. It first shows how the process for finding the TimedBlockDB is applied in a concrete case starting from defining blocks. Further, the results of simulation and block model creation are illustrated. Finally, the determination of the TimedBlockDB by solving the linear system of equation is stated.

It also verifies the estimation process by comparing the results to reference implementations. Therefore native developed applications have been analyzed and modeled with the above described block model. Also, the native application has been implemented as Java Card applet and the performance was measured.

6.1 Setup of estimation rules (TimedBlockDB)

In this section the setup of estimation rules (which results in a complete TimedBlockDB) is shown in a concrete example. It starts with identification of the needed blocks, after that, sample applications are created to analyze the performance behavior of that blocks.

6.1.1 Identifying blocks

The finding of estimation rules generally starts with creation of the needed block structure. This structure is principally defined by elements found in a pseudo programming language. It needs to contain all blocks which are necessary to implement a feasible program flow like loops and branches. Further, elementary arithmetic operations on variables like additions, subtraction, multiplication and division are needed in the model. Finally, some more complex functions are added to the model as well, since they are implemented on a lower level and can not be described with previously defined elements. Examples for those advanced functions are cryptographic operations, random-number related functions and also the access to arrays. The following overview in Table 6.1 shows the chosen blocks and their description. In this example, only the most important blocks are shown. In practical work, some more have been analyzed but those are not necessary to get a principal understanding of the process.

Block name	Example source	Description
Add	a+b;	Performs an addition of two variables. As shown, the operation alone is not executable, so there will always be an assignment as well.
Sub	a-b;	Performs a subtraction of two variables.
Mul	a*b;	Performs a multiplication. A multiplication also needs to be probed in combination with an assignment.
Div	a/b;	Performs a divide operation on two variables. It also needs to be probed in combination with an assignment.
Increment	i++;	Increments a variable by 1.
Decrement	i– –;	Decrements a variable by 1. Java resolves a decrementation to a subtraction and an assignment, so it can be seen as a combination of Sub and Assignment block.
Shift	a<<8;	Shifts a variable according to the given value. Operation is also just measurable in combination with an assignment.
Assignment	a=b;	Assigns a value to a variable. This block contains reading out variable b as well as assigning its value to variable a. An assignment always needs reading out a either variable or a constant; their runtime difference is not significant.
Loop	while(i<0)	Performs the contained operations as long as the condition is true. Since the variable needs to be altered within the loop body, side operations are compulsory to execute a loop in a proper way. That makes it impossible to measure a loop's performance independently.
If	if (i==0)	Performs a conditional branch. The contained operations are just executed if the condition is true. Of course, the condition can contain several parts which are linked logically. Internally, every part of the condition is executed as a conditional jump. If the condition contains more than one part, the number of jumps increases accordingly. Also the condition can contain any other type of block.

Table 6.1: A subset of possible blocks used for creation of TimedBlockDB

Block name	**Example source**	**Description**
ArrayAccess	array[1]=10; a=array[1];	Performs a read or write access to an array. Internally, a lot of boundary checks and memory dereferenciation needs to be performed.
AesInit	aesCipher.init(key, mode);	Initializes a cryptographic object to perform AES encryptions. The initialization is detached from the actual cryptographic operation, since those initializations are often done in a separate step. Further, the cryptographic operation can be performed more often after the initialization.
AesEncrypt	aesCipher.encrypt(data);	Encrypts given data with the previously given key from the initialization step.
AesDecrypt	aesCipher.decrypt(data);	Decrypts data with the previously given key from the initialization step.
GenerateRandom	generateRandomData(...)	Generates a random number.
SetShort	Util.SetShort(...)	Deposits the short value as two successive bytes at the specified offset in the byte array.

Table 6.2: Advanced Java Card specific composite blocks, which are implemented on a low level. Those are analyzed as a whole and are not split up in their low level components

During the practical part it turned out that the blocks mentioned in Table 6.1 are necessary to model basic Java Card applications. Basic means in this context that the program flow needs to have a procedural character, so no object oriented programming is supported by this model. To enable object oriented modeling, one needs to think about how classes and methods could be integrated into such a model. Further, the composite blocks in Table 6.2 are necessary to model complex behavior like cryptographic operations and other operations which are implemented on a lower level.

For a later matrix representation, blocks are taken in the order they appear in Table 6.1, so the matrix columns will look like following:

$$(Add\ Sub\ Mul\ Div\ Increment\ Loop\ Shift\ Assignment\ Decrement\ If) \tag{6.1}$$

6.1.2 Sample applications

In the following, a subset of created sample applications is displayed. First, their source code representation and second the associated application block model are shown. All following concrete sample applications use 255 as loop count. As a consequence, the sample application block model needs to be created with respect to that loop count. For simplicity, just the execution loop is shown, all other parts of the sample application (as stated in Section 5.2.3) are discarded.

Sample application for analyzing overhead

A special role is played by processing overhead. This constant value is used for clearing out the overhead, which is generated from start of the processing on the card until getting to the point, where performance measurement shall start (the operations of interest). Measuring cycles with a hardware simulation does not need that step, since measurement starts and stops at the desired point. But for time measurement with an oscilloscope and for an actual estimation, this time needs to be considered. Of course it is possible to clear this value out per hand, but since it has to be added later on anyway if one wants to know the overall execution time of the application, the decision has been taken for including it in the process.

```
1    while(p1 > 0)
2      {
3        //Here comes the operation to measure
4        p1--;
5      }
```

Listing 6.1: Sample application to measure overhead

```
1 <application name="sample overhead">
2   <loop loopcount="255">
3     <Decrement/>
4   </loop>
5 </application>
```

Listing 6.2: Sample application block model to measure overhead

The sample application block model is parsed as described in Section 5.3.1, which gives an execution count for each block. For this sample application, the *Loop* block is executed 256 times (because the condition is checked once more before the loop ends) and the *Decrement* block is executed 255 times. So the matrix row C_0 contains 255 at the corresponding positions. After simulation, the analysis of timing logfiles for the specified section results in a runtime of 82875 cycles. This value is the first row of vector T.

$$C_0 = (0\ 0\ 0\ 0\ 0\ 256\ 0\ 0\ 255\ 0)$$

$$T_0 = (105998)$$

Sample application for analyzing an *If* and *Assignment* block

```
1    while(p1 > 0)
2        {
3            if (p1>127)
4                p2=p1;
5            p1--;
6        }
```

Listing 6.3: Sample application to analyze If and Assignment

```
1 <application name="sample_if">
2   <Loop loopcount="255">
3     <If probYes="0,5">
4       <Condition ConditionPartsNum="1"/>
5      <yes>
6       <Assignment/>
7      </yes>
8     </If>
9     <Decrement/>
10  </Loop>
11 </application>
```

Listing 6.4: Sample application block model to analyze If and Assignment

In this sample application, the *If* block itself is executed 255 times, while the condition is true in only 50%. So the *Assignment* block is executed only 128 times. The simulation took 150961 cycles. Thus, the corresponding row in matrix C and vector T is stated as follows:

$$C_1 = (0\ 0\ 0\ 0\ 0\ 256\ 0\ 128\ 255\ 255)$$

$$T_1 = (174084)$$

These two sample applications describe performance behavior of the Loop-, SubDecrement-, Assignment- and If-block. At this point it is not possible to calculate the timing for any block. Therefore more sample applications are needed that describe these blocks further.

Sample application for analyzing an *Assignment* block

```
1 <application name="sample␣assignment">
2   <Loop loopcount="255">
3         <Assignment/>
4     <Decrement/>
5   </Loop>
6 </application>
```

Listing 6.5: Sample application block model to analyze an Assignment block

This sample has one additional Assignment block compared to the blocks mentioned in the overhead section. Timing analysis for this sample showed 105594 cycles, so the matrix row and vector element are as follows:

$$C_2 = (0\ 0\ 0\ 0\ 0\ 256\ 0\ 255\ 255\ 0)$$

$$T_2 = (128717)$$

Sample application for analyzing an *Add* block

As mentioned before, it is not possible to analyze arithmetic operations on their own, because their execution just makes sense if the result is assigned to another variable. So the execution of an Add operation, always contains an Assignment as well. So, the sample application is designed as follows:

```
1 <application name="sample␣add␣assignment">
2   <Loop loopcount="255">
3     <Add/>
4       <Assignment/>
5     <Decrement/>
6   </Loop>
7 </application>
```

Listing 6.6: Sample application block model to analyze Add

With an execution time of 151388 cycles, this sample is presented in following mathematical form:

$$C_3 = (255\ 0\ 0\ 0\ 0\ 256\ 0\ 255\ 255\ 0)$$

$$T_3 = (128717)$$

Sample application for analyzing a *Decrement* block

The Decrement block is already contained in the sample application for measuring the overhead since for executing a loop some kind of counting is necessary. To analyze the Decrement block itself, a second Decrement block needs to be inserted into the main loop. This sample application together with the sample application for measuring overhead allows finally to get timing for both, the *Loop* block and the the *Decrement* block.

```
1 <application name="sample_add_assignment">
2   <Loop loopcount="255">
3       <Decrement/>
4     <Decrement/>
5   </Loop>
6 </application>
```

Listing 6.7: Sample application block model to analyze Decrement

This results in executing the Decrement block 510 times in the sample application. With a total execution time of 153867, it leads to following mathematical representation:

$$C_3 = (0\ 0\ 0\ 0\ 0\ 256\ 0\ 0\ 510\ 0)$$

$$T_3 = (153867)$$

When finding these sample applications, it is really important that their execution count of the different blocks is linear independent. This is fulfilled by the found sample applications during the practical work of this thesis.

The presented sample applications give an overview, how those can be designed to find timing information for each block. Further sample applications are not presented here, since they only differ in the operation, which is executed in the loop.

6.1.3 Joining sample applications and deducing TimedBlockDB

The procedure of creating sample applications is repeated until every block has been described in at least one sample application. All these sample applications are put into the setup application shown in Figure 6.1. The left side allows to choose a sample application block model, which is analyzed concerning the execution count of each block. Further, the count of consumed cycles for this sample application is provided. The *Add* button adds the loaded sample application to the set of sample applications shown on the right hand side. This step is repeated until a sufficient number of sample applications is achieved to be able to figure out the execution cycles of each block.

For each sample application in that set, a row in matrix C containing the execution counts of each block is added. Further, the number of execution cycles (or the time needed) is added to vector T. All these steps are done by the application, when starting the calculation. The matrix representation for an example set of sample applications is

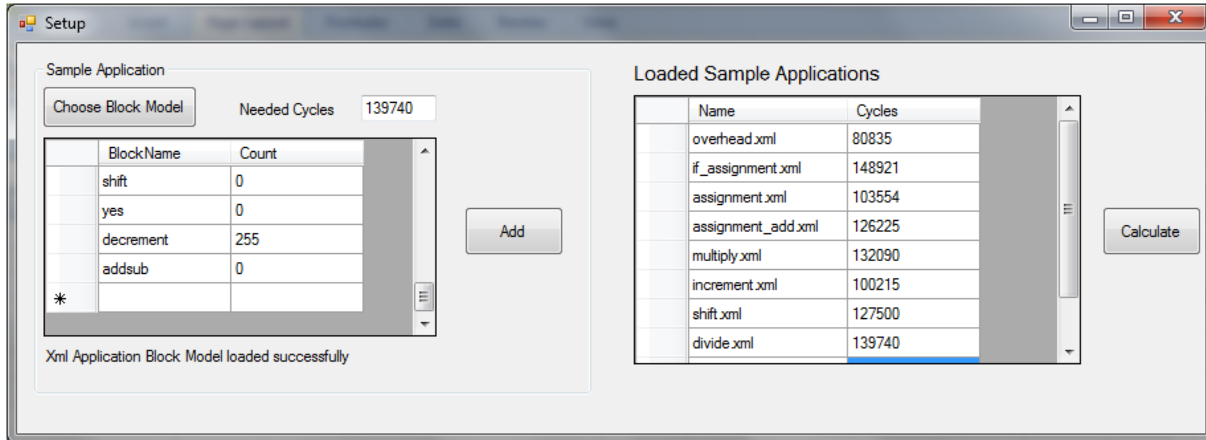

Figure 6.1: Application to calculate estimation rules

shown in Equation 6.2.

$$C = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 256 & 0 & 0 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 & 256 & 0 & 128 & 255 & 255 \\ 0 & 0 & 0 & 0 & 0 & 256 & 0 & 255 & 255 & 0 \\ 255 & 0 & 0 & 0 & 0 & 256 & 0 & 255 & 255 & 0 \\ 0 & 0 & 255 & 0 & 0 & 256 & 0 & 255 & 255 & 0 \\ 0 & 0 & 0 & 0 & 255 & 256 & 0 & 0 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 & 256 & 255 & 255 & 255 & 0 \\ 0 & 0 & 0 & 255 & 0 & 256 & 0 & 255 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 & 256 & 0 & 0 & 510 & 0 \\ 0 & 255 & 0 & 0 & 0 & 256 & 0 & 255 & 255 & 0 \end{pmatrix} \tag{6.2}$$

The matrix C represents a summary of how often every block is executed in different sample applications. The columns contain the different blocks according to Equation 6.1, while every row stands for one single sample application. For example, the first row contains the sample application for the overhead. The *Loop* block is executed 256 times and the *Decrement* is executed 255 times as already mentioned in the description of the first sample application. In the second row, an *Assignment* inside an *If* block are executed. Here, the *If* block got a condition-true-probability of 50 %, which means its body (the *Assignment* block) is executed only 128 times. The third row already allows to calculate the time for one *Assignment* block. With every row, the system gets more information about the timing behavior of every block.

The vector T contains the overall execution time for each sample application, the rows

are related to the rows in the matrix C.

$$T = \begin{pmatrix} 105998 \\ 174084 \\ 128717 \\ 151388 \\ 157253 \\ 125378 \\ 152663 \\ 164903 \\ 153867 \\ 153448 \end{pmatrix} \tag{6.3}$$

To get the runtime information for each block, vector t is calculated by using the least square algorithm according to Equation 5.6. Therefore the C# application calls a Matlab procedure, which solves this equation.

$$t = (C^T \times C)^{-1} \times C^T \times T = \begin{pmatrix} 88.90 \\ 96.98 \\ 111.90 \\ 141.90 \\ 76.00 \\ 227.06 \\ 93.90 \\ 89.09 \\ 187.72 \\ 222.28 \end{pmatrix} \tag{6.4}$$

The Matlab application creates a CSV file which contains performance information for all analyzed blocks. Therefore it needs to join the blocknames with the corresponding rows in vector t. The found timing for all analyzed blocks is shown in Table 6.3.

Block name	**Timing [cycles]**	**Comment**
Add	88.90	
Sub	96.98	
Mul	111.90	
Div	141.90	
Increment	76.00	
Loop	227.06	These cycles are needed for one loop execution, starting from checking the condition. If a loop is executed several times, the cycles need to be multiplied with the loopcount.
Shift	93.90	
Assignment	89.09	
Decrement	187.72	Obviously, a decrement operation is resolved as subtraction and assignment, since the sum of those two blocks conforms to one decrementation.
If	222.28	

Table 6.3: Result of block analysis: blocks and their timing in cycles

6.2 Estimation of a security module

Such a module serves as a secure element to encrypt the traffic between a card reader and a smart card. It contains a key store to use different keys via parameterization. That allows to encapsulate security relevant parts into that module and the reader does not need to contain that critical information. This technique allows to exchange the secure element without actually changing the whole reader.

To verify the found estimation algorithm, the secure module was implemented on Java Cards. Further, a sample transaction was defined to profile the performance on both smart cards, the native implemented version and the Java Card.

There are three use cases which have been analyzed. Each of those consists of two parts: In the first part the security module is in charge of encrypting/signing the data which should be transmitted to the card. The second part deals with the answer of the smart card, it checks if an authentication was successful or if the received signature is correct.

The sample transaction consists of:

- Authentication: The process that authenticates smart card and reader in an encrypted way. After initiating the communication an encrypted message which contains a random number is received from the smart card. The secure element needs to decrypt the message, generate its own random number, perform an encryption and send this message to the smart card. The smart card itself checks if everything is alright and sends another encrypted message back to the secure element. The secure element itself performs some checks, if everything is alright with the authentication, a session key for further use in read and write operations is generated.

- Read Data: Reads data on the smart card. The transmitted data is verified by a message authentication code (MAC). The MAC is calculated on the sending device and verified by the receiver. Since transmission is done in both directions, the security module has to be able to sign and verify the sent/received data. In the first part, it signs the command which is sent to the smart card, in the second part it verifies the gotten data from the smart card.

- Write Data: Write data on the smart card. Here, the transmission is also verified by a MAC. In the first part, it signs the message which is sent to the smart card (which contains command and data to write), in the second part it verifies the gotten answer from the smart card.

Table 6.4 shows the result of the estimation compared to the actual execution time. Every use case with its sub parts is listed there and the relative deviation of the estimation is pointed out.

As seen in Table 6.4 the estimation error lies between 1,4 and 10,0 percent. The reasons for this are found in implementation details. Different levels of optimization can not be considered in an application block model. Further, crypto operations' runtime is depending a lot on their input data, especially their input length. This is the reason for slight differences in every execution. To clear this quite big parts out of the overall execution

Operation	Cycles needed	Cycles estimated	Error
Authenticate Init	150683	135632	10,0%
Authenticate Verify	193001	178981	7,3%
Read Sign	111941	122189	9,2%
Read Verify	148075	146002	1,4%
Write Sign	143056	148765	4,0%
Write Verify	119661	112612	5,9%

Table 6.4: Results for estimation of a security module

time, test cases without crypto operations have been created.

In the example above crypto operations consume 65 % up to 80 % of the overall execution time of the application. To get an overview, the execution of this application is analyzed concerning different operation classes.
Following operation classes are concerned:

- Cryptographic operations
- Random Number generator
- Array access
- Processing overhead

The diagram in Figure 6.2 shows the distribution of execution time on above mentioned operation classes. It shows, that crypto operations take more than 60 % of the overall execution time. Therefore, the previously presented results are strongly dependent on how accurate the cryptographic operations are estimated in the process. To clear out this fact, testcases were created without executing cryptographic operations.

So, the verification process is repeated with the same tasks as before, but cryptographic operations are not considered. Those operations are deleted from the application block model on one side, whereas they also are striked out from the verification application on the Java Card. Following Table 6.5 shows the estimated and measured cycles for previously discussed tasks. Here, the error tendencially goes down since crypto operations can not be estimated so accurately because their runtime is dependent on the processed data length. Further, cryptographic operations contain some EEPROM write accesses. Empirically, it turned out that those need some milliseconds, and have quite a big variance, which results in less accuracy of the estimation.

6.3 Summary

This chapter showed the results found during this diploma thesis. First the found blocks which are necessary to model Java Card applications are presented. Here it turned out

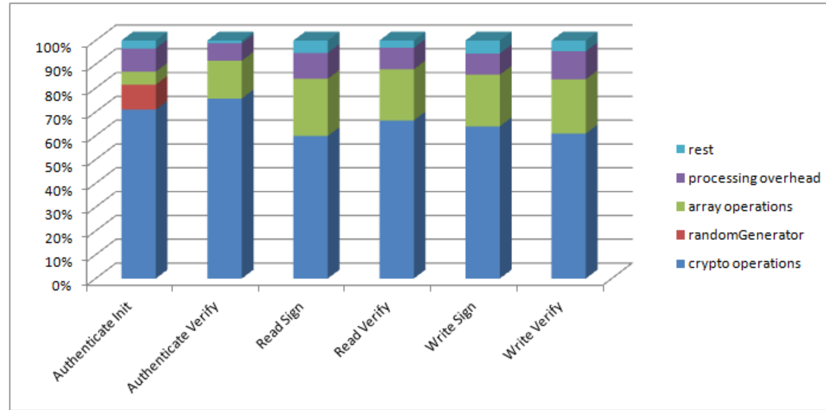

Figure 6.2: Overview, how operations are distributed on above mentioned operation classes in different test cases

Operation without cryptography	Cycles needed	Cycles estimated	Error
Authentication Init	44094	43928	0,4%
Authentication Verify	42067	44303	5,3%
Read Sign	45594	49017	7,5%
Read Verify	47961	50113	4,5%
Write Sign	51968	53644	3,2%
Write Verify	47237	45558	3,6%

Table 6.5: Results for estimation of a security module, cryptographic operations are discarded

that it is possible to model basic program flows with 10 elementary blocks. Further, six composite blocks are necessary to model more complex applications, these contain for example cryptographic operations and random number generation. With these blocks, a block model was created. To be able to make accurate estimations with these blocks, each block needed to be provided with a timing. Thus, sample applications to get the timing for every single block were created. Since not every block could be executed as a single measurable operations, the set of sample applications had to be analyzed as a whole, because one sample application could contain more different blocks. The analysis of sample applications gave a timing for each block.

To analyze the accuracy of the found estimation algorithm, a real native application on a smart card was taken as an example. More detailed, a secure element which encrypts the traffic between a smart card and the corresponding smart card reader. Here, six different operations were considered, where it turned out that the estimation produced an error between 1.4 % and 10.0 %. Cryptographic operations took quite a big part of the execution time of every single operation (from 65 to 80 %). This made the entire estimation dependent of the accuracy of cryptographic operations. That's why another analysis was done without cryptographic operations, in that case the error could be lowered to a range from 0.4 to 7.5 %.

Chapter 7

Conclusion and Future Work

This master's thesis dealed with the estimation of the performance impact when porting native applications to Java Cards. There are great differences between applications implemented for Java Cards and those which are implemented in native programming languages. The first, and probably most important fact is the interopability of applications for Java Cards. Those are implemented as so called applets and can be loaded on every standard-compliant Java Card. This Java Card standard is defined in the Java Card platform specification by Oracle. Since a Java Card only contains the operating system and no application, it can be used as a universal platform for every kind of application. Further, changes in the application can be implemented without changing the entire smartcard. Hence, additional applets can be installed on demand as well.

All these features are enabled by the usage of an intermediate layer (the Java Card Virtual Machine), where all applications are executed on. However, this virtual machine causes the major drawback of such solutions, the execution performance of applications. The processing in the virtual machine consumes a lot of processing time and memory resources, so the execution time increases. This thesis considers the performance of both, native and Java Card applications. Further it shows a method to estimate the performance impact when re-implementing a native application on a Java Card. The challenge was, to find an estimation methodology for native applications, before actually implementing them in Java.

For a good estimation, it is not enough, only to measure the performance of the native application to conclude to the performance in case of an implementation on Java Cards. The performance impact strongly depends on the operations which are executed in the application. That's why the native application's source code needs to be analyzed. In order to find out the performance of a Java Card application that has the same behavior as a native application, one has to think how this behavior can be implemented on Java Cards. Thus, a mapping between native source code and Java needs to be found. Since those two languages are basically different concerning memory access and their structure, a mapping is not easy to apply. Therefore, generic blocks have been identified which represent generic elements of any programming language. Every block needs to be measured on the desired platform to get its execution time. For an estimation, the native application has to be modeled with those blocks to get a generic model of the application flow. This

generic model finally needs to be analyzed and combined with the execution time for each block to get the final estimation for the application.

The overall estimation is split into two parts. First, the creation of the generic block model including deducing execution time for each block is considered. Therefore, a set of sample applications is created and analyzed concerning the performance. The outcome is a fully specified block database, which contains performance information for each block. Secondly, an estimation of a native application itself is considered. Therefore the native application has to be modeled with the previously defined generic blocks. It is especially necessary to model control structures accordingly. The created application block model is then analyzed to find out the execution count for each block. Finally, the application's performance is estimated by combining the application block model to the block database.

During the thesis, a number of generic blocks was found, which allow to model common smartcard applications. These contain basic blocks like arithmetic and logical operations as well as complex blocks like cryptographic operations. For all these blocks, sample applications have been created to analyze their performance. Every sample application contained one or (in most cases) more generic blocks. Here, special attention had to be put to the fact that the number of different blocks in every sample application had to be linear independent, otherwise it would not have been possible to deduce timing for each block. These applications were executed on a SystemC model to get highly accurate timing information about the application. With timing for each sample application, and the information which block is executed how often in it, the time needed for execution of each block could be deduced.

Found methods are applied to practical examples, where only the native version had been existing. To verify these methods, native applications are first estimated and then re-implemented on a Java Card. In this thesis, a security module on a smartcard which is implemented in native programming languages was taken as a verification example. It consisted of some operations like authentication and read/write commands. The commands theirselves are performing basic operations as well as complex cryptographic operations to ensure security. So, every single command was estimated with the previously described technique, afterwards it had been reimplemented to run on Java Cards. The estimation was compared to the actual performance, the estimation error in these cases lies between 1.5 and 10 %.

7.1 Future Work

For future work, the usability of the found methods in case of changing system parameters could be analyzed. First of all, the influence of the block model layout on the estimation accuracy could be considered. It probably depends a lot, how blocks are designed. Big blocks, which contain a lot of operations are probably also dependent on the given input data, they are operating on. Estimation of small blocks, however, which contain simple operations is maybe more accurate, therefore the effort for creating application block mod-

els increases. A detailed analysis of these effects could be performed in some forthcoming work.

In this work, applications estimations for Java Cards are considered. As future tasks, estimations for other platforms could be executed as well, since the estimation system is designed to be easily adaptable to other platforms. Therefore a block database with execution times needs to be deduced for the desired platform. After that, any created application block model can be estimated for that platform as well. Such cases will show, how good the system works for other platforms, respectively where possible problems are located.

Appendix A

Glossaries

A.1 Acronyms

APDU application protocol data unit.

API application programming interface.

C-APDU command APDU.

CAD card access device.

COM component object model.

CPU central processing unit.

CSV comma separated values.

DTD document type definition.

EEPROM electrically erasable programmable read only memory.

GSM global system for mobile communication.

I/O input/output.

IEC International Electrotechnical Commission.

ISO International Organization for Standardization.

JCRE Java Card runtime environment.

JCVM Java Card virtual machine.

JIT just in time.

JVM Java virtual machine.

MAC message authentication code.

MMU memory management unit.

PC personal computer.

PC/SC personal computer / smartcard.

PIN personal identification number.

R-APDU response APDU.

RAM random access memory.

ROM read only memory.

SFR special function register.

UMTS universal mobile telecommunication system.

USB universal serial bus.

VM virtual machine.

WTX waiting time extension.

XML extended markup language.

A.2 Symbols

C Matrix which contains execution counts for each block of every sample application.

c_{ij} Execution count of block j in sample application i.

i Counting variable.

j Counting variable.

N Number of sample applications.

T_i Overall execution time for one sample application.

t_i Execution time for one specific block.

Bibliography

[BBE+99] Michael Baentsch, Peter Buhler, Thomas Eirich, Frank Hoering, and Marcus Oestreicher. JavaCard - From Hype to Reality. *IEEE Concurrency*, pages 36–43, 1999.

[Car11] Cardwerk. The ISO 7816 Smartcard Standard. `http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816.aspx`, October 2011.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[DDJ03] Gilles Grimaud Damien Deville, Antoine Galland and Sébastien Jean. Smart card operating systems: Past, present and future. Technical report, 2003.

[EMV11] EMVCO. The EMV Specification. `http://www.emvco.com`, October 2011.

[Erd04] Monika Erdmann. Benchmarking von Java Cards. Master's thesis, Ludwig-Maximilians-Universität München, 2004.

[Fis06] Mario Fischer. Vergleich von Java- und Native-Chipkarten. Technical report, Institut for Informatics, Ludwig-Maximilians University Munich, 2006.

[GASE] P. González-Aledo, L. Díaz Suarez, and P. Sanchez Espeso. Embedded software execution time estimation at different abstraction levels. Technical report.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.

[KKW+06] Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, Rainer Leupers, and Heinrich Meyr. A SW performance estimation framework for early SystemLevelDesign using finegrained instrumentation. Technical report, 2006.

[LY99] Tim Lindholm and Frank Yellin. *The JavaTM Virtual Machine Specification*. Addison-Wesley, second edition edition, 1999.

[Mat11] The Mathworks. Matlab automation service. `http://www.mathworks.nl/help/techdoc/matlab_external/brd0vd4-1.html`, October 2011.

[Mic11] Micropross. Micropross MP 300. `http://www.micropross.com/product.php?id=28`, October 2011.

[Ora11] Oracle. The Java Card Platform Specification 2.2.2. `http://java.sun.com/javacard/specs.html`, October 2011.

[Par06] Pierre Paradinas. Measuring the performance of the Java Card platform. Technical report, CNAM-Cedric, Paris, 2006.

[PCB07] Pierre Paradinas, Julien Cordry, and Samia Bouzefrane. Performance Evaluation of Java Card Bytecodes. In *WISTP'07*, pages 127–137, 2007.

[PCB09] Pierre Paradinas, Julien Cordry, and Samia Bouzefrane. MESURE Tool to benchmark Java Card platforms. volume 1, pages 49–57, 2009.

[Pöl11] Andreas Pöllabauer. Design and Implementation of a Smart Card Application Analysis. Master's thesis, Technische Universität Graz, 2011.

[Pro11] Micro Profiler. Micro profiler. `http://code.google.com/p/micro-profiler/`, October 2011.

[RE08] Wolfgang Rankl and Wolfgang Effing. *Handbuch der Chipkarten*, volume 5. Carl Hanser Verlag München Wien, 2008.

[Sch04] Michael Schmid. Applikationsspezifscher Prozessor zur Performance-Steigerung von Java Card Applikationen. Master's thesis, Technische Universität Graz, 2004.

[Sha02] Nik Shaylor. A Just-In-Time compiler for memory constrained low-power devices. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium, Berkeley*, 2002.

[SML07] Mark Stoodley, Kenneth Ma, and Marius Lut. Real-time Java, Part 2: Comparing compilation techniques. Technical report, 2007.

[Zha] Jianjun Zhao. Analyzing Control Flow in Java Bytecode. Technical report.