Massimiliano Zilli, Dott. Dott.Mag.

Hardware/Software Co-Design for Small-Footprint Java Cards

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht an der

Technischen Universität Graz

Betreuer

Em.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhold Weiß

Institut für Technische Informatik

Graz, März 2015

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.

Datum

Unterschrift

Kurzfassung

Der Markt von Smartcards zeigt ständiges Wachstum speziell in den drei Domänen Kommunikation, Identifikation und Banking. Typischerweise verfügen diese Produkte über einen 8 bis 16 Bit Prozessor, einige Kilobyte an RAM und mehrere Hundert Kilobyte an nicht flüchtigen Speicher (ROM, EEPROM und Flash-Speicher). Aufgrund dieser limitierten Hardwarekonfigurationen ist es gängige Praxis Applikationen in C oder Assembler zu schreiben, um eine möglichst hohe Leistung bei geringem Speicherverbrauch zu erzielen. Nachteil dieses Ansatzes ist, dass der Aufwand für die Portierung auf unterschiedliche Plattformen enorm ist. Java Card ist eine reduzierte Teilmenge der Java Umgebung, zugeschnitten auf Smartcards und löst dadurch das Problem der Portierbarkeit der Applikationen. Darüber hinaus unterstützt Java Card Objekt orientierte Programmierparadigmen und vereinfacht dadurch die Entwicklung von Applikationen.

In dieser Arbeit werden neuartige Komprimierungstechniken für die Komprimierung von Java Bytecode und eine neue Interpreter Architektur erforscht. Dictionary Compression für Java Cards wird analysiert und zwei erweiterte Verfahren basierend auf verallgemeinerten Dictionary Makros werden vorgestellt. Darüber hinaus wird ein Komprimierungsverfahren basierend auf Bytecode Instruction Folding eingeführt. Dies ermöglicht die Reduktion des benötigten Speicherbedarfs bei gleichzeitig erhöhter Geschwindigkeit der Ausführung der Applikationen. Die Kombination der Dictionary Compression und der Folding Compression für die Erstellung von neuen Komprimierungsverfahren wird evaluiert. Darüber hinaus wird eine neue Architektur für Java Interpreter entworfen, um die Ausführung von Applikationen zu beschleunigen. Die Architektur erlaubt es mit modernen Mitteln aus dem Hardware/Software – Co-Design Teile des Interpreters in die Hardware auszulagern. Verglichen mit dem klassischen softwarebasierten Java Interpreter, lassen sich dadurch bessere Performance Werte erzielen. Abschließend wird das Dictionary Decompression Modul auf der neuen Java Card Architektur implementiert, wodurch die Ausführung von komprimierten Applikationen ermöglicht wird. Der Dictionary Decompressor wird in zwei Varianten realisiert: Eine reine Softwarelösung und eine Hardware/Software – Co-Design Lösung für die neue Architektur.

Abstract

Smart cards have a continuously growing market, which is spread over three main groups: communications, identification, and banking. Typical hardware configurations have an 8 or 16 bit processor, several kilobytes of RAM and several hundred kilobytes of non-volatile memory (including ROM, EEPROM and flash memory). In such limited hardware configurations it is common practice to write the applications in C or assembly code in order to keep the memory footprint small and the execution performance high. The main drawback of this approach is the high effort needed to port the applications to different platforms. Java Card technology is a reduced set of the Java environment tailored to smart cards and solves the portability problem of the application thanks to its architecture based on the Java interpreter. Moreover, Java Card allows object oriented coding and eases the application development by third parties.

In this thesis novel compression techniques for the reduction of the Java Card application and a new Java interpreter architecture are explored. Dictionary compression for Java Card is analyzed and two extended dictionary compression techniques based on generalized dictionary macros are proposed. Moreover, a compression technique based on bytecode instruction folding is introduced. The latter allows space savings in the code memory and at the same time speeds up the execution of the compressed application. The combination of the dictionary compression with the folding compression for the creation of a new compression technique with considerable space savings is evaluated. In addition, to speed up the execution of the application a new architecture for the Java interpreter is designed. The new architecture permits the movement of parts of the interpretation into hardware by means of a hardware/software co-design. The new Java interpreter with hardware support achieves better runtime performances compared with the classic interpreter. Finally, the dictionary decompression module that allows the interpretation of applications compressed with dictionary techniques is implemented on the new Java Card architecture. The dictionary decompressor is realized in two variants: one in software and the other with a hardware support on the new architecture.

Acknowledgements

First of all, I would like to thank Prof. Reinhold Weiss for allowing me to work in the hardware/software co-design research group at the Institute for Technical Informatics. Especially, I am thankful for the invaluable comments and advice that he gave me during the review phases of this work.

I also want to thank the Austrian Federal Ministry of Transport, Innovation and Technology for the funded project called David, which is the backbone of my thesis. In this regard I would also like to thank Christian Steger not only for his successful supervision of the project but also for the helpful advice provided in all the phases of the project. A relevant role in the project was assumed also by the industrial partner NXP Semiconductors Austria GmbH, where I found the constructive help of Johannes Loinig and finally the kind support of Franz Krainer in the review phase. Special thanks go also to my project teammate Wolfgang Raschke and to my PhD colleagues and project assistants Michael Lackner, Reinhard Berlach, Andreas Sinhofer, Stefan Orehovec and Erik Gera-Fornvald. Other colleagues at the ITI that I need to mention for their support are Christian Kreiner, Norbert Drumml, Manuel Menghin, Christopher Perschen, Carlo Alberto Boano, Matteo Lasagni, Silvia Reiter and Engelbert Meissl.

Last but not least, I would like to express my gratitude to my family for their continuing support and encouragement in this experience. Beyond the support, I want to thank my lovely girlfriend Monica for her patience in these years.

Graz, March 2015 Massimiliano Zilli

Extended Abstract

Entering a restricted access building without a metal key, calling someone with a smartphone, paying at the supermarket without physical money are all activities that are possible thanks to smart cards. Smart cards are small embedded systems mainly used in the fields of telecommunications, banking, and identification. They consist of an integrated circuit embedded in a plastic support, which has components for processing, storing and transmitting data. Due to their diffusion, such systems have to be cheap and therefore have relatively limited resources. Usually they mount an 8 or 16 bit processor, several hundred kilobytes of persistent memory and several kilobytes of RAM.

The applications running on these small systems are often written in C and assembler to maximize the execution performance and to minimize the ROM size occupation. The drawback of this approach is the high effort needed to obtain a high level of portability from one platform to another. With the growth in the number and complexity of applications present on the smart card, the software architecture has evolved from simple libraries to small layered operating systems. The separation between operating system and application reduces the problem of the portability between one platform and another but still does not completely support the application development from third parties. The adoption of a software model based on an interpreter like Java solves the portability issue, and additionally offers a standardized platform over which third parties can develop their applications. To meet the resource constraints in smart cards Java Card, a subset of the Java language targeted for smart card applications, has been specified. In Java Card, an application written in Java language is compiled off-card to be installed on the smart card at a later stage.

The Java Card model intends the smart card to be a host for many applications. In this context, memory resources assume a relevant role in the overall cost of the smart card. Reducing the permanent memory needed for storing the applications or superfluous parts of the Java Card system results in a relevant decrease of the costs. The David project's [1] intent is to propose and evaluate new techniques for low-end smart cards enabled with Java Card. The topics of interest of the David project can be grouped into two main branches:

- The development of optimization techniques for reducing and enhancing the use of hardware resources
- The development of a design flow for tailoring different variants of the Java Card operating system to different product lines

[1] *DAVID - Design-Flow für Java Betriebssysteme auf Low-End Smart Cards, collaborative research project of the Graz University of Technology, NXP Semiconductors Austria GmbH. Funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG832171*

The objective of this thesis is the development of optimization techniques for reducing the tradeoff between application ROM size and application execution speed. Two compression methodologies are proposed for compressing Java Card applications off-card and then decompressing them during run-time. To overcome the slow-down of the execution introduced by the decompression phase, by means of a hardware/software co-design approach, parts of the Java virtual machine have been moved into the hardware architecture making the execution faster.

Figure 1 shows an overview of the enhancements introduced into the overall Java Card architecture. Two different phases can be distinguished: the application compression that takes place off-card before the installation of the application, and the application decompression that takes place on-card during the execution of the application. Two

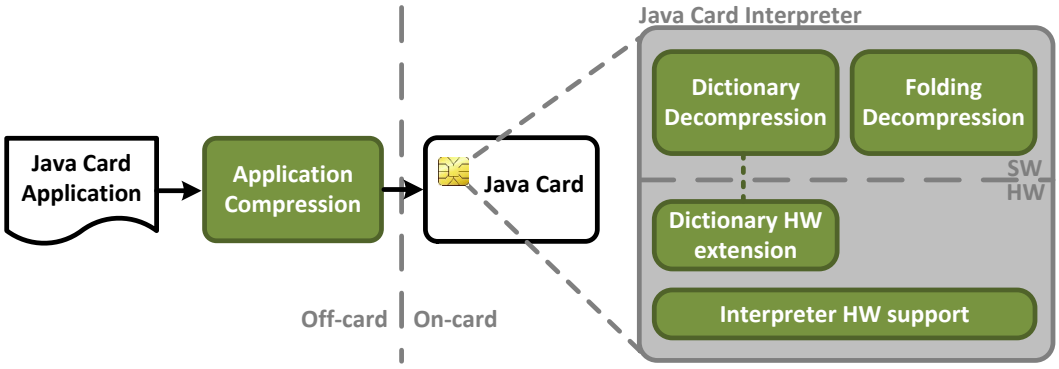

Figure 1: The compression is a post issuance phase. A Java Card virtual machine enabled for compression can install a compressed application. The additional modules in the virtual machine permit the run-time decompression of the applications. Moreover, the hardware support for the interpreter allows a faster interpretation.

main compression techniques have been proposed, the *dictionary compression* and the *folding compression*. In the *dictionary compression* [2] repeated portions of the application code are substituted with macros, whose definitions are stored in a dictionary. Beyond the plain version, two additional modalities are evaluated. In the latter, the macro definitions have some internal generalizations that make the macros usable for similar sequences.

In addition to saving space, the *folding compression* [3] also has the advantage of increasing the execution time of the application. In this compression methodology, specific sequences of Java bytecode instructions (foldable instructions) are substituted with new superinstructions that perform the same task with a reduced use of the Java operand stack. The two compression techniques can be combined in a *light-weight compression* [4], but they are not orthogonal. In fact they interfere with one another in both space savings

[2] *On the dictionary Compression for Java Card Environment, Proceedings of the 16th Workshop on Software and Compilers for Embedded Systems (M-SCOPES'13), St.Goar, Germany, 2013*

[3] *Instruction Folding Compression for Java Card Runtime Environment, 17th Euromicro Conference on Digital Systems Design (DSD'14), Verona, Italy, 2014*

[4] *A light-weight compression method for Java Card technology, The 4th Embedded Operating Systems Workshop (EWiLi'14), Lisbon, Portugal, 2014*

and run-time execution of the compressed application.

Due to the fact that Java is an interpreted language, the execution speed of an application is lower compared with the speed achievable with a native application. To improve the execution speed performance, the Java Card interpreter has been enhanced using hardware/software co-design methodologies [5] [6]. In the first place, the software architecture of the interpreter has been modified in order to make the co-design possible. In the second place, parts of the interpreter are moved into the microcontroller architecture allowing a consistent increase in speed of the interpretation. Additionally, the execution of applications compressed with techniques based on a dictionary is slower than the execution of non-compressed applications. The further application of the hardware/software co-design to the part of the interpreter responsible for the dictionary decompression reduces the impact of the dictionary decompression on the execution speed [7].

To assess the entire system, a prototype is build. As there are multiple configurations for different variants, the build system has to make possible the creation of different prototypes. Hence, a variant modeling build system is implemented to combine all the possible features in the final prototype [8].

To summarize, this dissertation presents hardware/software co-designed improvements to the Java Card virtual machine that reduce the memory needed to store the applications and improve the execution speed of the interpreter. For the compression of the application we propose two compression techniques derived from the plain dictionary compression and one technique based on the instruction folding mechanism. On the interpreter side, we enhanced the interpretation speed by means of hardware support, where features for speeding up the dictionary decompression were also integrated.

[5] *A High Performance Java Card Virtual Machine Interpreter Based on an Application Specific Instruction-Set Processor, 17th Euromicro Conference on Digital Systems Design (DSD'14), Verona, Italy, 2014*

[6] *Hardware/Software Co-Design for a high-performance Java Card Interpreter in Low-end Embedded Systems, under review for the journal Microprocessors and Microsystems: Embedded Hardware Design (MICPRO), Elsevier*

[7] *An Application Specific Processor for Enhancing Dictionary Compression in Java Card Environment, 5th International conference on Pervasive and Embedded Computing and Communication Systems (PECCS'15), Angers, France, 2015*

[8] *Embedding Research in the Industrial Field: A Case of a Transition to a Software Product Line, International Workshop on Long-term Industrial Collaboration on Software Engineering (WISE'14), Vasteras, Sweden, 2014*

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
BA	Bytecode Area
BB	Basic Block
CPU	Central Processing Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
FPGA	Field Programmable Gate Array
HW	Hardware
JPC	Java Program Counter
JVM	Java Virtual Machine
LVs	Local Variables
OS	Operating System
PC	Personal Computer
RAM	Random-Access Memory
ROM	Read-Only Memory
SW	Software
SoC	System on Chip
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VM	Virtual Machine

Glossary

Hardware/Software Co-design
"Hardware/software co-design investigates the concurrent design of hardware and software components of complex electronic systems. It tries to exploit the synergy of hardware and software with the goal of optimizing and/or satisfying design constraints such as cost, performance, and power of the final product. At the same time, it targets the reduction in the time-to-market considerably."[1]

Java Virtual Machine
"A virtual machine (VM) is a software abstraction of a computer that often executes as a user application on top of the native operating system. "[2]
A Java virtual machine is a "Virtual machine that enables Java programs to execute on many different architectures without recompiling Java programs into the native machine language of the computer on which they execute. The JVM promotes application portability and simplifies programming by freeing the programmer from architecture- specific considerations. "[3]

Java Interpreter
"Interpreters are programs which directly execute source code or code that has been reduced to a low-level language that is not machine code. Programming languages such as Java compile to a format called bytecode (although Java also can be compiled to machine language), which acts as machine code for a so-called virtual machine. "[3]

Java Bytecode
Java bytecode can be referred to as two concepts distinguishable from the context where they are used. The first case refers to the entirety of the executable code of a Java class. In the second case Java bytecode is used to indicate a Java opcode. The plural refers to a multitude of Java opcodes.

Chapter 1

Introduction

1.1 Motivation

Smart Cards were born in the fifties as a support for data storage to secure against forgery and tampering. The information on the card holder was printed on the surface and the card was presented to the person responsible for the payment. Such a system relied on the quality and integrity of the personnel. A step further in the evolution of smart cards is the introduction of smart cards that have a magnetic strip and are provided with a secret PIN number that is entered for payments. In due course, the security level of smart cards with a magnetic strip was revealed to be not sufficiently secure against attacks, because anyone supplied with the necessary equipment could read the magnetic strip. In the seventies, thanks to improvements in electronics, the first chips with data storage and logic integrated together were available. Ultimately the first exemplars of smart cards with logic and data storage were issued [4].

Nowadays, smart cards are considered an embedded system with very limited resources. They are usually equipped with 8 or 16 bit processors, several kilobytes of RAM and several hundred kilobytes or megabytes of non-volatile memory (ROM, EEPROM and Flash memory). Smart cards are very pervasive in many sectors of society. It is possible to group the applications in three main fields:

- Telecomunications (SIM cards)
- Banking (credit cards, ATM cards)
- Identification (health Insurance card, transportation)

As can be seen in Figure 1.1, the volume of shipments is growing every year. In telecommunications the number of shipments has begun to stabilize, whilst in banking and government applications the trend is increasing. In a market where the shipments are in the order of billions, the optimization of resources means consistent improvements in profit margins.

1.1.1 Java Card

Smart card applications are often programmed in C and assembly in order to keep their ROM size small and the execution performance high. This development approach has a

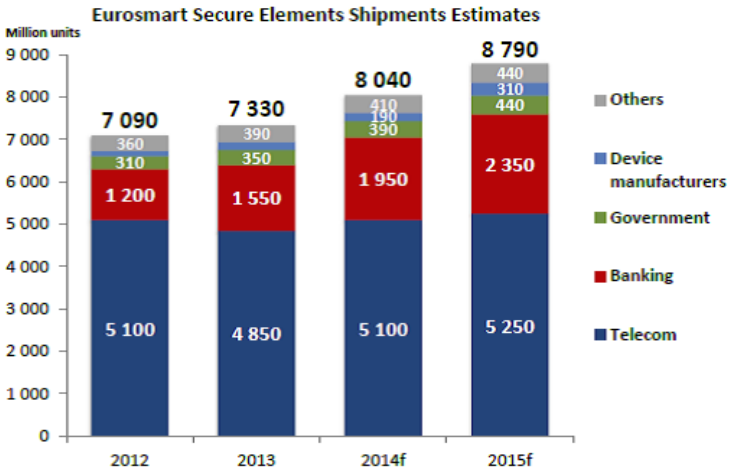

Figure 1.1: Smart card market

high cost in terms of effort required to keep the application portable on other platforms. Moreover, in a possible future scenario where more applications are hosted in a single card, the installation of applications developed by third parties is thwarted by the absence of a standard platform. For these reasons, in the last few years, Java technology has been adopted in the development of applications for smart cards. In small footprint devices like smart cards the deployment of a standard Java environment is unfeasible, because it would need a system with resources of a higher order of magnitude. For this reason, in the late nineties, a new Java standard for smart cards was defined under the name of Java Card [4, 5, 6, 7, 8].

In the last few years, Java Card has continuously gained market share as reported in the histograms of Figure 1.2. The penetration of Java Card is particularly growing in the identification sector as shown in the histogram on the right of the figure.

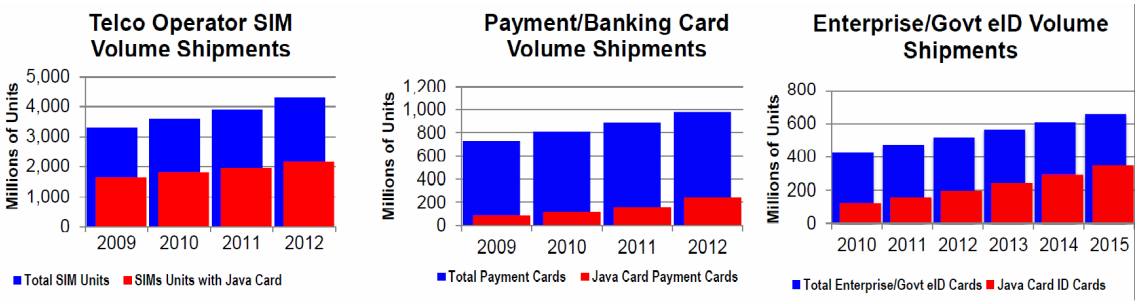

Figure 1.2: Java Card penetration in the smart card market

As standard Java, the Java Card environment is based on a Java virtual machine that interprets the Java bytecode. Differently from standard Java, the applications for Java Card are shipped in the format of Java Card converted applet (CAP) file. The CAP file is the result of the conversion of the Java *class* file into a pre-linked version. Before the

installation, the CAP file is checked with the Java Card bytecode verifier to assure the correctness and security of the application that will be installed. The CAP file consists of different components used at the time of installation. Among the components the most relevant for the work in this thesis is the *method* component that contains the bytecode of all the methods of the classes constituting the application. Statistically, the method component occupies most of the ROM size of an installed Java Card application [9]. Hence, the development of techniques for the reduction of the size of the method component would decrease the overall ROM occupation of the installed applications, allowing the use of smaller non-volatile memories.

The drawback of applying compression techniques is a general time overhead in the execution due to the restoration of an executable form. Additionally, the Java VM is based on an interpreter that, because of its interpretative nature, executes applications slowly, compared with applications running natively on the hardware platform. In this context, a hardware/software co-design approach helps to reduce the compromise between execution speed and ROM size of the applications.

1.2 Hardware/Software Co-design for small-footprint Java Cards

The David Project

This thesis is part of the "Design Flow for Low-End Java Card Operating Systems" (David) collaborative research project, shared between the Institute for Technical Informatics at the Graz University of Technology and NXP Semiconductors Austria GmbH. The David project focuses on the trend of building Java Card systems with adequate security level and run-time performances at a low price. Figure 1.3 sketches the area of interest of the David project with a parallel view of the Java technology and the hardware platforms that host the Java environment.

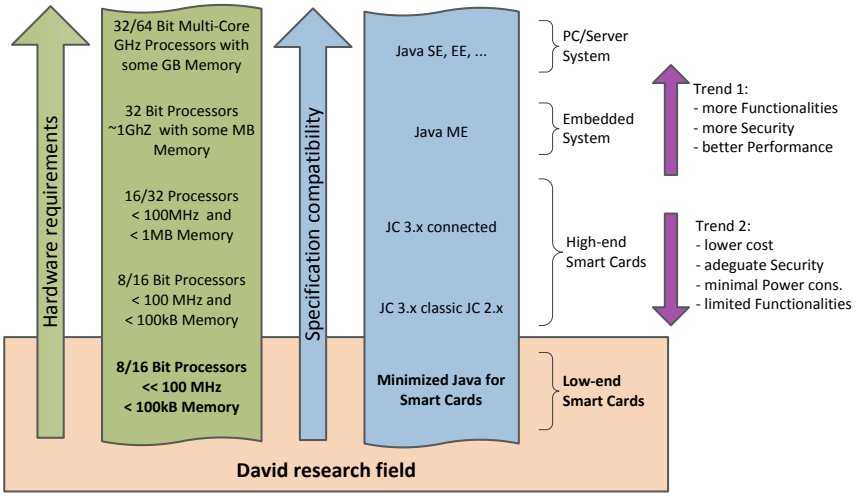

Figure 1.3: Area of interest of the David project

As the memory is one of the most expensive components in a smart card system, the David project proposes to minimize the memory footprint of a Java Card system. For this purpose two strategies are addressed:

- Minimizing the Java Card application ROM size by means of compression techniques
- Minimizing the Java Card runtime environment (JCRE) memory footprint keeping out of the ROM image unnecessary code and features

This thesis focuses on the first of the two strategies and investigates the opportunities for the reduction of the applications footprint without compromising the run-time performances of the Java Card system.

An advantage of the David project is the presence of the industrial partner NXP Semiconductors Austria GmbH, a company involved in the development of a Java Card operating system as well as in the design of smart cards. This combined design strategy allows a complete view of the overall Java Card system and better access to the functional concepts of the Java Card environment needed for a hardware/software co-design approach [10, 1].

1.2.1 Problem statement

Smart card shipments as well as Java Card penetration in the smart card market are continuously growing in the past few years. Because of the large diffusion of the technology in many disparate fields, some use cases are satisfied by cheap products with scarce resources. In the context of smart cards, using small memories and cheap processors is the solution for keeping the cost low. Therefore, small non-volatile memories means less space for the applications with the consequent reduction of the functionalities available. Analogously, the use of low-end processors lowers the run-time performance of the system with the possible extreme consequence of no longer satisfying the timing constraints demanded by the customers.

In the scientific community, the reduction of the application ROM size is faced with the introduction of compression techniques while the issue of low-end processors is mitigated through the use of application specific processors. These solutions leave open problems:

- Compression techniques have the drawback of a slow-down in the execution of the compressed application
- Research works in the context of compression for Java Card are limited to the dictionary compression
- The coordinated design of hardware and software for Java Card is usually limited to security issues
- The adoption of application specific processors for Java environments lowers the flexibility of the system
- Hardware acceleration techniques have never been considered for compression in Java Card

These limitations have been addressed by the development of new compression techniques and by the proposition of a flexible hardware support for the Java Card system. In the field of compression, extensions of the dictionary compression methodology are investigated as well as the development of a new compression technique based on execution optimization. To improve the performance of the virtual machine a new approach for the interpretation has been proposed; from this approach it is possible to develop hardware solutions that improve the execution speed, without compromising the flexibility needed in the software development of the Java Card system. Finally the dictionary compression technique has been integrated in the new architecture, developing an additional hardware support to speed up the dictionary decompression.

1.2.2 Contribution of this Thesis

In summary, this thesis provides contributions to the following fields:

1. **Extension of dictionary compression and folding compression:** Dictionary compression is deeply investigated in this thesis. The static and dynamic dictionary modalities are evaluated, analyzing the advantages and disadvantages. Moreover, a dictionary compression technique based on Java subroutines is introduced. The dictionary compression is extended in two variants that make use of generalized macros with arguments. An additional compression technique based on the optimization mechanism called instruction folding is proposed. Finally, the latter technique is integrated with the dictionary compression resulting in a new compression technique with high space savings and low impact on the run-time of the applications.

2. **Hardware support for the Java Card interpreter:** The Java interpreter is redesigned with a new interpreter architecture. The fetch and decode phases of the interpretation are moved from a central common loop to each procedure implementing the Java bytecodes. Beyond a full software implementation, two variants are proposed. The first has the decode phase implemented in hardware, the second has the fetch and the decode phases realized in hardware. The two implementations with the hardware support show a sensible improvement in the execution speed.

3. **Hardware support for the dictionary compression:** The last contribution of this thesis consists of the integration of the dictionary compression with the interpreter with hardware support. In the first realization the integration is done in software while in the second an additional hardware module is developed to support the interpretation of the dictionary macros directly in hardware.

1.2.3 Thesis Structure

The rest of this thesis is organized as follows:

Chapter 2 reports the research work related to compression techniques and interpreters. The first part is dedicated to compression techniques with particular attention to the compression of executable code. The second part is relative to possible execution optimizations for Java environments, with a description of the JIT compilation and the concepts behind it. The last part refers to hardware realizations of the Java virtual machine with an overview of Java processors and Java co-processors.

Chapter 3 presents the concepts and the design so that the Java card can be enabled to execute compressed code with a hardware supported interpreter. In the first part, two extensions of the dictionary compression and a new compression technique based on the folding instruction optimization are proposed. In the second part the architecture of the pseudo-threaded interpreter is described. Moreover, the design of the hardware support for the pseudo-threaded interpreter is shown. In the last part, the hardware integration between the dictionary decompression and the interpreter with hardware support is disclosed.

Chapter 4 provides the evaluation of the proposed designs. In the assessment of the compression techniques, particular emphasis is given to the space savings and to the execution speed of a compressed application. In the sections regarding the Java interpreter with hardware support and with hardware support for dictionary decompression, a complete analysis of the ROM size, hardware overhead and execution speed-up are provided.

Chapter 5 draws conclusions and directions for future work from this thesis.

Chapter 6 collects the publications that constitute the back bone of this thesis.

Chapter 2

Related Work

This chapter provides a view of the related work on code compression and performance enhancement for Java. In this section, with regards to compression we analyze the most popular compression techniques and how they are applied to embedded systems. In the second section we report on the most significant methodologies for speeding up the execution in Java, taking into consideration both software and hardware solutions. Finally in a summary of this chapter the main differences between this thesis and related work are shown.

2.1 Data Compression

Code compression encompasses many aspects of information technology. Whatever the field, the objective of its application is the reduction in the space needed for the storage of the information. In general it is possible to differentiate compression techniques into those that are statistical and those that are dictionary based [11, 12].

The most popular statistical method is the Huffman coding [13]. As the name suggests, statistical methods are based on statistical analysis of the data to be compressed. The statistical analysis contains the probabilities for each symbol of the input data; from this statistical table the decoding tree is built and a Huffman code is assigned to each symbol. Generally, the higher the probability of a symbol is, the lower is the number of bits that composes its Huffman code. To allow the decompression, beyond the compressed code, it is necessary to forward the Huffman table with any correspondence between Huffman codes and the original symbols.

The dictionary compression techniques are not based on statistical information but on the analysis of the data [11, 12]. The algorithm for the compression analyzes the data to be compressed and substitutes *phrases* with *tokens*. The phrases relative to the tokens are stored in a dictionary that can be static or dynamic. In the case of a static dictionary, its phrases are defined per context. In the case of a dynamic dictionary, the dictionary is dynamically built during the analysis of the input data. Hence the phrases of the dictionary are specific to the data to be compressed and the dictionary is also strictly related to the data being compressed. In the decompression phase, the dictionary is needed by the decoder, therefore the dictionary is sent with the compressed data. The task of the decoder in the decompression phase is simple; it consists of substituting the

token in the compressed code with the associated phrases contained in the dictionary.

Popular subcategories of the dictionary compression are the LZ77 and the LZ78 compression techniques [14, 15]. LZ77 and LZ78 are the base algorithms for the DEFLATE [16] and LZMA [17] algorithms that are used in the PKZIP utility [18] that creates the .zip archives, the most popular archive format used in general purpose PCs. LZ77 and LZ78 techniques are based on a slide window where the phrases of the dictionary are selected from. The main difference between the two techniques regards the decompression: in the LZ77 the decompression has to start from the beginning while in the LZ78 the decompression can start at any point in the compressed data.

In Java systems, the PKZIP utility is used for the compression of the .class file and the creation of the JAR archive, the format for the storage and the shipment of Java applications. To execute an application stored in a JAR file, firstly the JAR archive is usually decompressed in the RAM memory and, at a later stage, the application can be executed [19]. Such a system is quite demanding in terms of RAM consumption, therefore is not applicable in a system with relatively limited resources like smart cards.

2.1.1 Compression of Executable Code

Before dealing with compression techniques, we briefly mention the code compaction that is an active field of research in compiler theory of the last years [20] [21]. Compaction techniques optimize the code, for example eliminating unreachable code or redundant code and performing code factorization. The code is fully executable after the application of the compaction techniques and therefore does not require and additional actions before it is executed.

Code compression is a different approach and it consists of a post-compilation process. The reasons for compressing the application can be categorized into two bottlenecks: memory occupation and transmission to the CPU. In the case of transmission, the code can be compressed in a non-executable form, to be decompressed later, before the execution. On the other hand, if the bottleneck is the memory, the compressed code should be directly executable because there is not enough space to decompress it before the execution [22].

The previous section mentioned JAR archives, an example of compression for *class* files, the executable code of Java. JAR files are an example of compression where the bottleneck is the transmission. The executable code of a JAR file is completely decompressed before the execution. The principal issue with this approach is the start-up time needed for the complete decompression of the application and the space required to store it temporarily.

In embedded systems, where the memory available is a hard constraint, compressing the executable code is a good way of saving space in the permanent memory. At the same time, the decompression of the code needs temporary space either in the permanent memory or in the random access memory. This trade-off limits the degree of freedom in embedded system with very limited resources. There are systems where keeping an entire application in memory is not possible. To overcome this problem, an approach presented in literature is the adoption a profile-guided compression where only the parts of the application less likely to be executed are compressed [23]. In this way, the decompression only takes place if the compressed code needs to be executed, and the memory needed for storing the decompressed code is relatively small.

Smart cards are embedded systems with very low resources [4]. An approach with a

profile-guided compression can hardly be applied, because the RAM is scarce and therefore it is unlikely to have hundreds of bytes available even for a partial decompression. The only method that does not use much RAM is the dictionary compression.

The application of dictionary compression techniques implies the modification of the architecture of the processor or of the interpreter that execute the compressed code. In the case of compression for machine code, a post compression process analyzes the application, creating the dictionary and compressing the code [24]. For the decompression, the standard hardware architecture is modified to allow the handling of the macros with the dictionary in the decode phase of the instruction execution. Other examples of hardware architectures able to execute code compressed with other compression techniques can be found in [25] and [26].

In Java, the compression with dictionary techniques is similar to the compression in a hardware architecture. The compression concerns the Java bytecode that the virtual machine executes [9] [27]. The main difference is that, instead of implementing it in a hardware architecture, the decompressor is implemented in the Java virtual machine. The dictionary compression is applied to embedded Java and to Java Card; the difference in the two technologies consists of the presence of an additional file format (the CAP file) and in the split architecture (off-card and on-card virtual machine) in the Java Card technology [6, 5]. In the compression, the CAP file, the result of the conversion of a *class* file, is analyzed. During the compression, the dictionary is created and the macros that are defined in it are used for the substitution in the Java bytecode of the repetition of their associated definition. After the installation of the application, the virtual machine enabled for decompression is able to run the compressed application; when it encounters a macro within the code, it uses the dictionary to execute the sequence of instructions that the macro represents.

2.2 Execution Optimizations in Java Systems

The execution speed of a general software application depends on three main factors: the hardware architecture, the compiler and the software architecture. In systems based on a virtual machine, we can consider the virtual machine as an additional element between the hardware platform and the application. The adoption of *superoperators* is a methodology for improving the performance of a virtual machine [28]. Superoperator is an entity that groups a bunch of instructions together, performing optimizations among the instructions, similar to what happens in register forwarding in a RISC architecture [29]. Taking as an example a stack based virtual machine, as the Java virtual machine is, some sequences of operations use the stack inefficiently, storing values into the stack before immediately fetching them in the next instruction. Superoperators optimize these kinds of inefficiencies by bypassing the stack and behaving like an instruction of a register based processor.

Popular Java environments, like those running on a personal computer, include the *Just In Time (JIT)* compilation for speeding up the application execution in Java [30, 31, 32, 33, 34]. JIT compilation, as the name suggests, compiles the Java bytecode and transforms it into machine code during the run-time. The compilation needs a considerable amount of RAM for keeping the compiled code in memory, and a *warm-up* time for the code that is compiled for the first time. During the compilation, the JIT compiler

performs optimizations among the bytecode instructions similar to those performed in superoperators.

In embedded systems, JIT compilation may not be implementable, because of the limited resources. For systems where the JIT compilation cannot be applied, an option is the extension of the instruction set of the Java virtual machine with superinstructions. Superinstructions consist of superoperators added into the instruction set of the Java virtual machine. The advantage of superinstructions can be summarized in two main points [35, 36, 37, 38]:

- A single bytecode dispatch is reduced to one compared to the number of dispatches needed for the equivalent sequence of bytecodes
- The superinstruction takes advantage of the superoperator optimizations, hence reducing the operands stack accesses

The superoperators principle has also been used in picoJava II, a hardware realization of a Java virtual machine [39]. In this Java processor, a particular run-time mechanism (folding mechanism) able to recognize sequences of bytecodes instructions (foldable instructions) is implemented. When the Java processor encounters a foldable sequence, it performs a sort of register forwarding optimization for eliminating the unnecessary accesses to the operand stack.

Another way to exploit superoperators specifically for foldable sequences in virtual machine consists of the usage of Java annotations [40, 41]. This methodology is particularly useful in systems with limited resources. A post compilation analysis searches for foldable sequences within the bytecode and annotates them. At execution time, the annotation aware Java virtual machine recognizes the annotations and, instead of executing the foldable instructions, executes an equivalent superoperator, saving instructions dispatches and accesses to the operands stack.

Virtual machines are based on interpreters, a software part responsible for the transformation of the bytecodes into computational actions. The most popular techniques for the interpreter realization are based on a big *switch* statement or on a token-threaded interpreter [42, 43, 44]. An alternative and more efficient interpreter is the *direct threaded* interpreter. In the direct threaded interpreter, the executable code consists of a sequence of addresses of the procedures that have to the executed [45, 46, 35]. The executable code is created at the compilation time; hence, the time needed for the translation of tokens into addresses is no more part of the interpretation, with a substantial improvement of the execution performance. The main drawback of this technique is the limited portability of the executable code that is strictly implementation dependent.

2.3 Java in Hardware

In the last section, the opportunity to implement the Java virtual machine in hardware is mentioned. It is possible to distinguish two kinds of hardware implementations:

- Java co-processors
- Java processors

Java co-processors are a support architecture to a main classic processor architecture. The co-processor speeds up the execution of Java bytecode. The most famous example of a Java co-processor is represented by the *Jazelle* technology developed by ARM [47, 48]. The Jazelle co-processor executes most of the Java instructions directly in hardware, translating, by means of a hardware unit, the bytecodes in sequences of processor instructions. The remaining Java bytecodes are handled with exceptions and software procedures. The Java co-processor is an example of a more general scheme known as application specific instruction set processor (ASIP). In ASIPs the instruction set is extended to allow access to new hardware features [49, 50, 51]. In the case of the Jazelle co-processor, an extended instruction set allows integration with the native code, e.g. with an operating system. Similar examples of a Java co-processor are proposed in [52], [53], [54] and [55]

Differently from co-processors, Java processors are a complete implementation of the Java virtual machine in hardware. The Java bytecodes are directly executed on a stack architecture without intermediate translations to register machine instructions. The prominent example of Java processor are *picoJava* and its successor *picoJava II* [56] [57] [58]. The main drawback of this solution is that the processor can only execute Java code; hence, all the software on the system has to be written in Java, including, for example, the peripheral drivers. Other related works on of Java processors are [59] and [60].

2.4 Summary and Difference to the State-of-the-Art

The work of this thesis is focused on Java for low-end embedded systems such as smart cards, where resources are very limited. The aspects that we take into consideration are the memory footprint of the applications and the execution speed. Usually, these two aspects are opposed to one other and hence it is necessary to find a trade-off. Regarding the ROM size of the applications, two different compression methods and their combination are investigated. For the execution speed-up, a hardware/software co-design solution for the interpreter and its combination with a compression technique are proposed. The following points summarize the main difference from the current state-of-the-art:

- This work investigates the dictionary compression taking into consideration the static and the dynamic dictionaries, evaluating their architectural impact in the Java Card design flow. Moreover, the plain dictionary technique is extended with two derivations. The first takes the arguments of the bytecodes into the macro definition as the argument of the macro. The second uses generalized instructions into the macro definitions that are specified by the argument of the dictionary macro. Additionally, a dictionary method that uses Java subroutines is evaluated.

- The second compression method is based on the folding mechanism. In literature, the folding mechanism is used for a Java processor, and superinstructions are used only to improve the execution speed. The approach presented here adopts the folding mechanism for compressing the bytecode, speeding up the execution of the compressed application at the same time. Moreover, the combination of folding compression and dictionary compression is evaluated.

- The second element to this thesis regards the hardware/software co-design for the interpreter. In a different approach to that of the Java processor and co processor,

the implementation proposed here does not execute the bytecodes in hardware and keeps the execution phase of the interpretation in software, giving an higher degree of freedom to the developers. In the proposed implementation the fetch and the decode phases of the bytecode interpretation are performed in an application specific processor.

- Finally, the interpreter with hardware support is extended to handle the dictionary compression. Previous research works investigated the opportunity of implementing dictionary compression in RISC architectures for the compression of the machine code, but there is no approach regarding the hardware support for dictionary compression on Java virtual machine.

Chapter 3

Hardware/software Co-design for small-footprint Java Cards

This chapter provides an overview of the publications regarding bytecode compression and interpretation enhancement.

3.1 Architectural Overview

The first part of this work regards the compression of the executable code. The Java Card virtual machine can be subdivided into an off-card part and an on-card part. As a consequence, the compression methods that we propose are distributed between the two parts: the compression phase is part of the off-card Java virtual machine mechanism, whereas the decompression phase is part of the on-card Java virtual machine. In the second part of the work we perform hardware/software co-design to enhance the performance of the Java interpreter that is sited in the on-card virtual machine. Figure 3.1 schematizes the structure of the Java Card architecture, with the contribution of this work mapped to the scientific publications.

Starting with the Java Card applet in the form of a CAP file, it is possible to apply a compression mechanism based on a dictionary [9]. Exploiting the dictionary compression opportunities, we propose two techniques derived from the base dictionary compression in *Publication 1*. The two techniques make use of generalizations for the macros, allowing a more flexible usage of a single macro for sequences that are *similar*. In the same publication the dictionary compression is also implemented by means of the Java bytecodes *JSR* and *RET*, therefore using procedures and keeping the realization in the Java layer. After the installation of the compressed application, the Java virtual machine can execute the compressed code thanks to the extension for the decompression.

In *Publication 2* a compression method based on the folding mechanism (the *folding compression*) is proposed. Usually, compression methods have the drawback of slowing down the execution of the compressed application because of the decompression. In the proposed compression method the decompression is possible with the help of the extension of the Java virtual machine with the new *superinstructions*. The execution of the compressed code is faster than in the case of non-compressed code, thanks to the optimizations brought in by the superinstructions.

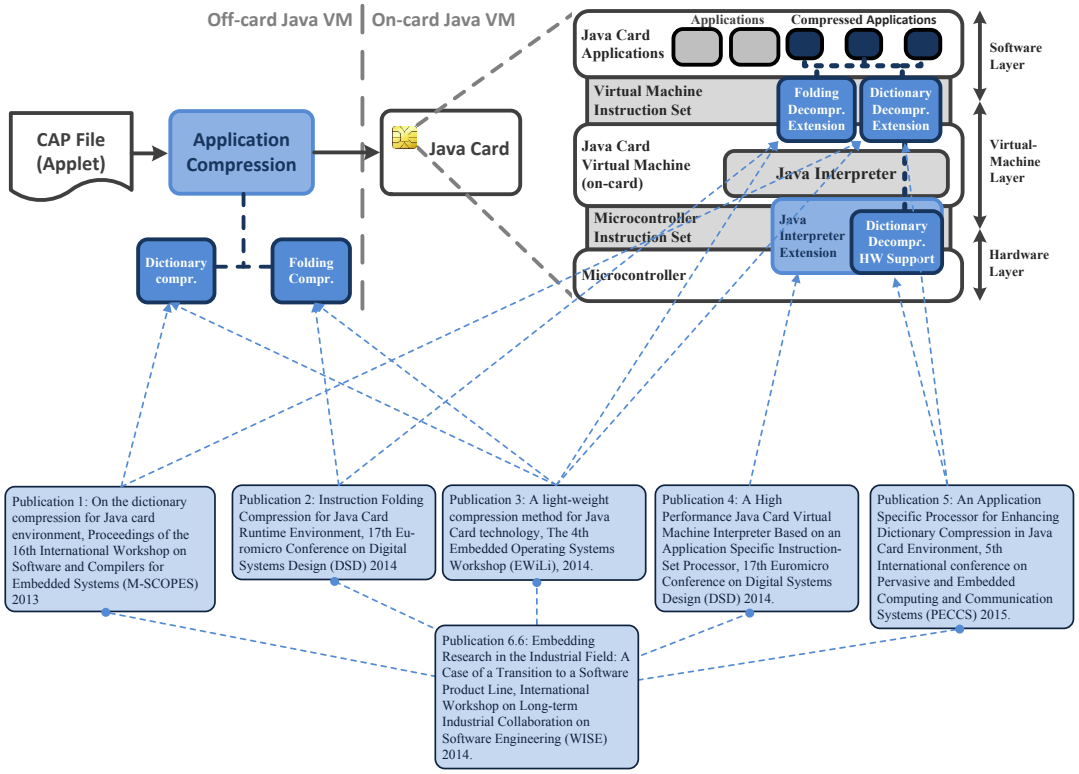

Figure 3.1: Publications overview for a Java Card with hardware support and enabled for compression

Publication 3 proposes the combination of the dictionary compression and the folding compression and the analysis of the interactions of the two methodologies. The result is a new compression method with higher space savings and with a very low impact on the execution time of a compressed application.

Besides the proposition of compression methodologies, effort is put in improving the performance of the interpreter. In *Publication 4* a *pseudo-threaded* architecture for the Java interpreter is introduced. By means of the application of a hardware/software co-design, parts of the interpreter are shifted into the microcontroller architecture with the result of a consistent reduction in the interpretation time.

The last part of the design consists of the link between the new interpreter architecture and the compression techniques. As previously reported, in the folding compression an extension of the instruction set of the Java interpreter with superinstructions makes the decompression and the interpretation of the compressed code possible. The interpreter treats the superinstructions in the same way as the standard Java bytecodes. In the case of dictionary compression, the decompression makes use of a dictionary table. *Publication 5* deals with a hardware support built over the *psedo-threaded* interpreter to speed up the dictionary decompression. The hardware support accelerates the decoding of dictionary macros performing in hardware the address resolution and the fetch of the address of the dictionary definition.

Finally, a variant management system is built to manage all the variants of Java Card environments obtained with the different enhancements. The description of a transition to a product lines system organized with variant management is given in *Publication 6*.

3.2 Compression in Java Card

Differently to standard Java, Java Card architecture is split into two parts, an off-card part and an on-card part. Figure 3.2 clearly sketches the processes composing the off- and on-card Java virtual machine. The off-card Java Card refers to the processes that transform and verify the *class* and install it on the smart card. The scheme in the figure shows that

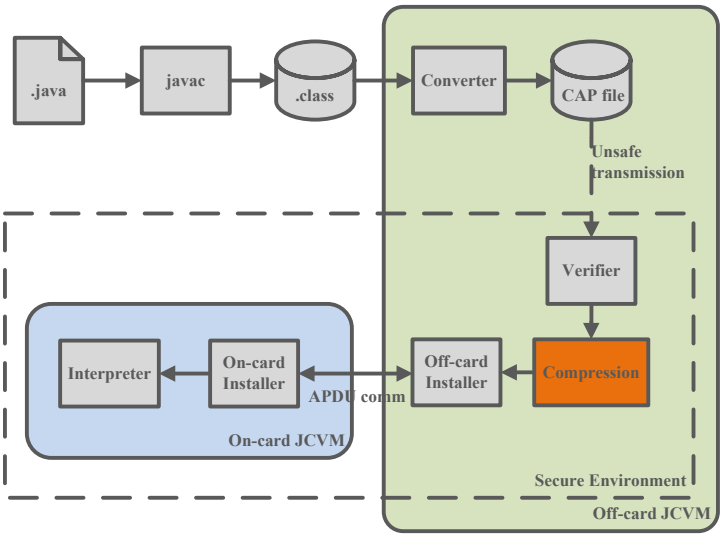

Figure 3.2: Java Card virtual machine enabled with compression. The Java VM is split into off- and on-card VM. The compression is applied after the verification, before the installation.

the compression takes place between the verification and the installation processes. This is not the only possible choice but it is the most convenient one. Assuming for example the compression takes place after the conversion, the dictionary has to be stored into a custom component of the CAP file. Java Card specification [6] states that the CAP file can have additional custom components, but this kind of practice is discouraged in industry, because of the possible incompatibility with existing systems. Moreover the Java Card verifier should be able to handle the applied compression technique, increasing the complexity of it and opening new critical scenarios in the field of security. The adopted design solution with the compression after the verification allows the issue of applications in standard CAP file and the use of a standard verifier. Moreover, as the verification, compression and installation are located in a secure environment, the compression does not decrease the level of security in the overall architecture. Both dictionary and folding compressions accord with this scheme.

3.2.1 Dictionary Compression

Dictionary compression consists of the substitution of repeated sequences of information with tokens whose definitions are stored in a dictionary. The dictionary is used in the decompression phase for substituting the tokens in the compressed code with their definitions in order to reconstruct the original information. For the reduction of the code size in Java Card, the code compressor analyzes the *method component* of the CAP file where it searches for repeated sequences of bytecodes [9]. Figure 3.3 sketches the principle of dictionary compression.

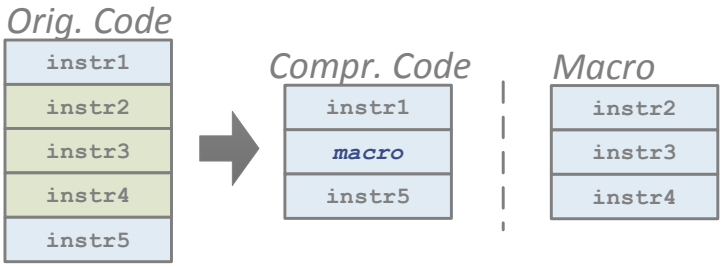

Figure 3.3: Principle of the dictionary compression.

Section 6.1 discusses accurately the search algorithm, paying attention to the constraints for the search resumed in the definition of SISE block to which each possibility has to be compliant. Moreover, the creation of the dictionary is evaluated; a dictionary can be related to a specific application (dynamic dictionary) or can be general for each application (static dictionary) [61].

During the execution of the compressed code, the Java VM has to provide for the decompression. Dictionary decompression fits very well in an architecture based on an interpreter, allowing the decompression during the run-time without the need to decompress the application before its execution. When the interpreter encounters a dictionary macro, it looks up the macro definition in the dictionary macro table and starts the interpretation of the instructions contained in the macro [61].

3.2.2 Dictionary Compression with Generalized Macros

Similar sequences of bytecodes repeated a number of times have a space saving potential from which the plain dictionary compression cannot benefit. In Section 6.1 two dictionary compression techniques with generalized macros are proposed [61]. The two techniques collect not only sequences equal to each other, but also sequences that are similar to better exploit the information redundancy.

Generalization of the Java Op-code Arguments

In the Java Card instruction set part of the bytecodes are composed by an op-code and an argument. The first technique with generalized macros is based on the observation that many sequences of Java bytecodes have the same op-codes but not the same arguments.

The general macro has the Java op-codes in its internals but not the varying arguments that are left out, as an argument of the general macro.

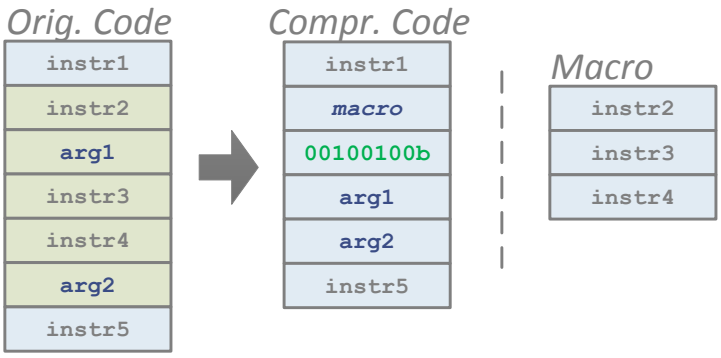

Figure 3.4: Representation of the dictionary compression with the generalization of the op-code arguments. The arguments of the Java op-codes are left outside the macro definition as arguments of the macro in the compressed code.

Figure 3.4 graphically explains the composition of a macro with generalized arguments. A bit mask is necessary to provide the option of dynamically specifying which of the arguments are left outside the macro definition and therefore the ability to rebuild the original sequence correctly. During the execution, the interpreter is therefore able to build the original sequences inserting the op-code arguments contained in the macro argument in the positions described by the bit-mask. Additional information is available in Section 6.1 [61].

Generalization of the Java Op-code

The generalization in Section 3.2.2 is not the only possible one. Sequences of Java bytecodes may also differ for one or more Java op-code. Figure 3.5 shows how it is possible to represent two similar sequences by means of the same macro introducing wildcard instructions in the macro definition. The wildcard instruction inside the macro definition substitutes a bytecode that is in turn specified by the argument of the macro.

At the run-time, the interpreter substitutes the wildcard instruction with the original instruction, on the basis of the argument of the macro. A more detailed explanation of the technique with the support of clarifying examples is reported in Section 6.1 [61].

Dictionary Compression with Java subroutines

A disadvantage of dictionary compression is the complexity that it introduces for the management of the dictionary during the decompression phase. In fact, the VM needs to be modified in order to recognize the dictionary macros and to look for them up in the dictionary table.

In the Java Card standard there are two bytecode instructions, *JSR* (jump subroutine) and *RET* (return from subroutine) that are used for building subroutines. The JSR and RET have a scope limited to the method that they belong to. In Section 6.1, a dictionary

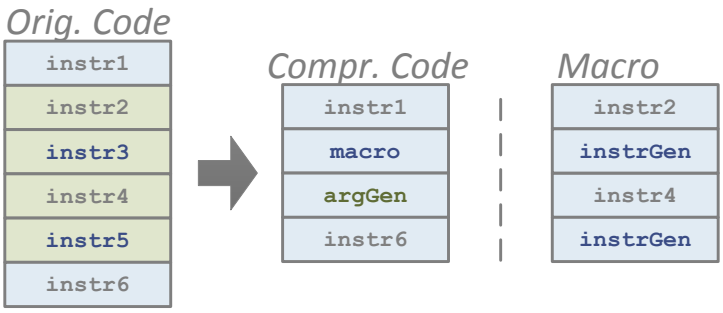

Figure 3.5: Representation of the dictionary compression with the generalization of the op-code. Some of the Java op-code in the macro definition are substituted with a wildcard instruction. The argument of the macro in the compressed code contains the information needed to identify the original Java op-code inside the macro definition.

technique based on subroutine is proposed [61]. The main advantage of the technique consists of its compliance with a standard Java Card VM. The limitation of the technique is its scope, that is limited to a Java method, instead of the entire method component as in the previous dictionary techniques; this limitation lowers the probability of finding repeated sequences and hence the space saving potential.

3.2.3 Folding Compression

The Java VM is a stack based architecture; the operators are copied onto the stack, an operation is performed on them, and then they are copied from the stack to a destination. This computation scheme is in some cases (*foldable sequences*) redundant and can be reduced to a register machine operation that directly accesses the resources and destination. Less use of the stack has the consequence of an execution acceleration. The folding mechanism is implemented in picoJava as a hardware acceleration module similar to a register forwarding mechanism in a pipelined architecture [39].

Folding compression finds its basis in the folding mechanism. The folding compressor searches for foldable sequences in the method component, and substitutes them with new superinstructions. The folding superinstructions constitute an extension of the Java Card VM instruction set. As for the folding mechanism, the folding compression allows for a reduction of the operands stack use. Moreover, it creates space savings in the method component due to the fact that the superinstructions and their arguments occupy less space than the foldable sequences that they substitute. A more complete treatment of the folding compression is presented in Section 6.2 [62].

An example of how the folding compression works is reported in Figure 3.6. The original code is a typical foldable sequence formed by two load instructions, an operator instruction, and a store instruction. At the right side of each sequence of instructions there is a representation of the operations inside the Java stack machine. In the original sequence there is a high use of the operand stack with copies from and to the local variables. On the right part of the figure the compressed code with the superinstruction is shown. The argument of the superinstruction specifies the index of the local variables in use and the

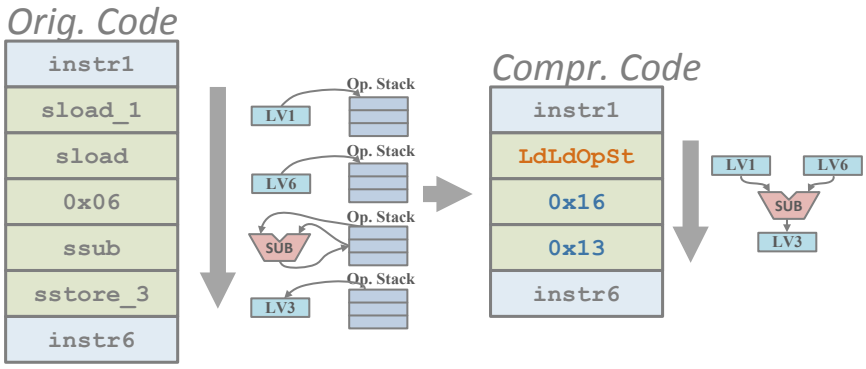

Figure 3.6: Example of the folding compression. A foldable sequence of Java bytecode is reported on the left side of the figure. The right side shows the equivalent folded superinstruction. On the right side of the code examples there is a representation of the operations performed in the Java VM.

operation to be performed. In this example the compressed code occupies three bytes that means two bytes saved compared with the original code. To do this we expressed the local variable indexes with a nibble (four bits) instead of a byte as the standard says. It is not possible to compress sequences involving a local variable with index higher than fifteen, but such sequences are statistically improbable [62].

3.2.4 The light-weight Compression

Section 6.3 presents the light-weight compression, a compression methodology created by the combination of the dictionary compression and the folding compression [63]. The light-weight compression consists of two successive steps: the application of the folding compression and, in the second step, the application of the dictionary compression. The order of application is dictated by the need to minimize the decompression time overhead.

Applying the folding compression in the first step means maximizing the advantages that it generates in the run-time decompression. After the application of the folding compression, the dictionary compression is applied to a code that is already compressed, therefore to a code with a lower number of possible sequences. The total space saving S can be expressed with the formula

$$S = S_f + (1 - k_1) \cdot S_d$$

where S_f and S_d are the space savings due to the application on the non-compressed application of the folding compression and the dictionary compression, respectively; k_1 is a coefficient that expresses the reduction of the dictionary compression efficiency in space savings due to the previous application of the folding compression. The value of k_1 is always positive and ranges between 0 (absence of interaction) and 1 (the sequences compressed by the folding compression overlap with the sequences potentially interested by the dictionary compression making the dictionary compression not effective).

Analogously to the space savings, the final effect of the compression (R) on the execu-

tion speed of the compressed code is described by the formula

$$R = R_f + (1 - k_2) \cdot R_d$$

where R_f and R_d are the effect on the execution time due to the folding compression and to the dictionary compression, respectively; the coefficient k_2 accounts for the reduction of the dictionary compression due to the interference from the folding compression. The value of k_2 ranges between 0 and 1, but a high value entails a lower contribution from the dictionary compression. The contribution of the dictionary compression is negative on the execution (slow-down); hence, in this case, the interaction improves the overall effect of the compression on the execution time of the compressed application. Section 6.3 presents an exhaustive explanation of the interaction of the two techniques and the influence that such interaction has on the space saving and on the execution time of the compressed code [63].

3.3 Hardware Support for the Java Interpreter

The Java interpreter is the part of the Java VM in charge of the translation of the Java bytecodes into actions of the Java VM. This section proposes a new architecture for the Java interpreter based on the thread interpretation. The new architecture is the starting point for the hardware/software co-design of the support for the new interpreter.

3.3.1 The pseudo-threaded Interpreter

In embedded systems, the Java interpreter is usually coded in C or assembly by mean of a big switch statement or a table of functions pointers [42, 43, 44]. The interpreter of a VM covers a role that is functionally very similar to a processor. Indeed, in the interpretation of a Java bytecode, it is possible to distinguish three phases that are executed cyclically:

- *Fetch*: the actual Java bytecode is read from the code memory at the address contained in the Java program counted (*JPC*)
- *Decode*: the actual Java bytecode is used as index to fetch from the look-up table the address of the corresponding procedure; the address of the procedure is then written in the processor program counter (PC)
- *Execute*: the procedure corresponding to the actual Java bytecode is executed

Figure 3.7 depicts the functioning of a classic Java interpreter (classic token interpreter) based on a while loop and a look-up table of function pointers to the procedures implementing the Java bytecodes. As can be seen in the figure, after the execution of the procedure implementing the Java bytecode, the execution flow returns to the end of the while loop before then branching again at the beginning of the loop and starting a new cycle.

The return branch from the functions implementing the Java bytecodes can be avoided unrolling the while loop and repeating the fetch and decode parts of the interpretation at the end of each function implementing a Java bytecode as shown in Figure 3.8. As can be seen in the figure, each bytecode procedure ends with the fetch and decode of the next

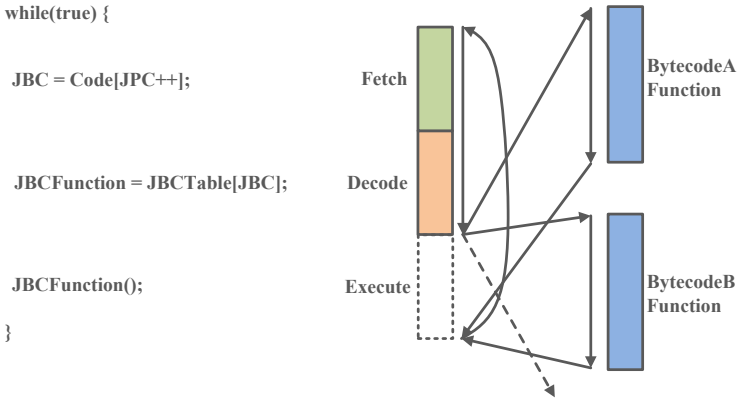

Figure 3.7: Classic Java interpreter in embedded systems. On the left side the pseudo-code for the interpreter is reported. The corresponding representation of the execution flow is reported on the right part of the picture.

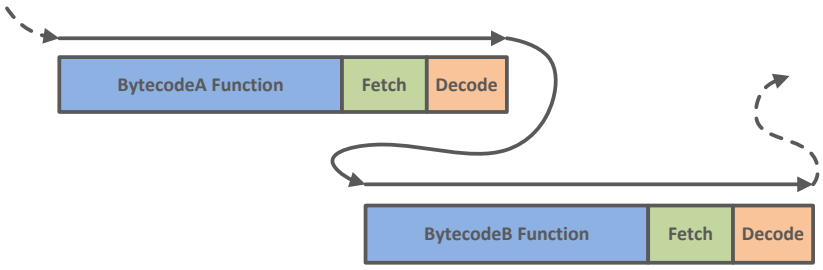

Figure 3.8: Representation of the *pseudo-threaded* Java interpreter. The fetch and the decode phases of the interpretation are repeated at the end of each function implementing a Java bytecode.

bytecode and with the jump to the relative bytecode procedure. The new architecture takes the name of *pseudo-threaded* interpreter to distinguish it from the threaded code in [45]. An overview of the compiler extension for handling the pseudo-threaded code with an analysis of the main advantages and drawbacks is given in Section 6.4 [64]. The main advantage of this interpretation technique is the faster execution, there being only one jump at the end of each Java bytecode function instead of a return at the end of the while loop and a jump to the beginning of the loop. The main disadvantage consists of the increment of the code size because of the repetition of the fetch and decode code at the end of each Java bytecode. For this reason a complete software solution is not practicable, but the architecture is valid for the creation of the hardware support as it is reported in Section 3.3.2 and Section 3.3.3.

3.3.2 Hardware Support for the Decode Phase

Section 3.3.1 proposes the pseudo-threaded interpreter architecture for a faster interpretation of the Java bytecodes. This section presents the hardware support for executing

the decode phase of the interpretation in hardware. In the decode phase, the interpreter translates a Java bytecode in an address from where the execution part of the interpretation starts. During the decode phase, the interpreter uses a look-up table where all the addresses of the procedures implementing the Java bytecodes are contained.

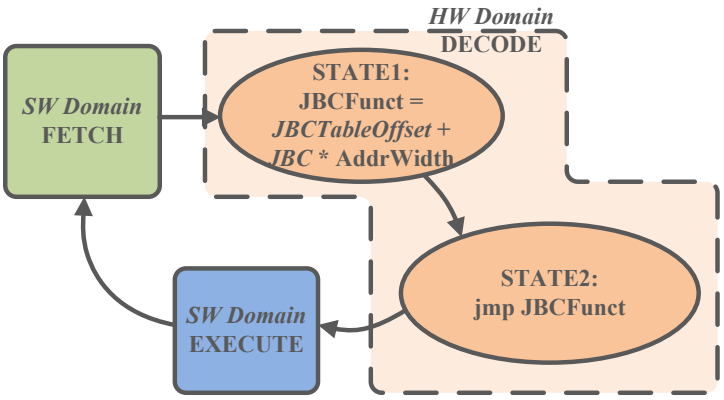

Figure 3.9: *Pseudo-threaded* Java interpreter with the decode phase in hardware (Obtained with modifications from [64]).

Figure 3.9 shows the finite state machine of the interpreter. As reported in the figure, the hardware architecture needs the addition of a register where the offset of the look-up table with the bytecode function addresses is stored. The value of the actual Java bytecode is taken as input. The resolution of the address within the look-up table is possible thanks to a dedicated adder that sums the offset of the look-up table with the Java bytecode value multiplied for the address width of the code memory. This multiplication is substituted with a simple shift operation in the case of a 16 bit memory width. Section 6.4 presents a more detailed description of the design [64].

3.3.3 Hardware Support for Fetch and Decode Phases

The second variation of the hardware support extends the functionalities discussed in Section 3.3.2 for the fetch phase. In the interpretation the fetch phase is based on the Java program counter (JPC) that, analogously to the program counter (PC) of a processor, contains the address of the next instruction to fetch. The Java bytecode fetch consists of reading one byte from the code memory and making it available for the decode phase.

Figure 3.10 sketches the state machine of the interpreter with the hardware support. The only part of the interpretation that remains in software is the execution phase that corresponds to the implementation of the bytecode functions. For the implementation of the hardware support the JPC is introduced in the hardware architecture and acts similarly to the PC, but for the Java bytecode. In addition to the elements already introduced in Section 3.3.2 for the decode, an additional internal register for the JPC and the logic needed for its management is needed. A deeper analysis of the hardware support with fetch and decode in hardware is present in Section 3.3.2 [64]. In the same section, an additional variant with a security check over the JPC is proposed [64]. Two additional

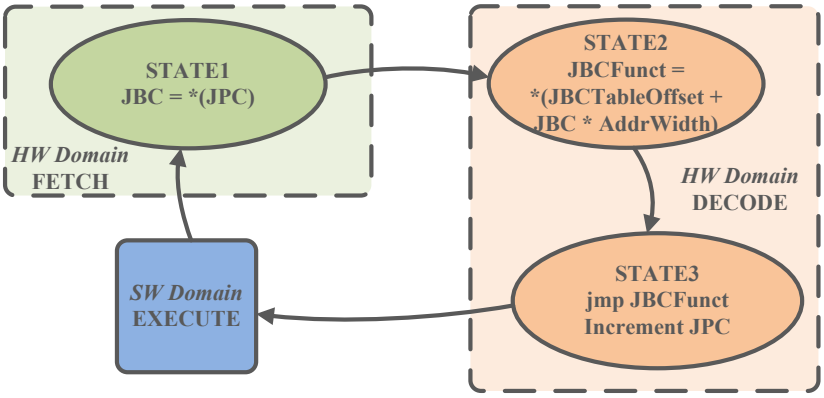

Figure 3.10: *Pseudo-threaded* Java interpreter with fetch and decode phases in hardware (Obtained with modifications from [64]).

registers contain the bounds within which the values of the JPC can range. If the JPC has a value out of bounds, an exception is thrown. Beyond two additional registers for the bound values, the security enhancement requires two comparators for the comparison of the JPC against its limits.

3.3.4 Hardware Support for Dictionary Decompression

In this section the final step for the integration of the interpreter with hardware support and the compression methodologies is presented. As reported in Section 3.2, the decompression techniques are integrated in the Java VM for the run-time decompression. Both folding and dictionary compressions can be integrated into the interpreter with hardware support transparently as with a classic interpreter. Section 6.5 gives a more accurate explanation of the integration.

From the point of view of the interpreter, the folding compression is an extension of the Java instruction set. Differently from folding compression, in the dictionary compression the interpreter handles the dictionary that consists of a look-up table containing pointers to the definitions of the dictionary macros as sketched in Figure 3.11. It is possible to create a hardware support for the dictionary decompression that accelerates the jumps through the macros look-up table.

The hardware support for the dictionary decompression is built over the hardware support presented in Section 3.3.3. It takes advantage of the hardware architecture of the fetch and decode phases, manipulating the JPC when a dictionary macro is encountered. The functionality implemented is similar to a *CALL* instruction in a microcontroller architecture. The state machine of the hardware functionality for the dictionary decompression is schematized in Figure 3.12. In the first state the JPC and the PC are stored in registers that will be used afterwards for continuing the execution flow. In the second state the PC is stored with the result of the addition between the offset address of the macro table and the actual macro value multiplied for the address width. This value in the PC corresponds to the address where the starting address of the actual dictionary macro definition

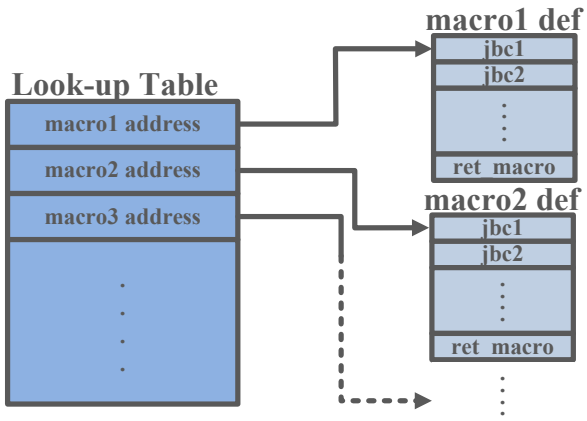

Figure 3.11: Representation of the structure of the dictionary containing the macro definitions (Obtained with modifications from [65]).

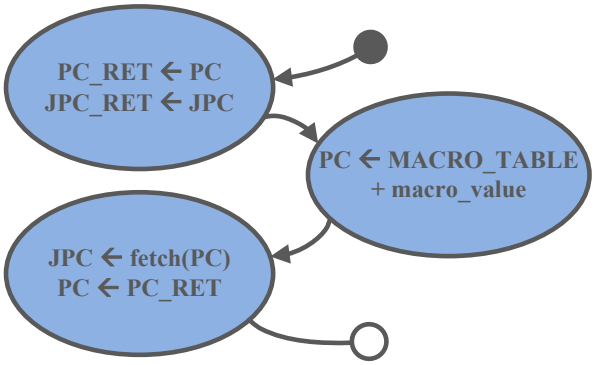

Figure 3.12: State machine of the hardware support for the dictionary decompression (Obtained with modifications from [65]).

is stored. In the third state the starting address of the macro definition is stored in the JPC, and the PC is restored with the value contained in the return register. After the interpretation of the macro content, the interpreter encounters a *RET_MACRO* instruction that restores the value of the JPC with the value contained in JPC_RET. Section 6.5 contains a more complete view of the hardware support for the dictionary compression [65].

3.4 Management of different variants

In this chapter many enhancements to the Java Card system are proposed. Their evaluation would request different builds of the software application (i.e. with the different

compression techniques, alone or combined, with different interpreters...) on different hardware platforms (i.e. a standard microcontroller platform, one with the hardware support for the interpreter, one with the hardware support for the dictionary decompression...). The result is a considerable number of variants of the Java Card system. To manage the different variants of the Java Card prototype, a variant management system, similar to the one used in industry for product lines management is adopted. All the aspects of the transition to a systematic product line management engineering are visited in Section 6.6 [66].

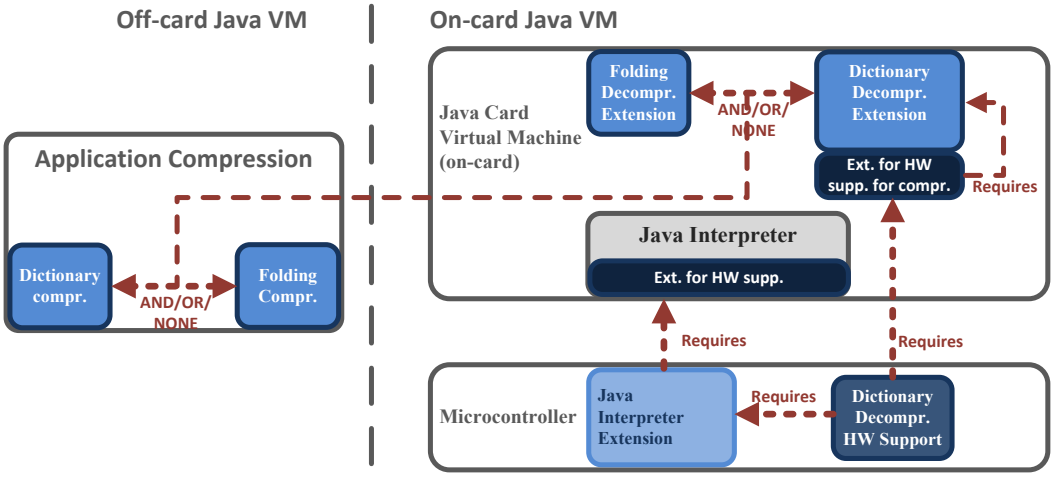

Figure 3.13: Scheme of the Java Card VM architecture under variant management.

The final prototype is shown in Figure 3.13. The compression methodologies can be activated singularly or together. The Java VM, and more specifically the interpreter, depends on the hardware architecture where they are built; indeed, the interpreter exploits the hardware enhancements presented in Section 3.3. The interpreter modules needed for the decompression are all independent from the hardware architecture, except for the decompression module that uses the features of the hardware support for the dictionary decompression.

Chapter 4

Results and Case Studies

This chapter gives an evaluation of the compression techniques and of the proposed hardware architectures. The first part relative to the compression techniques considers the space savings obtained and the influence that the compression has on the execution of the applications. The second and third parts of the chapter contain the assessment of the proposed hardware supports with particular attention paid to the execution time and implementation costs.

4.1 Compression

This section starts with the depiction of the evaluation workflow. It then analyzes in detail the performances of the compression techniques giving some insight into their implementations for the contextualization of the results.

4.1.1 Compression Evaluation Workflow

Figure 4.1 illustrates the evaluation of the compression techniques. At the top of the workflow there are the applications, which are diversified in industrial applications and test-benches. The industrial applications (MChipAdvanceed, MChip and XPay) are good tests for space saving since they consist of a very variegated code and hence they statistically give sound measurements. The test-benches (BubbleSort and BigInteger) are basically created for the evaluation of the execution performances.

The applications under test are compressed by means of a compressor developed in Java. The compressor contains both algorithms for dictionary (comprehending all the derived dictionary techniques for static and dynamic dictionaries) and folding compression. The pseudo-code for the compression techniques is available in Section 6.1 and Section 6.2. After the compression phase, the values of the space savings are available.

Finally, the application is deployed on the Java Card system for the measurement of the execution time of the applications. Only the test-benches are deployed, because the industrial applications make use of proprietary libraries not available in the source code of the Java Card reference implementation provided by Oracle [1]. The time measurements

[1] `www.oracle.com`

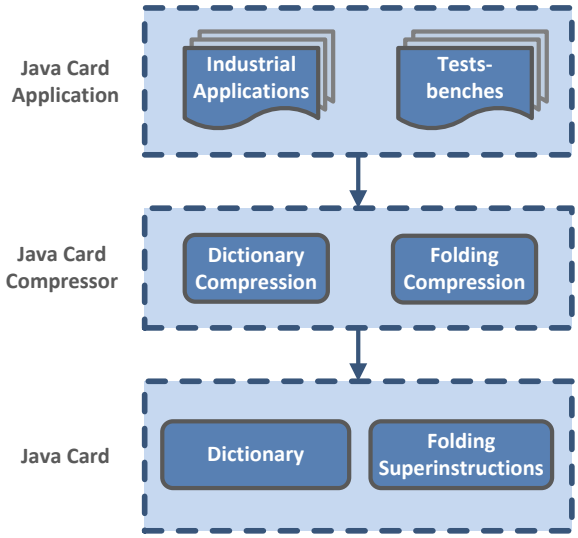

Figure 4.1: Java Card tool-chain for the evaluation of the code compression. The applications under test are compressed for quantifying the space savings and second for measuring the execution performance.

are performed running the Java Card environment on the simulator of the 8051 processor provided by the Keil μVision IDE [2].

4.1.2 Space savings

This section reports an analysis of the space savings obtained with the dictionary compression techniques and with the folding compression technique.

Dictionary Compression

As reported in Section 3.2.1, dictionary compression for Java consists of the substitution of repeated sequences of Java bytecodes with macros whose definitions are stored in a dictionary created apart. As described in Section 6.1, the implementation of the dictionary compression entails the introduction of a new Java bytecode (*ret_macro*) with the functionality of returning from a macro. Each macro definition has the *ret_macro* instruction as the final instruction. When accounting for space savings, beyond the space saved in the method component, the dictionary with all the macro definitions is taken into account.

Table 4.1 reports the space savings for the different dictionary compression techniques proposed in this thesis; the results are relative to the case in which a dynamic dictionary is used. The plain dictionary compression performs better with the dynamic dictionary. The techniques with the generalized macros (fourth and fifth columns) are more suited when a static dictionary is used and few dictionary entries are available [61]. Section 6.1 gives a more complete picture of the space savings, inclusive of the case with a static dictionary.

[2] `http://www.keil.com/`

Table 4.1: Space savings in the application after the application of the dictionary techniques

Application	Size [B]	Space Savings [%]			
		Plain Dict.	Gen. Arg.	Gen. Istr.	Subroutine
XPay	1784	12.3	5.4	9.5	4.0
MChip	23305	9.2	9.6	9.5	2.5
MChip Advanced	38255	10.5	8.9	8.6	3.9

Folding Compression

The concept and design of the folding compression is generally presented in Section 3.2.3 and with more detail in Section 6.2. The implementation of the folding compression needs the extension of the Java Card instruction set. The folding superinstructions constituting the extension of the instruction set are reported in Table 4.2. The extension set consists

Table 4.2: Instruction set extension for the folding compression (Obtained with modifications from [62]).

Instruction	Argument	Opt. Arg.
LdSt	B1[St:Ld]	-
PshSt	B1[Op:Cnst]	B2[BPsh]B3[SPsh]
OpSt	B1[St:Op]	-
LdIf_s2b	B1[Op:Ld]	B2[Br]B3[Brw]
LdPshAdd	B1[Cnst:Ld]	B2[BPsh]B3[SPsh]
LdPshOp	B1[Cnst:Ld]B2[Op:Ord]	B3[BPsh]B4[SPsh]B5[Br]B6[Brw]
LdLdOp	B1[Ld2:Ld1]B2[Op]	B3[Br]B4[Brw]
LdPshOpSt	B1[Cnst:Ld]B2[St:Op]	B3[BPsh]B4[SPsh]
PshLdOpSt(PPOC)	B1[Ld:Cnst]B2[St:Op]	B3[BPsh]B4[SPsh]
LdLdOpSt(PPOC)	B1[Ld2:Ld1]B2[St:Op]	-

of ten folding superinstructions. As can be seen, the argument of the superinstructions concerning the local variables are four bits long. Section 6.2 presents a coverage analysis of the foldable sequences and shows that the ten superinstructions cover about 95% of the possible sequences in the set of industrial applications [62, 63].

Table 4.3: Space savings obtained with the folding compression (Obtained with modifications from [63]).

Application	Size [B]	Space Savings [%]
XPay	1784	6.7
MChip	23305	4.0
MChipAdvanced	38255	3.7

Table 4.3 reports the space savings obtained with the folding compression. The data does not comprehend the space needed for the implementation of the instruction set extension. The implementation of the instruction set extension determines an increment of

about 5kB in the ROM size of the Java VM [62, 63].

Light-weight Compression

The publication in Section 6.3 contains an extended analysis of the light-weight compression. For the evaluation, the light-weight compression is implemented with the plain dictionary compression and the folding compression [63]. The space savings for the industrial applications are reported in Table 4.4. In comparison with the results of Table 4.1

Table 4.4: Space savings of the light-weight compression (Obtained with modifications from [63]).

Application	Space Savings [%]
XPay	15.7
MChip	12.4
MChip Advanced	11.7

and Table 4.3, the space savings of the light-weight compression are almost the sum of the space savings of the dictionary compression (plain version) and of the folding compression. The difference between the effective space savings of the light-weight compression and the sum of the singular space savings of the plain dictionary compression and the folding compression is given by the interference between the two techniques. This interference lowers the contribution of the dictionary compression [63].

4.1.3 Run-time Performances

This section investigates the execution performances of compressed applications. The tests are run on the test-benches BubbleSort and BigIntegers, two small applications that can be run on the Java Card reference implementation released by Oracle [3]. The Java card environment runs on a 8051 simulator, but the results can be replicated for any hardware architecture, since the performance comparison is dependent on the Java VM architecture and not on the hardware architecture.

Table 4.5 lists the space savings obtained with the dictionary compression, the folding compression and the light-weight compression. The space savings data are the indicator of the effectiveness of various compression techniques. It is possible to notice that the

Table 4.5: Space savings on the test-benches after compression (Obtained with modifications from [63]).

Application	Size [B]	Space Savings [%]		
		Dict. Compr.	Fold. Compr.	Light-w. Compr.
BubbleSort	239	5.4	2.5	6.7
BigInteger	650	3.4	1.5	4.5

dictionary compression is not as effective as in the industrial applications, because of

[3] `www.oracle.com`

Table 4.6: Execution time of the compressed test-benches. The results are in comparison with the execution time of the non-compressed test-benches (Obtained with modifications from [63]).

Application	Execution Time [%]		
	Dict. Compr.	Fold. Compr.	Light-w. Compr.
BubbleSort	+3.2	-6.8	-3.8
BigInteger	+1.7	-4.0	-2.2

the small size of the applications and therefore the lower probability of finding repeated sequences.

Table 4.6 reports the execution time increment and decrement in the different compression cases compared to the execution time of the non-compressed applications. As expected, the dictionary compression slows down the execution, whereas the folding compression speed it up. In the light-weight compression, the execution speed is the result of the contrasting behaviors of the dictionary compression and of the folding compression. In the examined cases, applications compressed with the light-weight compression are executed faster than the non-compressed applications, because of the low effectiveness of the dictionary compression and of the dominant effect of the folding compression. In the case of industrial application, where the dictionary compression has a higher impact, the execution of an application compressed with the light-weight compression is expected to be slower than the execution of a non-compressed application, but in any case faster than in the case of an application compressed only with the plain dictionary compression.

4.2 Hardware Support for the Java Interpreter

This section regards the evaluation of the hardware support for the Java interpreter. After the work-flow is clarified, the performance improvements and the costs are analyzed. A classic interpreter is compared to the interpreter with the pseudo-threaded architecture realized in three variants:

- Full software solution
- With hardware support for the decode phase of the Java bytecode
- With hardware support for the fetch and decode phases of the Java bytecode

The classic architecture and the pseudo-threaded interpreter run on an 8051 architecture. The interpreter with hardware support for the decode phase and the interpreter with hardware support for both fetch and decode phases run on the 8051 architecture extended with decode of the Java bytecode in hardware and with fetch and decode of the Java bytecode in hardware, respectively. Moreover, the results for the interpreters are proposed as a solution with and without bounds check, as discussed in detail in Section 6.4 [64]. The proposed variants are evaluated for ROM size, execution performance and hardware costs.

4.2.1 Workflow for the Hardware Support Evaluation

The workflow used for the evaluation of the hardware support is shown in Figure 4.2. Starting from above, the Java interpreter is written in C and assembler code. The assem-

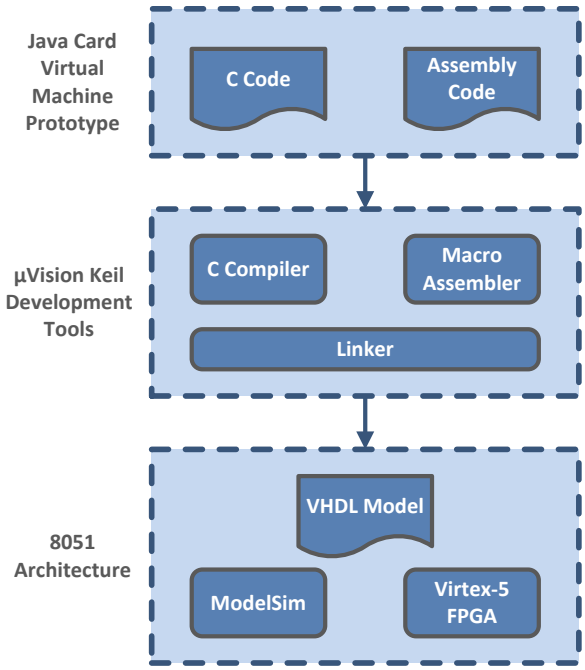

Figure 4.2: Tool-chain for the evaluation of the hardware support for the Java interpreter.

bler code is necessary especially for accessing the new hardware functionalities by means of an extension of the processor instruction set [64]. The commercial tool-chain supplied by Keil (Keil μVision [4] is used for the compilation of the interpreter. The hardware is described by a VHDL model of a 8051 core architecture developed by Oregano [5] in collaboration with the Vienna University of Technology and available under LGPL. The model is synthesized with the Xilinx ISE tool-chain [6]. The executable ROM created with the Keil tool-chain is synthesized together with the 8051 IP core and run on the ISE Simulator ISim and on a FPGA starter-kit (Xilinx Virtex-5 FXT FPGA ML507).

4.2.2 Hardware Overhead

For the realization of the hardware support, the 8051 architecture is extended. The original architecture consists of a finite state machine (FSM) that describes each op-code instruction of the 8051 architecture. The hardware support for the interpreter is inserted in the main FSM and the additional hardware registers in the internal memory of the architecture. Additional details concerning the hardware realization are available in Section 6.4

[4] http://www.keil.com/
[5] http://www.oreganosystems.at/?page_id=96
[6] http://www.xilinx.com

[64].

Table 4.7: FPGA utilization for the different architectures with hardware support for the pseudo-threaded interpretation (Obtained with modifications from [64]).

Architecture	FPGA Util.			
	FFs	Diff. %	LUTs	Diff. %
Std8051	582	-	2623	-
DI8051	597	2.6	2721	3.7
FDI8051	614	5.5	2885	10.0

Table 4.7 reports the FPGA consumption after the deployment of the different architectures. The values are expressed in terms of flip-flops (FFs) and look-up tables (LUTs). The architectures with hardware support for the decode (DI8051) and for both fetch and decode (FDI8051) are compared to the standard 8051 architecture (Std8051).

4.2.3 ROM Size of the Interpreter

Access to the new functionalities is made possible by an extension of the instruction set of the 8051 architecture. A deeper understanding of the new instructions can be obtained in Section 6.4. The pseudo-threaded architecture needs a final sequence of instructions for each function that is implementing a Java bytecode that allows the fetch and the decode (comprehensive of the jump to the next Java bytecode function) of the next Java bytecode [64].

Table 4.8: ROM memory size of the code for the fetch and decode phases (Obtained with modifications from [64]).

Interpreter	ROM Size [B]	
	w/o JPC Check	w/ JPC Check
CWI	241	283
PTCI	9812	17108
PTCIHwD	4888	12784
PTCIHwFD	376	376

Table 4.8 lists the ROM size of the code for the fetch and the decode phases in the Java interpreter. The results are relative to the interpreters with and without the bounds check [64]. The data in the table shows the main limit of the pseudo-threaded interpreter: the code size for the version completely in software (PTCI) is huge compared with the code size for the classic version (CWI). The code size of the pseudo-threaded interpreter with the hardware support for the decode phase (PTCIHwD) is smaller than the one completely in software, but still big compared to the classic interpreter. The pseudo-threaded interpreter with fetch and decode phases in hardware (PTCIHwFD) has a code size similar to the classic interpreter. In the PTCIHwFD interpreter, the code size for the versions with and

without the bounds check is the same because the bounds check is executed in hardware and hence there is no code to be added.

4.2.4 Execution Speed

The new architecture of the interpreter allows time reduction for the fetch and the decode phases. The execution phase of the interpretation remains the same for all the proposed Java interpreters. In Table 4.9 the times needed for the fetch and the decode phases are reported. The table collects data for the interpreters with and without bounds check and reports the percentage difference with the classic interpreter. As can be easily inferred

Table 4.9: Times for the fetch and decode phases in the proposed interpreters (Obtained with modifications from [64]).

Interpreter	Run-time [Clk Cycles]			
	w/o JPC Check	Diff. %	w/ JPC Check	Diff. %
CWI	76	-	138	-
TCI	60	-21	122	-12
TCIHwD	38	-50	100	-28
TCIHwFD	6	-92	6	-96

from the data in the table, the pseudo-threaded interpretation gives a better performance in general than the classic interpretation. Analyzing the variant without the bounds check, the comparison between the classic interpreter and the pseudo-threaded interpreter realized completely in software shows a reduction in time of 21%. The reduction in time increments to 50% when the hardware support for the decode phase is used, and to 96% when the fetch phase is also performed in hardware [64].

The fetch and the decode phases are the only common part of the interpretation for each Java bytecode. The rest is represented by the execution phase that is different for each Java bytecode, since it consists of the functionality of that Java bytecode. For an estimation of the impact of the new architectures on the interpretation, a set of most frequently interpreted Java bytecodes is taken into consideration. The interpretation times are averaged and drawn in the histogram of Figure 4.3. The execute time is constant for each architecture taken into consideration, because the code for the execute phase does not use the hardware support. As can be seen, the contribution of the fetch and decode phase time of the classic interpreter (CWI) is relevant and counts for almost 50% of the overall time. The histogram clearly shows the benefit of the new interpreters, especially of the ones with hardware support: the Java interpreter with hardware support for the fetch and decode phases (TCIHwFD) presents a reduction in the overall interpretation time of 41% compared with the classic interpreter (CWI) [64].

Similar considerations can be made for the energy consumption needed for the Java bytecode interpretation, due to the fact that it heavily depends on the execution time. Section 6.4 contains a complete evaluation of the energy consumption needed for the Java bytecode interpretation.

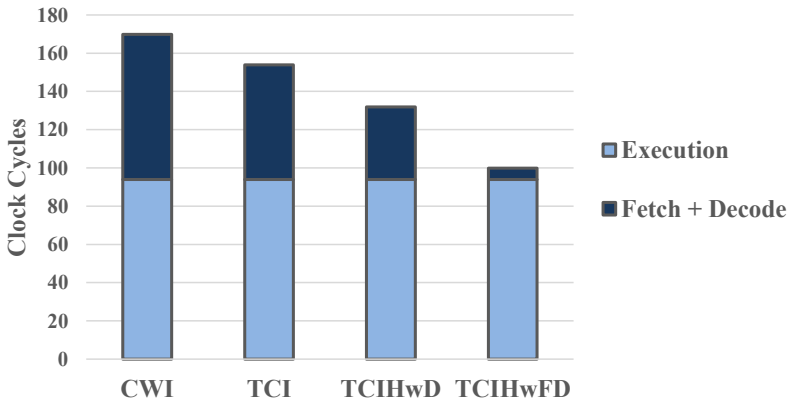

Figure 4.3: Interpretation time for the Java interpreters. The execute time is in light-blue, the fetch and decode phases in dark-blue (Obtained with modifications from [64]).

4.3 Hardware Support for the Dictionary Decompression

This section reports the results for the hardware support for the dictionary decompression. The workflow for the evaluation of the hardware support for the dicionary decompression is the same as that described in Section 4.2. The dictionary decompression is implemented for three architectures:

- The classic interpreter based on a while loop
- The interpreter based on the hardware support for the fetch and decode phases
- The interpreter based on the hardware support for the fetch and decode phases with the hardware extension for the dictionary decompression

The hardware costs and the benefits in terms of execution time are discussed for the three proposed architectures.

4.3.1 Hardware Overhead

The hardware support for the dictionary decompression is built over the Java interpreter with fetch and decode phases in hardware. To compare the hardware overhead, Table 4.10 reports the data relative to the standard 8051 architecture, to the variant with fetch and decode in hardware and to the model with the hardware support for the dictionary decompression. Since the hardware realization has been implemented on a FPGA board, the hardware overhead is expressed in terms of flip-flops (FFs) and look-up tables (LUTs). The increment in the FPGA utilization observed in the architecture with the hardware support for the dictionary compression is due to the addition of the registers for handling the look-up table containing all the addresses of the macro definitions. The complete discussion regarding the hardware overhead due to the introduction of the support for the dictionary decompression can be found in Section 6.5 [65].

Table 4.10: FPGA utilization for the different architectures

Architecture	FPGA Util.			
	FFs	**Diff. %**	**LUTs**	**Diff. %**
Std8051	582	-	2623	-
FDI8051	614	5.5	2885	10.0
FDI8051DEC	666	14.4	2946	12.3

4.3.2 Performance Improvement

The dictionary decompression is implemented in the three architectures proposed for the Java interpreter. For the assessment of the execution time performances, the time for the interpretation of a macro with the average length of three bytecodes with the addition of the *ret_macro* is measured. The average length of the dictionary macro is determined during the compression with the plain dictionary technique of the industrial applications already used in Section 4.1.2.

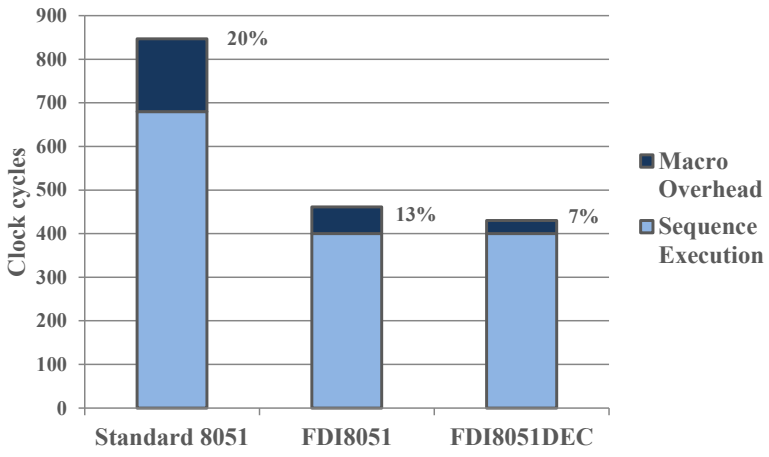

Figure 4.4: Execution time for the interpretation of an average dictionary macro. The execution times for the execution of the bytecodes constituting the macro are In light-blue, and the execution times for the macros overhead are in dark-blue (Obtained with modifications from [65]).

The histogram in Figure 4.4 reports the time measurement for the complete interpretation of a dictionary macro. The time for the interpretation of the Java bytecodes contained in the macro is colored in light-blue. As can be observed, the time for the interpretation of the Java bytecodes in the definition is 41% lower in the architectures with hardware support (FDI8051 and FDI8051DEC); the decrease in execution time is due to the hardware support for the interpreter as discussed in Section 4.2. The dark-blue part

of the bars represents the time needed for the dictionary macro overhead that consists of two contributions:

- The interpretation of the macro bytecode
- The interpretation of the *ret_macro* at the end of the macro definition

The graphic shows that the hardware support reduces the time overhead for the macro encapsulation from 13% (in the case of FDI8051) to 7% (FDI8051DEC). A more exhaustive exposition of the results regarding the hardware support for the dictionary decompression is in Section 6.5 [65].

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Java Card technology is a subset of the Java technology tailored for smart cards. The penetration of Java Card in the smart card market is growing continuously, because smart card producers are converging on a solution that offers a common platform where third parties can also develop applications. In the wide field of possible applications of Java Card that comprises of telecommunications, banking and identification, a diversification of the products is taking place in terms of performance. Therefore, aside from high-end smart cards, there is a growing sector of low-end smart cards equipped with scarce resources. In order to minimize the applications footprint, compression techniques based on the use of a dictionary are proposed in literature. In other research works the absence of adequate resources with which the JIT compilation can be applied is compensated with techniques that range from superinstructions for the Java virtual machine to the implementation of the Java virtual machine in hardware.

This thesis addresses two issues: the footprint of the Java Card applications and the execution performance of the Java virtual machine. Beyond saving some space in the code memory, compressing an application means introducing some time delay in the execution of the application because of the decompression. This work goes further and, instead of looking at compression as a technique that irremediably slows down the application execution, tries to alleviate this drawback with new compression methodologies and with hardware/software co-design approaches.

The first contribution is represented by an accurate examination of dictionary compression for Java Card. The use of the static and dynamic dictionaries in relation to the Java Card architecture is evaluated. Moreover, two new dictionary compression techniques based on generalized macros are introduced. The plain dictionary compression performs better in the case of dynamic dictionary, but the dictionary techniques with generalized macros are a good alternative when the dictionary is static and the number of dictionary entries is small. This thesis also presents a compression technique based on the folding mechanism, an optimization for the execution of Java bytecodes. Beyond a discrete space saving, the folding compression allows an increase in the speed of the application execution. A combination of the folding compression and the dictionary compression is proposed as a light-weight compression technique. The slow-down effect of the dictionary

compression is compensated by the speed-up introduced by the folding compression; the contributions produced by the two techniques in space savings are concurrent and allows an overall space saving that is about the same as the sum of the two contributions.

To mitigate the slow-down effect of the dictionary compression and to speed up the overall interpretation speed, the second part of this thesis proposes a new Java interpreter architecture that incorporates the fetch and the decode phases at the end of each function implementing the Java bytecodes. By means of a hardware/software co-design approach, the fetch and decode phases are moved into silicon creating a hardware support that dramatically speeds up the interpretation speed, at the cost of little hardware overhead. To complete the picture, dictionary compression is integrated in the new architecture having the Java interpreter with hardware support. In the first instance the dictionary decompression is realized in software, and the decompression takes advantage of the hardware support for bytecode interpretation. In the second instance the part of dictionary decompression through the dictionary look-up table is moved into hardware further speeding up the dictionary decompression.

5.2 Directions for Future Work

5.2.1 Security in Java Card enabled for Decompression

The implementation of compression techniques increases the complexity of the Java virtual machine. An interesting topic for future work regards the security issues introduced by the compression techniques. An example can be found in dictionary compression, where the management of the dictionary causes the Java program counter to point out of the method area. Security countermeasures against potential attacks can be designed following on the works [67, 68]. Security issues arise also in the hardware decompression unit introduced in [65], where the possible countermeasures can be implemented in hardware analogously to [68].

5.2.2 Java Stack Compression and Hardware Support for Operands Stack

The Java virtual machine is a stack machine. The Java stack is a fundamental part of the virtual machine and the information regarding the execution is stored in its internals. The stack is organized in frames; each method has its own frame that is active when the method is in execution. In each stack frame there is an *operands stack* that is used as working memory by the method owner of the stack frame. Because of the frequent use of the operands stack, approaching the problem with a hardware/software co-design can improve the performance on accessing it. A flexible design similar to the one proposed in [64] gives the software developers the freedom to add additional features such as, for example, security countermeasures. Another research direction is represented by the stack frames compression. The inactive stack frames may be singularly compressed to reduce the required memory and to allow the singular decompression at the moment of the reactivation. To limit the time overhead due to the on-card compression and decompression, hardware/software co-design techniques can be implemented to improve the compression and decompression speed.

5.2.3 Heap Compression in Java Card

The Java heap is a portion of memory where all the objects created during the execution of a Java application are allocated. In standard Java, there is a component of the runtime environment called garbage collector, whose functionality consists of deleting the objects that are no longer reachable by any other allocated object. The garbage collector implements specific algorithms for taking decisions about which objects to delete. When the garbage collector deletes an object, the remaining objects can be allocated in such a way that the memory is highly fragmented. At the end of the object deletion, the garbage collector provides for the compaction of the remaining objects such that the free memory is not fragmented anymore and can be entirely utilized. In this phase, it is possible to further reduce the memory occupation of the objects by means of compression of the Java heap [69, 70]. In Java Card, the garbage collector is not feasible because of the scarce resources of smart cards: there are manual mechanisms for the explicit deletion of the objects. In the context of Java Card there would be an interesting investigation in the field of the heap compression, also evaluating the opportunity for hardware support.

Chapter 6

Publications

This chapter presents the collection of publications which were created during this thesis. The publications explain the related work, methodology, results, and contributions of this thesis in more detail.

Publication 1: *On the dictionary compression for Java card environment*, Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems (M-SCOPES '13), St. Goar , Germany, 2013.

Publication 2: *Instruction Folding Compression for Java Card Runtime Environment*, 17th Euromicro Conference on Digital Systems Design (DSD), Verona, Italy, 2014.

Publication 3: *A light-weight compression method for Java Card technology*, The 4th Embedded Operating Systems Workshop (EWiLi), Lisbon, Portugal, 2014.

Publication 4: *A High Performance Java Card Virtual Machine Interpreter Based on an Application Specific Instruction-Set Processor*, 17th Euromicro Conference on Digital Systems Design (DSD), Verona, Italy, 2014.

Publication 5: *An Application Specific Processor for Enhancing Dictionary Compression in Java Card Environment*, 5th International conference on Pervasive and Embedded Computing and Communication Systems (PECCS), Angers, France, 2015.

Publication 6: *Embedding Research in the Industrial Field: A Case of a Transition to a Software Product Line*, International Workshop on Long-term Industrial Collaboration on Software Engineering (WISE), Vasteras, Sweeden, 2014.

The publications in this chapter discuss various HW/SW co-design aspects for a high performance Java Card enabled for compressed applications. This secure Java Card enables the secure installation and execution of different applications. Various checks and countermeasures against attacks are proposed in this thesis. The checks and countermeasures are either implemented in SW or HW.

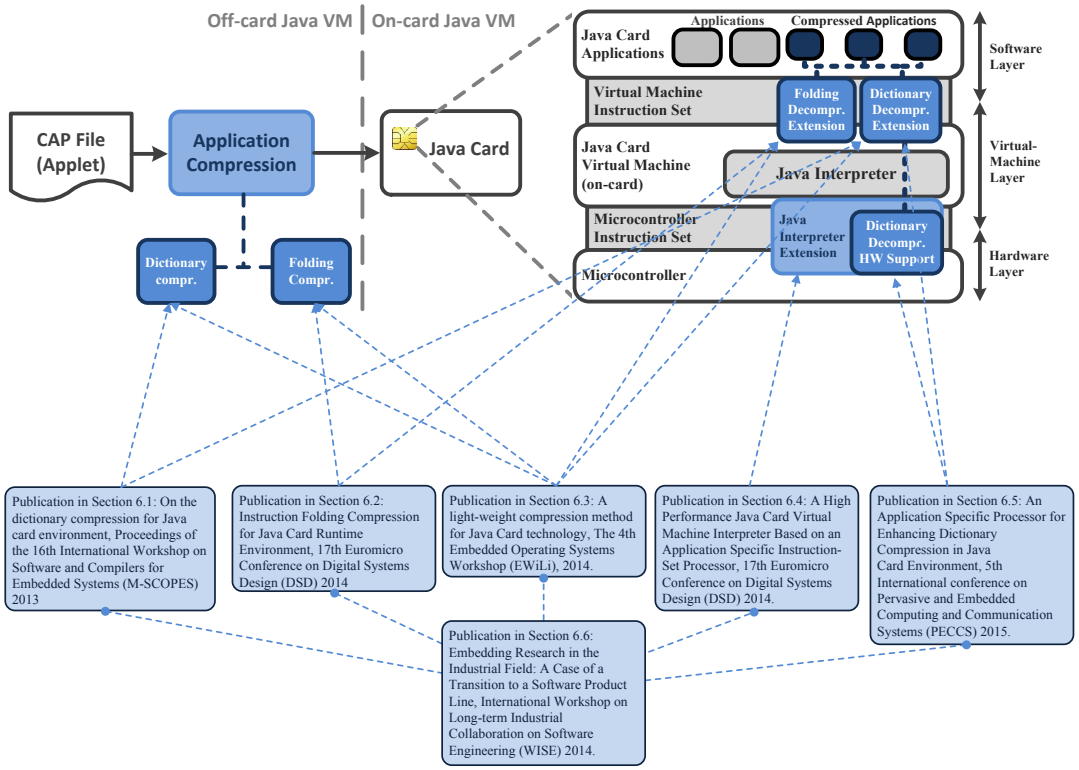

Figure 6.1: Publications overview for a high performance Java Card enabled for compressed applications.

The publications in the first three sections regard code compression techniques:

- The publication in Section 6.1 presents a complete analysis of dictionary compression, considering the use of static and dynamic dictionaries and proposes advanced dictionary techniques that use generalized macros
- The folding compression technique is presented in Section 6.2. It consists of an extension of the Java instruction set with superinstructions used for substituting foldable sequences,
- Section 6.3 proposes a light-weight compression consisting of the combination of dictionary compression and folding compression.

The publication in Section 6.4 describes a hardware support for the Java Card interpreter. The publication is structured into two main parts: the first presents a new

architecture for the interpreter that is used in the second part for the develop of the hardware support. Section 6.5 proposes a hardware support for the run-time decompression of dictionary compression; the enhancements in this publication are based on the work in Section 6.4. In Section 6.6 a transition to product lines engineering is presented. The work describes the same problems encountered in the creation of a single building system under variant management for the Java Card system enabled for compression.

M-SCOPES 2013

On the Dictionary Compression for Java Card Environment

Massimiliano Zilli
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
massimiliano.zilli@tugraz.at

Wolfgang Raschke
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
wolfgang.raschke@tugraz.at

Johannes Loinig
NXP Semiconductors Austria GmbH
Gratkorn, Austria
johannes.loinig@nxp.com

Reinhold Weiss
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
rweiss@tugraz.at

Christian Steger
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
steger@tugraz.at

ABSTRACT

Java Card is a Java running environment developed for low-end embedded systems such as smart cards. In this context of scarce resources, ROM size plays a very important role and compression techniques help reducing program sizes as much as possible. Dictionary compression is the most promising technique and has been taken in consideration in this field by several authors.

Java Card can adopt a dictionary compression scheme, substituting repeated sequences of bytecodes with new macros stored into a dictionary. This approach does not break the Java Card standard, but requires the use of an ad hoc Java virtual machine and an additional custom component in the converted applet (CAP) file. This paper presents two derived compaction techniques and discusses two scenarios: the first adopts an adaptive (dynamic) dictionary, while the second uses a static one. Although the base dictionary compression technique performs better with an adaptive dictionary, the two proposed techniques perform very close to the base one with a static dictionary. Moreover, we present a different compression mechanism based on re-engineering the CAP file through subroutines. This last technique achieves a higher compression rate, but it is fully compliant with the existing Java Card environments.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software architectures; D.3.4 [**Programming Languages**]: Processors—*Compilers Interpreters Optimization*; E.4 [**Coding and Information Theory**]: Data compaction and compression

M-SCOPES '13, June 19 - 21 2013, Sankt Goar, Germany

General Terms

Algorithms, Design

Keywords

Compiler, smart card, Java Card, dictionary compression

1. INTRODUCTION

Smart cards are low-end embedded systems with limited resources in terms of computing power, RAM and ROM storage [11]. Typical smart cards are based on 8/16 bit processor with one kilobyte of RAM and include up to few hundreds kilobytes of ROM. In such small devices the common practice is to develop applications in low level programming languages such as C or assembly in order to keep the code size small and the performance high. Problems with this approach are due to the difficulties in updating, reusing, and porting the code. A high level language like Java partially solves this set of problems, while also adding valuable security features to the runtime environment, aspects that are very important to most smart-card applications.

Due to resource constraints, smart cards cannot utilize the complete Java standard. However, it is possible to overcome this limitation using a subset of the language, called Java Card. Java Card enables programmers to take advantage of Java's object-oriented programming model and offers a common infrastructure for developing applications independent from the chosen hardware and software platforms [3]. Applications are distributed in the Java Card converted applet (CAP) file format [9], where all the classes utilized by an application package are stored. This file contains several components used during the installation, including the *Method Component* that contains all the classes' methods and typically occupies up to 75% of the entire CAP file. Another advantage of Java Card is security [8]. Java Card specification is well defined and assures a security environment where to protect sensitive information. Hence, modifications to the current model are dangerous since they may break standard security features.

One of the goals in embedded systems development is to keep ROM size as small as possible. Popular code com-

pression techniques usually demand resources that are not available in nowadays smart cards, as for instance they may require to store in RAM part of the program code for its decompression and successive execution [15]. Dictionary compression for smart cards is the most suitable compression technique, since it does not need large amount of resources in the decompression phase. This technique consists in substituting repeated sequences with new macros. The compression phase consists of the search of repeated patterns later stored in a dictionary and their substitution with macros. These macros are afterwards decoded in the decompression phase looking them up into the dictionary, from where the code can be directly executed without any need of additional translation phase.

Clausen et al. [4] have used dictionary compression and showed that a space savings up to 15% is reachable. A repeated sequence of Java bytecodes is substituted with a new bytecode. In the decoding phase the Java virtual machine translates the new bytecode looking up into a dictionary stored in an additional CAP file custom component [2]. The extension of the virtual machine with new bytecodes does not conflict with the standard. Some bytecodes are not utilized in Java Card specification [9] and thus they can be used for the extension.

This paper investigates two new techniques derived from the base dictionary one. The first uses wildcards to generalize the arguments of the java bytecodes, whereas the second introduces into the definitions of the dictionary generalized instructions that are fully specified by additional arguments. We evaluate these techniques in two scenarios, one with a dynamic dictionary and the other with a static one. Even though we will see that the base dictionary technique performs better with dynamic dictionary, we will also see that the two proposed techniques perform close to the base dictionary one with a static dictionary. Finally we propose a third approach that avoids extending Java Card base standard, and instead integrates the dictionary compression without adding new bytecodes. This last technique allows for smaller CAP file while also complies with all existing Java Card virtual machines. Space savings are between 2.5 and 4.0%.

The rest of the paper is structured as follows. Section 2 resumes the state of the art in the dictionary compression focusing on the Java Card area. In Section 3 we present our variants to classic dictionary compression and we propose a compression method by means of subroutines. Results from our tests and relative discussions are reported in Section 4. Finally in Section 5 we report our conclusions with a view on future work.

2. RELATED WORK

Dictionary compression is based on the encoding of symbols' strings into tokens using a dictionary [12]. Dictionaries may be static or dynamic (adaptive); the former is permanent, whereas the latter is created from the input analysis. In general the adaptive dictionary is the best choice, since it allows a better compression, but there are situations where a static dictionary is preferable or necessary, such as when it is not possible to send the dictionary with the application. A statistical compression like Huffman [12], in general has the drawback of a more complex decodification phase. This complexity is especially critical when the compression regards executable code, where the complete code, or parts of it, have to be decoded before execution. In a context where there is a small amount of available RAM and short execution time has a relevant importance, this kind of compression is not suited. A dictionary compression where the new introduced symbols can be easily translated jumping through look-up tables is a better solution for these kinds of systems.

A virtual machine (VM) is a software entity whose purpose is to interpret compiled code. Virtual machines usually execute applications slower than processors executing native compiled applications. Among the reasons for using a virtual machine there are the target independence and the smaller size of the compiled application. Proebsting [10] introduced the concept of *Superoperators* for bytecoded interpreters. These are sequences of instructions that can be grouped together to optimize execution time, and, as secondary effect, to reduce code size. Superoperators are introduced into VM instruction set and then found and substituted into the application code.

The work of Clausen et al. [4] presents for the first time an application of a dynamic dictionary compression method to the Java bytecodes. The application of this technique is taken in consideration for tiny embedded system such as Java Card. The compressor analyzes the method component of the CAP file and searches for repeated occurrences of patterns. These patterns are stored in a custom component of the CAP file [2] and every occurrence is substituted with a new bytecode. The technique is able to reach about 15% of compression with the drawback of the increase in execution time between 5 and 30%.

The approach of Saougkos et al. in [13] is based on the work of [4] and expands the patterns searching policy. In fact whereas in [4] the research is after patterns with fully specified sequence of instructions, in [13] a variable number of non-specified bytecodes, called wildcards, are introduced in the patterns. The problem of finding the set of possible sequences is handled by means of the agglomerative clustering technique. Heuristic algorithm are then used for finding the sets of patterns for the compression. The outcome is that non-parameterized and with small number of wildcards patterns are the best choices for most of the applications taken in exam.

3. DESIGN AND IMPLEMENTATION

From theory of compilers a *Basic Block* is a sequence of consecutive statements in which the control flow enters at the first statement and leaves at the last one, without halt or any possibility of branch except at the end [1]. This definition is useful for the creation of flow graphs but in the context of dictionary compression it is too strict. In fact factorizing a program under this definition creates a large number of small blocks. For this reason we introduce in our factorization model the definition of a pseudo basic block that extends the original definition making it suited for our purpose. In literature this entity is known as *Single Entry Single Exit (SESE)* region [6]. As shown in Figure 1 in a SESE the last statement cannot be a branch instruction.

Moreover the content of the region does not need to be linear, allowing the presence of internal branches having as target internal statements. External branches cannot enter into the block except into the first statement (entry is single); internal branches cannot go outside the block (this condition does not comprehend *return* and *invoke* instruc-

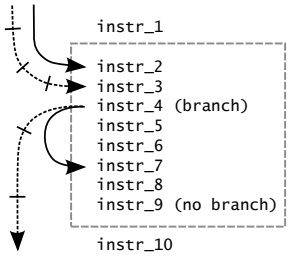

Figure 1: Single Entry Single Exit Block

```
Collection collections;

for( i=0; i<instructions.size(); i++ ) {
  Sequence seq;

  seq = instructions[i]; //init seq
  for( int j=1; j<MaxWindowSize-1; j++ ) {
    seq += instructions[i+j];
    if( isSeqCollected(seq) && isSeqSESE(seq) ) {
      collections += searchSESESeqInTheCode(seq);
    }
  }
}
```

Figure 2: Pseudo-code for the collecting algorithm

tions inside the block). Basing on SESE concept, blocks can have every dimension, theoretically until the entire program. However for the needs of a dictionary compression method, a reasonable limitation must be taken since the bigger the block is the smaller is the possibility to find its repetitions through the code.

In general code factorization is an NP-complete problem [5]. However our algorithm relies upon a simple moving window search that fits well to the faced problem. In Figure 2 we show the pseudo-code for the search algorithm. Starting from one instruction, the window extends until it reaches its maximum width. The latter must be of a reasonable size; in our experiments we see that patterns bigger than 20 instructions are very rare. For this reason the usage of larger windows is not convenient because they would extend the searching time enormously without any advantage. Every time that we include a new instruction in the window, the sequence of instructions contained is proposed as a new pattern. The first step consists in checking if it has already been collected, if not it is used for collecting equivalent patterns through the entire code into a group of equivalent patterns. Every equivalent pattern found, before adding it to the collection group, is checked for compliance with SESE characteristics, and discarded otherwise.

In Figure 3 we report the description in pseudo-code of the phase that associates new bytecodes to the discovered patterns. After collecting all possible patterns, a first purging is executed, with the goal of eliminating all those groups that consists of only one sequence and so not worth compressing. The successive step is to check the overlapping among every occurrence in every group. This information is

```
Collection collectMacros;

purgeSinglGroups(collections);
calculateOverlapping(collections);
collectMacros = getMostSavCollect(collections);
int i=0;
do{
  calcSpaceSaving(collectForMacros);
  collectMacros += getMostSavCollect(collections);
        i++;
}while(i<MaxNMacro || saveCollect(collections)!=0)
```

Figure 3: Pseudo-code for the decision algorithm

used in the following steps to give a value to every group in relation to the others. The selection of the groups that are worth representing with a new bytecode is not as simple as taking groups with more than one occurrence inside. In fact every occurrence in a group could overlap and so exclude an occurrence in another group. These relationships make the decision of which group to select not trivial.

A metric for the sake of taking decision is necessary and in this compaction context the number of saved bytes is the chosen one. In a group, every occurrence that does not overlap with others of an already chosen group, is substituted into the code with a new bytecode and the pattern is written only once into the dictionary component and terminates with the special *return from macro* instruction that brings back the execution flow to the method component. The new bytecode can be represented with a single byte (until the number of free bytecodes in the standard is reached) or two and more. Generally the undefined bytecodes can be used for several functionalities (i.e. additionally security) and using all of them for the compaction could not be possible. In our experiments we substituted only the first 10 groups with a single byte instruction and the remaining groups with two-byte instructions. In this way we let several undefined bytecodes for other possible purposes.

The decision process depends on the potential saved bytes of every group, in turn dependent on the overlapping sequences among the groups. For this reason the decision process needs to select one group as a starting point. We choose to select as first group the one that could save the most storage on memory. Once we selected an initial group, we can find in the remaining groups all the sequences that overlap with the ones of the initial group. The overlapping sequences are excluded in the computation of the saved bytes of the remaining groups. This process is iterated until a determined number of groups (and then a fixed number of new bytecodes) is reached or there are not any other groups that allow saving bytes. In general every group represents a different macro that shall be saved in an additional CAP file component where the virtual machine will be able to find them. From now on in this document we will call this compression mechanism *Plain Dictionary Compression*(PDC).

We show now by means of the very simple Java class in Figure 4 an example of how the technique is applied. In Figure 5 we report the relative compiled bytecode through which the three macros reported in Figure 6 have been found. We show the calculation of the saved space taking in consideration the first macro. In the original bytecode the in-

```
public class Point {
  private short x;
  private short y;
  private short z;
  public Point() {
    x=0;
    y=0;
    z=0;
  }
  public Point(short x, short y, short z) {
    this.x=x;
    this.y=y;
    this.z=z;
  }

  public short absXDist(short x) {
    return absDiff(this.x, x);
  }
  public short absYDist( short y ) {
    return absDiff(this.y, y);
  }
  public short absZDist( short z ) {
    return absDiff(this.z, z);
  }

  public short absDiff(short a, short b) {
    short diff;
    if( a > b ) {
      diff = (short)(a - b);
      return diff;
    }
    else {
      diff = (short)(b - a);
      return diff;
    }
  }
```

Figure 4: Example of a class file

structions sequence is 5 bytes long and it is repeated 2 times through the code. This sequence is stored into the dictionary adding the *macro_return* instruction in a 6 bytes long definition. The macro is represented with a new bytecode one byte long. The substitution with the new macro takes 2 bytes since it is repeated 2 times. The first macro finally lets save 2 bytes. Similarly, the second and the third macros save 6 and 1 bytes, respectively. In this paper we henceforth define the compression ratio as

$$r = \frac{S_{compressed}}{S}$$

where S is the size of the original size of the method component of the application, and $S_{compressed}$ is the size of the method component of the application after the compression. Complementary the space savings is defined as

$$s = 1 - r = 1 - \frac{S_{compressed}}{S}$$

In the example the three macros allow a total of 9 bytes saved, that, respect to an original amount of 71 bytes, means about the 12.7% of space savings (s) and hence 87.3% of compression ratio (r). To avoid misunderstanding we underline that we are aware that the function *absDiff* is redundant in its content, but it is written in that way for the sake of simplicity and to show that the compaction techniques can also be a tool against bad code style.

3.1 Variations to plain dictionary

Our work proposes some variation to the PDC, for the sake of making comparisons in a qualitative and quantitative aspect. The algorithm previously described in the previous section is adapted with opportune modifications for fitting these variations.

3.1.1 Dictionary Compression with Wildcards

Saougkos et al. in [13] proposed a variation to the method proposed in [4] which introduces the concept of *wildcards*. In a bytecode sequence using wildcards consists in not defining one or more bytecodes making the sequence more general. Every sequence is substituted with the new macro bytecode, followed by a bit mask that indicates the position of the wildcards and by the wildcards themselves. When the virtual machine interpreter encounters the new bytecode, it fetches the bit mask, from which it knows the number of wildcards (i.e. the number of set bits) and the wildcards that afterwards will be inserted into the pattern. We propose a similar approach that allows wildcard utilization only for the arguments of Java bytecodes. Not all the arguments are substituted with a wildcard. If a bytecode has the same argument through all the sequences collected, this argument will be not collected as a wildcard but as a normal bytecode into the dictionary, as in the plain dictionary compression.

Going back to the example of Figure 5, now we can expand the second macro of the plain dictionary compression as showed in Figure 7. In fact thanks to wildcard we can add also the *getfield* instruction letting blank the place for its argument. Referring to Figure 6 the second macro is applied as:

```
::short absXDist(short x)
P:00  macro2' 0x02 0x00
```

The arguments of *macro2'* are two bytes. The first is a bit mask that declares the position of the wildcard, while the second is the wildcard content (the argument of *getfield_s_this* bytecode instruction).
This new macro allows to save a total of 5 bytes, one less than the same one in the plain compaction, and we can save 11.3% of the utilized space.

3.1.2 Dictionary Compression with Generalized Instructions

We propose a second technique, derived from the plain one, that inserts into the pattern generalized instructions for certain groups of java bytecodes.
The Java Card standard presents some groups of instructions like *SLOAD_0*, *SLOAD_1*, *SLOAD_2* and *SLOAD_3* that have the same functionality but differ only for the target (in these cases a short integer is loaded respectively from the local variable 0, 1, 2, 3 onto the Java operand stack). Sequences that have this kind of instructions will be stored into the dictionary with the generalized instruction in place of the generalizable one; every occurrence of the pattern is substituted with a new op-code followed by the needed bytes for the encode information about the generalized instructions (i.e. for covering the four SLOAD instructions only 2

```
::Point()
P:00  aload_0
P:01  invokespecial 00 07
P:04  aload_0
P:05  sconst_0
P:06  putfield_s 00
P:08  aload_0
P:09  sconst_0
P:0a  putfield_s 01
P:0c  aload_0
P:0d  sconst_0
P:0e  putfield_s 02
P:10  return
::Point( short x, short y, short z)
P:00  aload_0
P:01  invokespecial 00 07
P:04  aload_0
P:05  sload_1
P:06  putfield_s 00
P:08  aload_0
P:09  sload_2
P:0a  putfield_s 01
P:0c  aload_0
P:0d  sload_3
P:0e  putfield_s 02
P:10  return

::short absXDist(short x)
P:00  getfield_s_this 00
P:02  sload_1
P:03  invokestatic 00 08
P:06  sreturn
::short absYDist(short y)
P:00  getfield_s_this 01
P:02  sload_1
P:04  invokestatic 00 08
P:06  sreturn
::short absZDist(short z)
P:00  getfield_s_this 02
P:02  sload_1
P:03  invokestatic 00 08
P:06  sreturn

::short absDiff(short a, short b)
P:00  sload_0
P:01  sload_1
P:02  if_scmple 08
P:04  sload_0
P:05  sload_1
P:06  ssub
P:07  sstore_2
P:08  sload_2
P:09  sreturn
P:0a  sload_1
P:0b  sload_0
P:0c  ssub
P:0d  sstore_2
P:0e  sload_2
P:0f  sreturn
```

Figure 5: Bytecodes of the compiled class file

Macro1:	*Macro2:*	*Macro3:*
aload_0	sload_1	ssub
invokespecial 00 07	invokestatic 00 08	sstore_2
aload_0	sreturn	sload_2
		sreturn

Figure 6: Macros for Plain Dictionary Compression

Macro1:	*Macro2':*	*Macro3:*
aload_0	getfield_s_this xx	ssub
invokespecial 00 07	sload_1	sstore_2
aload_0	invokestatic 00 08	sload_2
	sreturn	sreturn

Figure 7: Macros for Dictionary Compression with Wildcards

bits are needed). Also in this case where all the occurrences of a pattern have the same generalizable instruction, the algorithm puts the original instruction into the pattern of the dictionary, as in the normal PDC.

In our example of Figure 5, starting from the plain compaction results we can expand the third macro as in Figure 8. It is important to notice into the macro the presence of *SLOAD_N*, a new bytecode to indicate a general *SLOAD* that has specified the referring local variable into the arguments of the macro bytecode (in this case the arguments of the two load can be stored in one byte since only 2 bits are needed for each one). With reference to Figure 6 an example of the application of the third macro within the code is:

```
::short absDiff(short a, short b)
P:00  sload_0
P:01  sload_1
P:02  if_scmple 08
P:04  macro3' 0x04
```

The argument of *macro3'* is a byte. The first two bits of the byte are 00 (*SLOAD_0*) and the third and fourth bits are 01 (*SLOAD_1*).
With this expansion we save the same amount of bytes of PDC.

3.2 Static and Dynamic Dictionary

Security is a major requirement for smart cards. Customers are very concerned about every potential security threat, and they typically adhere to the principle that simpler architectures are easier to protect. Hence, the introduction of new components is not always considered a good design practice, even if allowed by the standard [9]. The additional component in the case of dictionary compression is the dictionary itself that is sent along with the application in the form of a custom CAP component, as described in [4, 2]. In this case the dictionary is dynamic since it is

Macro1:	*Macro2:*	*Macro3':*
aload_0	sload_1	*sload_N*
invokespecial 00 07	invokestatic 00 08	*sload_N*
aload_0	sreturn	ssub
		sstore_2
		sload_2
		sreturn

Figure 8: Macros for Dictionary Compression with Generalized Instructions

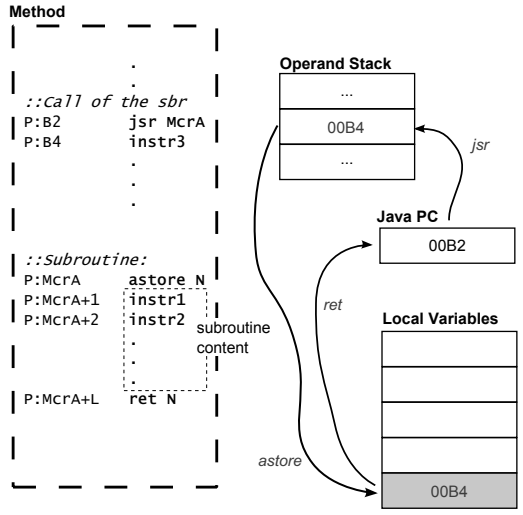

Figure 9: Structure of a Subroutine Macro

built during compression and is strictly related to the application. A solution to this problem is the substitution of the dynamic dictionary with a static one, internally stored into the virtual machine. Static dictionaries generally target a specific context. The building of the dictionary can not be based on a single application but on a set of applications. With high probability, only the applications taken as base in the build process of the dictionary will take advantage of the definitions stored in it. This solution is less effective in terms of compression respect to a dynamic dictionary, since the dictionary has to be shared and therefore made more general.

3.3 Dictionary Compression with Subroutine

The previous techniques all need both a modification of the Java virtual machine and, to be really effective, an additional CAP file component for storing the dictionary. To overcome these two drawbacks we propose a pure Java solution that uses subroutine for compacting the code. In the Java instruction set [9] there are two bytecodes for the purpose of building subroutines.

JSR instruction, acronym for "Jump Subroutine", loads the Java program counter (JPC) of the following instruction into the Java operand stack and adds its argument (an offset) to the actual JPC. In this way the new JPC points the first instruction of the subroutine, whereas the "return" JPC is stored in the stack. The first instruction of the subroutine saves the return JPC on the Java operand stack into an ad hoc local variable (LV). After the execution of the subroutine content, the *RET* ("Return from Subroutine") instruction allows to return to the main execution, loading the address contained in the ad hoc LV into the JPC again. Figure 9 shows the structure of the macro with a view of the LVs where the ad hoc LV is highlighted with a gray fill.

This model needs to add a LV to every stack frame owned by a method interested by compaction. This LV has the purpose of storing the return JPC and is necessary for the sake of a complete isolation of the compression mechanism from the operational flow of the method. The main drawback of this schema is its applicability. In fact JSR and RET instructions can only manipulate the JPC to addresses within the method they belong to and this limits the potential compaction. Another limit is the cost of the structure; every occurrence is substituted by the *JSR* instruction (2 bytes) and the subroutine has an initial *ASTORE* instruction (2 bytes in the worst case) and a final *RET* instruction (2 bytes). Beyond the pure Java nature that does not need any extension of the virtual machine, the main advantages of this technique is that no additional component is needed to store the dictionary. In fact all the definitions used into a method are stored at the end of the method itself. In this way there is a different dictionary for every method.

Making now a short digression about the opportunity of using *JSR* and *RET* instructions, we are aware of the problems related to the verification process as described in [14]. JSR and RET are used for translating the *try-catch-finally* in the Oracle's implementation of compiler for the Java programming language prior to Java SE 6 [7]. The complexity that this process introduces is not easily handled by verifiers. These problems do not affect the proposed technique since we do not combine the JSR/RET instructions with other branch types, and we do not modify the order of creation/access of the local variables.

Application Name	Meth. Comp. Size [bytes]
xPAY	1680
MChip	23305
xPAYAdvanced	1784
MChipAdvanced	38255

Table 1: Sizes in bytes of the analyzed applications

4. RESULTS AND DISCUSSION

For analyzing the developed techniques we used a set of four industrial banking applications. These applications are written paying a lot of attention to performance and ROM size minimization. This means that code is written fully using Java Card language potential. The applications are of different sizes as can be seen in Table 1.

In the following sections we analyze first of all the behavior of the compaction methods with a dynamic dictionary that can be stored into an additional CAP file component, and then with a static dictionary. As mentioned before all the percentages are calculated respect to the method component.

4.1 Plain Dictionary Compression

In Table 2 we report all the compaction ratios reached for the examined applications. In the second column we show the number of new bytecodes needed to represent all the sequences found.

As already said it is not surprising that the number of new bytecodes is higher than the number of the available ones from the standard, since after the tenth, two bytes are used for every pattern. Comparing respectively the number of new bytecodes with the size of every application is interesting to see that the bigger the latter is, the higher is the first. We also see that the ratios we achieved here are comparable to the compression ratios reached in [4]. The

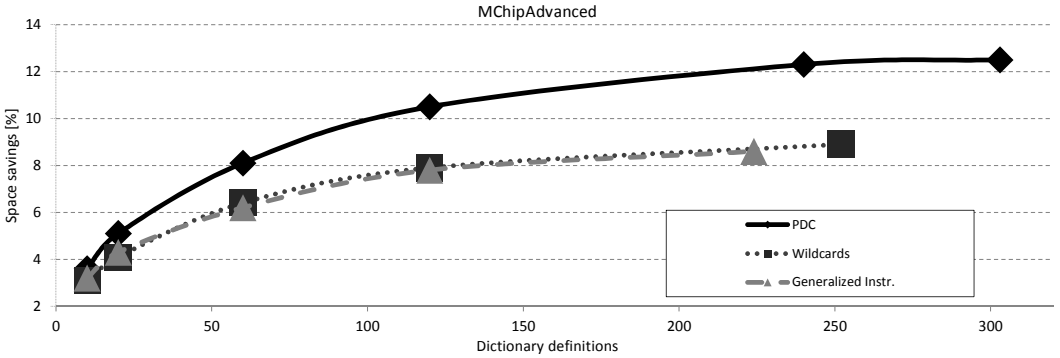

Figure 10: Trend of the MChipAdvanced application with dynamic dictionary

Appl.	Nr. of Bytecodes	Compr. Ratio [%]
xPAY	28	88.2
MChip	229	89.2
xPAYAdv	28	87.2
MChipAdv	303	87.5

Table 2: Compression ratios for PDC with dynamic dictionary

Appl.	Nr. of Bytecodes	Compr. Ratio [%]
xPAY	19	95.2
MChip	158	90.4
xPAYAdv	20	94.6
MChipAdv	252	91.1

Table 3: Compression using wildcards with dynamic dictionary

little gap in compression values is probably due to the high optimization of the applications tested.

4.2 Dictionary Compression with Wildcards and with Generalized Instructions

Tables 3 and 4 show the results for the dictionary compression with wildcards and with generalized instructions, respectively. Generally for every case we see that the compression ratios reached are higher than those from PDC.

The reason for this behavior can be individuated in the higher cost for the substitutions bound to the low number of similar sequences that can be grouped with these criteria. On the other side we see that compression rates saturate for lower numbers of new bytecodes. We also propose for the biggest application (MChipAdvanced) a graph in Figure 10 where we show the trend of space savings for the three techniques in relation to the number of new bytecodes introduced. Beyond recognizing that after a threshold the space savings does not improve any further, we see that also taking a finite number of new instructions the PDC performs in any case better than the other techniques.

Appl.	Nr. of Bytecodes	Compr. Ratio [%]
xPAY	8	92.4
MChip	160	91.4
xPAYAdv	17	90.5
MChipAdv	224	91.4

Table 4: Compression using generalized instructions and a dynamic dictionary

4.3 Compression with Static Dictionary

All previous results are achieved by means of dynamic dictionaries meaning that each application is associated to a dedicated dictionary. In this section we analyze the behavior of compression with static dictionary. The static dictionary as expected is bigger than the single dynamic ones. As an example in the case of PDC, maximum dictionary dimension, rated as number of definitions inside, contains up to 462 entries compared against the 290 entries of the dynamic dictionary for MChipAdvanced. Having a complete dictionary in this case makes sense only if all the applications are installed together on the same smart card and hence its cost can be shared among multiple programs. This is not the case of these applications since, for application-specific issues, only the couple *MChip* (or *MChipAdv*) - *xPAY* (or *xPAYAdv*) is installed. Figure 11 shows two important aspects of the static compression taking in exam the MChipAdvanced application. In the figure the horizontal axis indicates the definitions in the static dictionary. In the vertical axis on the left we have the space savings percentage for the same three compression techniques above examined, while on the right vertical axis we have the number of dictionary definitions not used in MChipAdvanced but for the other applications. Looking to this latter is interesting to notice that, about after 90 definitions, the number of unused definitions starts to grow faster. For the same horizontal axis point also the space savings percentage starts to grow slower. This crossed information tells that it is not worth keeping more than 90 definitions (new bytecodes) in the static dictionary which allow to save about the 7% of the size. In general, also the other applications have the same behavior with a reduction of the original size into the range between 5 and 7%. The compression performance is lower than in case of using dynamic dictionaries but this can be the best solution in case of design constraints where additional CAP components are not allowed. Looking again at Figure 11, it is interesting to notice that all space savings rates of the two techniques pro-

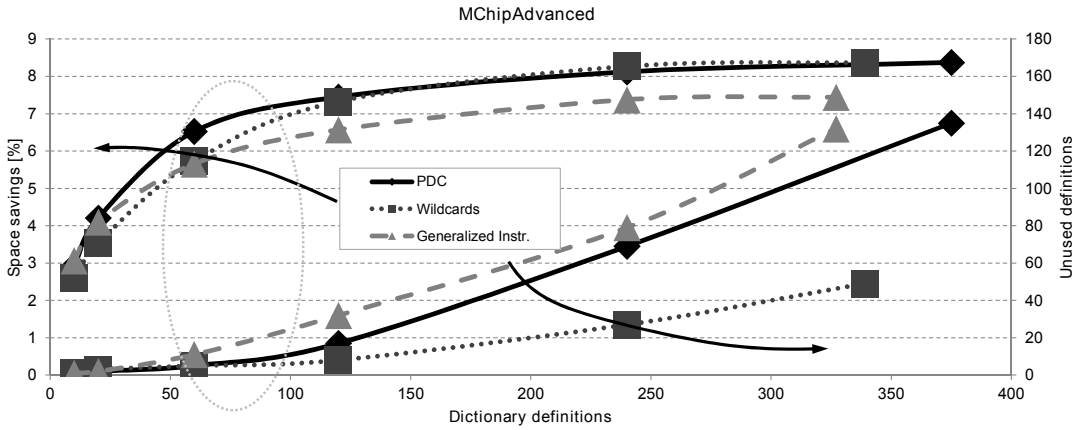

Figure 11: Trend of the MChipAdvanced application with static dictionary

Appl.	Compr. Ratio [%]	Additional LV
xPAY	97.6	2
MChip	97.5	28
xPAYAdv	96.0	3
MChipAdv	96.1	35

Table 5: Compression ratios and additional local variables using subroutine compression

posed in this paper are very close to the PDC one. This very low decrease of compression efficiency makes the proposed techniques a good choice for a static dictionary since more generalized definitions have a higher probability to be suited for other applications not used for the dictionary building process.

4.4 Compression with Subroutine

Compression with subroutines does not need additional components for the dictionary so there is no need to limit the number of definitions but, as Table 5 shows, ratios reached are higher than in the previous cases.

The same table also shows the number of additional local variables (local variables are words of two bytes). This information does not concern the additional RAM needed. The latter depends on the calling tree, hence the information on the table represents a worst case (all the methods with additional local variables are called one by the other, and there are not recursive loops) of additional RAM consumption. The higher compression ratio in comparison to the other techniques analyzed in this paper could make the other techniques preferable, but also other aspects have to be taken in consideration. First in order is its applicability to any application, that can be uploaded to any existing Java Card. Another strong point is its easy insertion into the software developing flow as a post compiling process that does not need additional effort by developers.

5. CONCLUSIONS

In this paper we proposed two novel dictionary-based techniques for compressing Java methods by means of the introduction of new bytecodes. When compared against a non-parametrized solution, we show that, on average, the latter performs better. A flexible combination of the three techniques presented here seems promising for achieving lower compression ratios, and we are evaluating the development of such a solution as future work.

Moreover, we investigated the applicability of compression techniques in scenarios where software architects' degree of freedom is low. In this context, a static dictionary stored inside the virtual machine is a first solution to avoid using additional custom components in the CAP file. The two dictionary-based techniques proposed in this work are a possible choice for a static dictionary. The higher generalization of the definitions makes them suitable with higher probability for applications not taken in consideration in dictionary building phase. This paper presented a first study of such approach, and we want to extend this work to develop a full and functional algorithm that could balance the weight of the applications taken as population respect to specific factors (i.e. percentage of devices where the application will be uploaded), in the phase of dictionary creation.

Finally, we analyzed the opportunity of compressing bytecode by means of Java subroutines. Since such solution complies with every existing Java Card, it can be applied to every application, regardless of where the application will be installed. This elegant solution can achieve a space savings between 2.5 and 4.0%. More importantly, this technique could be included as an automatic post-compiling process, greatly simplifying its deployment.

6. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, volume 1009. Pearson/Addison Wesley, 2007.

[2] G. Bizzotto and G. Grimaud. Practical java card bytecode compression. In *Proceedings of RENPAR14/ASF/SYMPA*. Citeseer, 2002.

[3] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Prentice Hall, 2000.

[4] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java Bytecode Compression for low-end Embedded Systems. *ACM Trans. Program. Lang. Syst.*, 22(3):471–489, May 2000.

[5] M. T. C. S. JIS. Computers and Intractability. A Guide to the Theory of NP Completeness. 1979.

[6] R. Johnson, D. Pearson, and K. Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. *SIGPLAN Not.*, 29(6):171–185, June 1994.

[7] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification. Java SE 7 Edition*. Oracle, September 2012.

[8] Oracle. *Java Card 3 Paltform. Runtime Environment Specification, Classic Edition. Version 3.0.4*. September 2011.

[9] Oracle. *Java Card 3 Paltform. Virtual Machine Specificatio, Classic Edition. Version 3.0.4*. September 2011.

[10] T. A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '95, pages 322–332, New York, NY, USA, 1995. ACM.

[11] W. Rankl and W. Effing. *Smart Card Handbook*. Wiley, 2010.

[12] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag New York Incorporated, 2004.

[13] D. Saougkos, G. Manis, K. Blekas, and A. Zarras. Revisiting Java Bytecode Compression for Embedded and Mobile Computing Environments. *Software Engineering, IEEE Transactions on*, 33(7):478 –495, july 2007.

[14] R. F. Stärk and J. Schmid. Java Bytecode Verification is not possible. *Formal Methods and Tools for Computer Science, Eurocast*, 2001.

[15] A. Wolfe and A. Chanin. Executing compressed programs on an embedded risc architecture. *SIGMICRO Newsl.*, 23(1-2):81–91, Dec. 1992.

2014 17th Euromicro Conference on Digital System Design

Instruction Folding Compression for Java Card Runtime Environment

Massimiliano Zilli, Wolfgang Raschke,
Reinhold Weiss, and Christian Steger
Graz University of Technology
Institute of Technical Informatics
Graz, Austria
Email: {massimiliano.zilli,wolfgang.raschke, rweiss,steger}@tugraz.at

Johannes Loinig
NXP Semiconductors Austria GmbH
Gratkorn, Austria
Email: johannes.loinig@nxp.com

***Abstract*—Java Card is a secure Java running environment targeted for smart cards. In such low-end embedded systems, ROM size and execution time play very important, usually opposing roles.**

Dictionary compression can be applied to the Java Card software architecture, but pays for the reduced ROM size of the applications with a higher execution time. On the other hand, acceleration mechanisms to speed up the execution need additional information or additional software complexity, with the effect of increasing ROM size.

In this paper, we propose a dictionary compression system based on an instruction folding mechanism that permits a reduction in the ROM size of Java Card applications, and at the same time, a speed-up of their execution.

***Keywords*-Smart card; Java Card; compression; instructions folding;**

I. Introduction

Smart cards are nowadays a very widespread technology. The main fields of application are banking, e-government, and identification. Because of this large diffusion, these systems have to be cheap, and therefore can only rely upon very limited resources. Typical hardware configurations are based on 8/16 bit processors with some kilobytes of RAM and up to a few hundred kilobytes of non-volatile memory, distributed between ROM, Flash and EEPROM memory. In such a category of embedded systems, the programming language for applications is often C or even assembly [1]. The usage of an object oriented, interpreted language (e.g. Java) would help improve portability, but with the side effect of a slower execution. Unfortunately, the adoption of the complete Java standard in smart cards is unrealistic because of the very limited resources available in these systems. This is the main motivation that led to the development of a Java standard expressly for smart cards called Java Card [2] [3]. The additional advantage of this environment is a high degree of security due to the Java architecture, and to specific functionalities such as cryptography.

In Java the applications are distributed in class files, while in Java Card they are distributed in CAP files. The structure of a CAP file is similar to that of a *class* file but it is a compressed form similar to a JAR file. Like the class file, the CAP file has several components. The Java Card environment relies on these components when it has to install an application [4]. We focus our attention on the *Method Component* that contains all the classes' methods compiled in the form of bytecode sequences. At run-time the Java virtual machine interprets the bytecodes, making the real hardware execute machine instructions. The method component and the Java virtual machine have been the object of several research contributions, focusing on code compression or instruction execution acceleration; however, none of them achieves both of these outcomes.

In this paper we propose a method to decrease the code size and to speed up its execution. With the introduction of a small number of new instructions in the Java Card instruction set, it is possible to substitute sequences of bytecodes into the method component, shrinking down the code size. Since the new instructions are the optimized version of the instructions contained within the substituted equivalent sequences, the execution of the new instructions is faster. The result of this combination is a reduction in the application code size, and at the same time a speed-up of its execution.

The structure of the rest of the paper is as follows. Section II reviews the published work on code compression and execution acceleration that forms the basis of this research. Section III explains our concept of folding compression. In Section IV, we provide a description of the implementation of the technique. Section V presents the results of the compression, focusing on ROM size and run-time. Finally, in Section VI we report our conclusions and a view on future work.

II. Related Work

Compression of applications is a very delicate topic and several compression mechanisms concerning executable code have been proposed. On the one hand, compression allows for a reduction in ROM size; on the other hand, there is the problem of the decompression phase. The latter can be reduced to two main aspects: the time necessary to decompress the program and the RAM resources needed for

978-1-4799-5793-4/14 $31.00 © 2014 IEEE
DOI 10.1109/DSD.2014.32

storing it. Classic compression methods such as Huffman coding need to decompress the entire compressed information. Even if the compression is profile-guided, as Debray et al. show in [5], the decompression phase needs an additional amount of RAM; this memory consumption hardly fits with smart cards, where the hardware is usually strictly tailored to software requirements, and where there are not many free resources left.

In the context of smart cards, dictionary compression is a more suitable solution due to its simpler decompression phase [6]. The compression phase consists of substituting repeated sequences of information units with macros, whose definitions are stored in a dictionary. In the decompression phase, the system looks into the dictionary for the macros encountered through the compressed stream and makes a substitution. In the case of Java Card, compression involves the bytecodes in the method component and can be done at compile time off-card [7]. Thanks to the software architecture of the virtual machine, the bytecode interpreter is easily customizable for handling the dictionary. Clausen et al. show that macros used in compression can be introduced as new instructions in the interpreter instruction set. At run-time, when the interpreter encounters a macro it looks up the macro definition in the dictionary, and can then execute directly the equivalent sequence of bytecodes. Therefore, there is no need to separate the entire decompression phase from the execution phase. The definitions of the macros are stored in a dictionary that is application specific and sent with the application CAP file [7]. Space savings of about 15% are obtained, with a run-time speed penalty of between 2% and 20%. In [8], we propose an architecture differentiation between a static and a dynamic dictionary. We also propose modifications to the classical dictionary compression to make the macros more general with the addition of arguments.

Software architectures based on interpreters such as Java pay for the *"write once, run anywhere"* property with a slower execution compared to applications compiled in native machine language. Hence, execution time has always been a critical factor for any system deploying Java. The main approach present in most widely spread Java environments is the *"Just-In-Time"* (*JIT*) compilation [9] [10] [11]. It consists of compiling the bytecode sequences during run-time into machine instructions performing optimizations that make execution faster than a simple interpretation on the Java virtual machine. This run-time compiler has to be fast enough to bring benefits to the total execution time and needs RAM space for temporary storage of the compiled code. In an embedded system such as a smart card, RAM is a very scarce resource and the processor does not have high speed; hence, a mechanism like JIT compilation is not feasible.

Superoperator theory, introduced by Proebsting in [12], is the idea behind one of the optimizations present in JIT compilers. According to superoperator theory, a sequence of bytecode instructions can be optimized, keeping intermediate result values in machine registers instead of pushing them into the operand stack. This reduction of memory accesses allows a speed-up in the entire execution.

For systems where JIT compilation is not possible, the introduction of *superinstructions* provides the possibility of improving the performance of the interpreter [13] [14] [15] [16] [17]. Superinstructions are superoperators that are added to the instruction set of the Java virtual machine as new instructions. In contrast to the equivalent sequence of instructions that they substitute for, superinstructions have the advantage of a single fetch and the possibility of eliminating redundant operations (i.e. operand stack accesses). In addition, superinstructions represent a form of dictionary compression, owing to the fact that a sequence of bytecodes is substituted with only one bytecode. In [16], Casey et al. discuss the policy for choosing the superinstructions from among the available sequences.

To enhance execution performance, a Java processor is proposed [18]. Java processors are CPUs designed to execute Java bytecode instructions directly in hardware. Beyond the direct bytecode execution, a Java processor such as *picoJava* has a hardware optimization mechanism called the *Instruction Folding Unit* [19]. This hardware feature analyzes a finite number of sequential instructions in the instruction buffer and decides if it is possible to fold them into a single register machine like instruction. As with superinstructions, this feature avoids redundant read/write operations onto the operand stack.

In [20], Badea et al. introduce an *Annotation Aware Virtual Machine* able to recognize foldable sequences. The virtual machine includes the superoperators in the instruction set. When it encounters an annotation, instead of executing the sequence of bytecodes, it executes the corresponding superoperator. The compilation phase takes charge of the bytecode analysis, the search of superoperators, and of the creation of the annotation. Hence, the virtual machine is relieved of this computational effort at the run-time. Azevedo et al. propose a similar concept for smart cards running the Java Card environment [21]. The annotation-aware JCVM allows for a run-time speed-up between 10% and 130% at the price of a class-file size increase between 6.6% and 14%.

III. Folding Compression

Folding compression is the combination of dictionary compression and folding mechanism. The latter is an optimization applicable on specific sequences of bytecodes. The key factor for the identification of these sequences is the *Java Operand Stack* utilization. In fact, a Java bytecode can be categorized respect to the number of operands that it pops from and pushes into the operand stack. We use the same classification introduced for the Java Processor in [19]. For example a SLOAD instruction, which pushes a word into the operand stack, is a *Producer*; a SSTORE, which pops a

```
P:00  sload_1
P:01  sload_2
P:02  ssub
P:03  sstore_3
```

Figure 1. Example of a foldable sequence

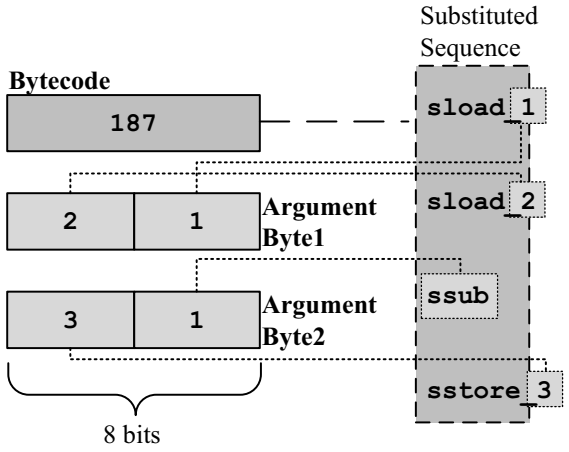

Figure 2. Example of a new instruction with argument specification

word, is a *Consumer*. A SSUB, which pops two words and pushes one, with a balance of a word popped, is an *Operator*. In Figure 1, we report a sequence of instructions where there is heavy use of the stack with three writes and three reads. If we consider the sequence as a block, we can reduce it into a register machine like instruction that operates directly on the local variables. This register machine instruction bypasses the stack, and hence, it avoids the three reads and the three writes with the consequent reduction in execution time.

As often happens, ideas come from observation. Searching for every possible folding sequence, we realized that most of the sequences do not use any other local variables than the first sixteen. Theoretically, Java Card instructions (e.g. SSTORE) can address up to 256 local variables, but a method is unlikely to have many more than sixteen local variables shared between automatic variables and parameters. From this observation arose the idea to abstract a register machine over the actual Java method frame. The abstract register machine can map up to the sixteenth local variable of the actual Java method frame as source and destination for its instructions. If a sequence does not fit in this model, it is not taken into consideration for the folding compression; the JCVM will execute the instructions contained in this sequence as standard bytecode instructions.

For the sake of a more clear explanation, in Figure 2 we report the structure of the instruction that will substitute for the sequence in Figure 1. The first byte is the new bytecode that a modified JCVM can decode as a sequence of SLOAD SLOAD [*Operator*] SSTORE instructions. The second byte is subdivided into two digits:

- the lowest digit contains the local variable number of the first SLOAD instruction
- the highest digit contains the local variable number of the second SLOAD instruction

The third byte is also subdivided into two digits:

- the lowest digit specifies the operation to execute (SADD, SSUB, SAND, SSHL, ...)
- the highest digit contains the local variable number of the last SSTORE instruction

The substitution of a sequence of instructions with a new one is definitely a dictionary compression, in which the dictionary consists on an extension of the instruction set of the JCVM. The presence of an argument makes the substituting instruction more general, and therefore suitable for similar foldable sequences (e. g. with the same new instruction of Figure 2 we can represent as well a sequence composed by SLOAD_3 - SLOAD_4 - SXOR - SSTORE_1, but with a different argument for a proper specification). In the example, whereas the starting sequence occupies four bytes in ROM memory, the new instruction occupies only three bytes; the final balance is one byte saved.

IV. Design and implementation

In this section we present the implementation of the entire system. We modified the Java Card virtual machine, extending its *Instructions Set*, and we added a post-compiling process to adapt the CAP files. The approach of simultaneously designing both of these components is analogous to a hardware-software co-design.

A. CAP File Transformation

Figure 3 shows the process for creating and installing an applet in Java Card. Unlike the standard Java virtual machine, JCVM is split into two parts: one that runs off-card and the other that works on-card. The converter is part of the off-card JCVM and creates the CAP file pre-linking the .class file. Pre-linking reduces the size of the application and allows for a faster installation process. The CAP file represents the means through which the application is distributed. Between the creation time and the installation time, the CAP file is exposed to an unsafe environment, where the corruption of its content is possible. Hence, before the installation, the verifier checks the content of the CAP file and assures that the application can be securely installed into the Java card and executed. During the installation process, the installer loads the verified CAP file into the permanent memory of the smart card. The installer consists on two parts, one off-card and the other on-card; the two parts exchange the CAP file content through an application protocol data unit (APDU). Once the application is installed, the interpreter starts its execution.

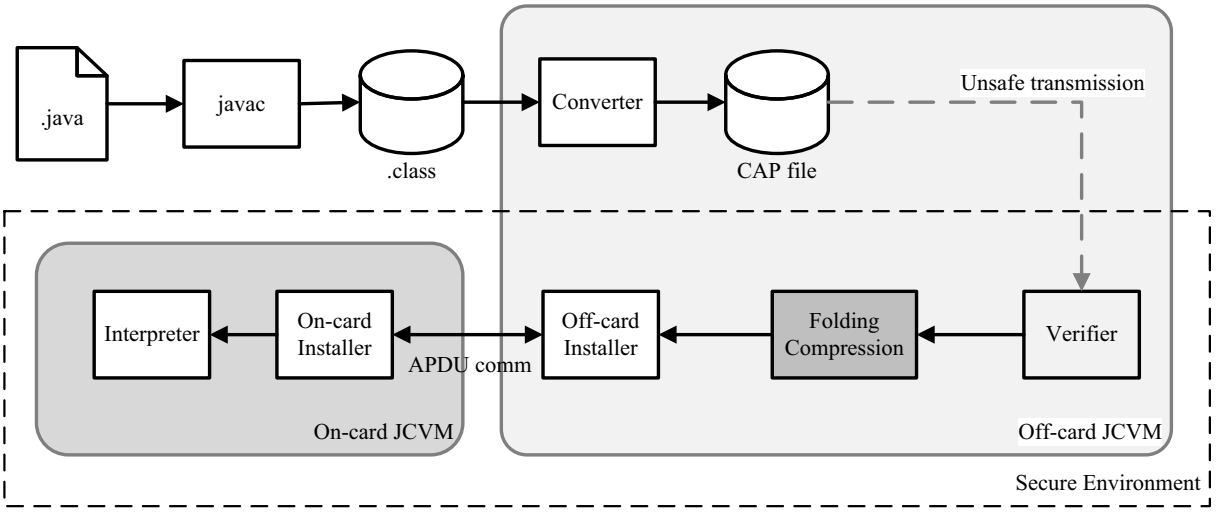

Figure 3. Java Card Virtual Machine architecture

In the JCVM architecture, we perform our folding compression after the applet verification process. Therefore, the distributed CAP file can be generic for every virtual machine. Only virtual machines enabled for executing the new folding instructions will transform the CAP file before the installation. The folding compression process could also be located after the converter but two issues would arise:

- the CAP file has to be specific for Java Card enabled for executing the new folding instructions
- the verifier has to be adapted to support the new folding instructions

Regarding the backward compatibility, the on-card JCVM enabled for the folding compression remains compliant with off-card JCVMs that are not enabled for the folding compression. In fact, due to the fact that the folded instructions constitute just an extension of the standard JCVM, the on-card JCVM enabled for folding compression is still able to run standard applications.

B. Code Parsing

In Java Card, the Java converter creates the CAP file that contains the executable bytecodes in the method component. The compression phase consists on the substitution, in the method component, of existing foldable sequences of bytecodes with new bytecodes.

In [8], the authors use *Single Entry, Single Exit* blocks for dictionary compression. The nature of the folding compression, as described in this paper, is slightly different. A foldable block cannot have internal branches but may finish with a branch. Once the flow of control starts at the beginning of the block, it leaves the block at the end without prior halt or branching. The block cannot have internal labels (i.e. destination of branches) because if the block were substituted with a single bytecode, an internal instruction would not be reachable anymore. These characteristics lead to the classical definition of a *Basic Block*, commonly used in compiler theory [22].

Table I
POSSIBLE FOLDABLE COMBINATIONS OF PRODUCER(P), CONSUMER(C), AND OPERATOR(O) BYTECODES

Nr. of instr.	Poss. combinations
2	P-C, P-O, O-C
3	P-P-O, P-O-C
4	P-P-O-C

Because the CAP file analysis is a post-compilation activity performed off-card, the computing resource is not critical. The first step is the interpretation of the bytes forming the method component, transforming them into a sequence of Java bytecodes with the respective arguments. For the sake of the foldable sequences search, every bytecode instruction is marked with an additional attribute that describes its behavior with respect to the Java operator stack. This attribute can identify the Java bytecode as a *producer*, a *consumer*, an *operator* or a neutral instruction. A neutral instruction does not access to the Java operand stack; hence, it will not be part of a foldable sequence. The instruction characterization is fundamental because not every sequence is possible. Table I shows a summary of the possible foldable combinations according to [18]. Foldable sequences have a minimum of two (i.e. SLOAD SSTORE sequence) and a maximum of four bytecode instructions (i.e. SSLOAD SSLOAD SADD SSTORE sequence).

```
#define MAX_NR_INSTR 4

FoldableSequence sequences[];

for(i=0;i<instrs.size();i++) {
 Instruction seq[];

 if(isFoldableInstr(instrs[i])) {
  for(j=1;j<MAX_NR_INSTR;j++) {
   if(isFoldableInstr(instrs[i+j]) {
    seq.add(instrs[i+j]);
   }
   else {
    break;
   }
  }
 }
 if(isFoldableSeq(seq)) {
  sequences.add(seq);
  i=i+j;
 }
}

for(i=0;i<sequences.size();i++) {
 substituteFoldSeq(sequences[i]);
}
```

Figure 4. Pseudo-code for the code compression

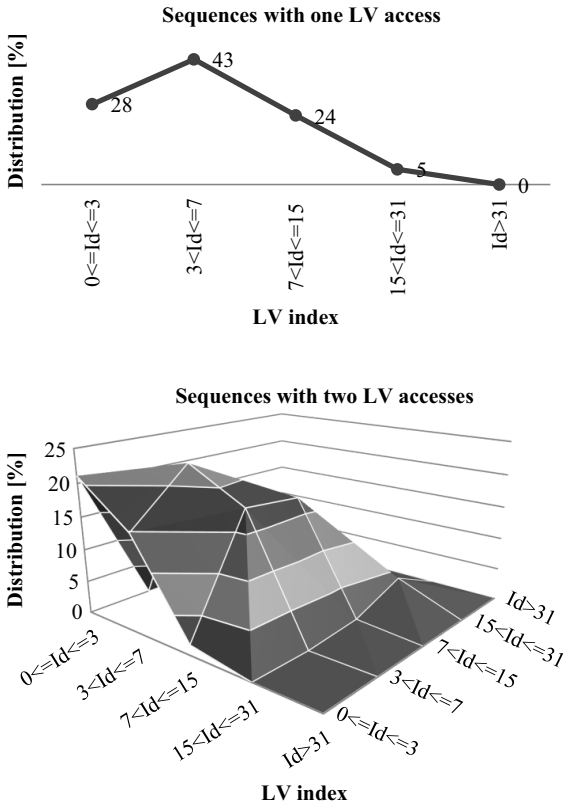

Figure 5. Distribution of local variable index

The bytecodes in the method component are not sorted; therefore, we used a linear search algorithm to find the foldable blocks. The algorithm is very simple and consists basically of finding the largest foldable basic blocks. In Figure 4 we report the pseudo-code for the search and substitute algorithm. This approach leads to a maximizing of both the space savings and the run-time performance. After the search, the process converts all the foldable sequences into new bytecode instructions that we present in Subsection IV-E.

C. Local Variables Coverage

As anticipated, the reduced set of new instructions does not cover all the possible foldable sequences. In Figure 5 we show the distribution of the local variables used in the sequences. This data is the result of the analysis of three industrial applications. The figure takes into consideration the foldable sequences that access one and two local variables (for sequences with three local variable accesses, the result is analogous). We discretize the local variable indexes in segments depending on the bits needed to express the indexes. We draw also line and surface between values with the purpose of showing the distribution trend, but the interpolation between values is meaningless. It can be seen that accesses to local variables with index higher than fifteen are very unlikely. The coverage of foldable sequences reachable considering only the first sixteen variables is about 95%. Based on this observation, we decided to use only four bits (instead of a byte, as in SLOAD and SSTORE Java Card instructions) to specify the local variable index in the argument of our new instructions.

D. Constants Coverage

A foldable sequence may contain also bytecode for constants usage (i.e. SSPUSH, CONST_0, ...). The strategy for encoding constants in the new bytecode arguments is similar to that used in Java bytecodes. In Java, there are seven immediate push constant instructions that write into the operand stack constants from -1 to 5. In addition, there are two push operators that write into the stack a short integer value stored in the code, in the form of a byte and of a short respectively.

Our approach is the same for the latter two instructions, but we use a digit in the bytecode option to specify constants in the range from -1 to 11. The last two combinations of the digit specify the presence of an additional argument for the constant expression. The optional argument may be one or

Table II
INSTRUCTION SET EXTENSION

Instruction	Argument	Opt. Arg.	Comb.
LdSt	B1[St:Ld]	-	PC
PshSt	B1[Op:Cnst]	B2[BPsh]B3[SPsh]	PC
OpSt	B1[St:Op]	-	OC
LdIf_s2b	B1[Op:Ld]	B2[Brf]B3[Brw]	PO
LdPshAdd	B1[Cnst:Ld]	B2[BPsh]B3[SPsh]	PPO
LdPshOp	B1[Cnst:Ld]B2[Op:Ord]	B3[BPsh]B4[SPsh]B5[Br]B6[Brw]	PPO
LdLdOp	B1[Ld2:Ld1]B2[Op]	B3[Brf]B4[Brw]	PPO
LdPshOpSt	B1[Cnst:Ld]B2[St:Op]	B3[BPsh]B4[SPsh]	PPOC
PshLdOpSt	B1[Ld:Cnst]B2[St:Op]	B3[BPsh]B4[SPsh]	PPOC
LdLdOpSt	B1[Ld2:Ld1] B2[St:Op]	-	PPOC

two bytes long and allows coverage of all the push variants.

E. Instruction Set Extension

Table II lists the ten instructions that we introduced into the virtual machine instruction set. The first column contains the intuitive name of the instruction that helps in understanding the kind of sequence involved. The second column has the fixed argument, and the third reports the optional argument that always contains constants that are not possible to express as a digit, as discussed in the previous subsection. In the fourth column we report the foldable combinations that the new instructions substitute for. We point out that the combination *P-O-C* is not represented with a new instruction because of its low incidence in the set of applications taken in consideration. Java Card specification has 68 undefined bytecodes available; using ten of them allows the remainder to be available for other eventual purposes.

The new instructions except *OpSt* do not access to the stack. *OpSt* pops one or two (depending on the operation being performed) instructions from the stack, executes the specified operation and stores the result in the specified local variable. The remaining new instructions have a behavior similar to the example shown in Section III. In addition, we can consider *LdPshAdd* instruction as optional, because it is a subset of *LdPshOp* instruction. The reason for the introduction of *LdPshAdd* instruction is the high number of SPUSH-SLOAD-SADD and SLOAD-SPUSH-SADD sequences in the applications. As a result of the hard-coded SADD operand, *LdPshAdd* instruction allows for the saving of one byte for the argument and for a faster execution compared to the *LdPshOp* instruction.

V. RESULTS

In this section we present the methodology and the results of the ROM size and run-time analyses for the method proposed in this work.

A. Methodology

For the evaluation of the *folding compression* we took as starting point the Oracle Java Card reference implementation, which constitutes the simulator for the Java Card Development Kit. For our purpose we isolated the code of the on-card software component and ported it into an 8051-based microcontroller. We used a commercial IDE with the capability to compile the system and run it on an instruction set simulator where execution time measurement is possible. The level of accuracy of the simulation is high enough for the time performance evaluation of the additional bytecodes in the virtual machine.

Table III
METHOD COMPONENT SIZE AND PERCENT SPACE SAVINGS

Application	Size[B]	Space Savings [%]
MChip	23305	4.2
MChip Advanced	38255	3.9
XPay	1784	5.6

For the evaluation of both the ROM size and the run-time performances we implemented the new instructions in two variants. The first variant has a normal C style; that means, it uses variables for intermediate results, internal of the instruction function implementations. In the second variant, we minimized manually the usage of variables to reduce the passage of values among variables. We refer to this implementation from now on as *Performance Folding Implementation (PFI)*. In both cases, the compilation was performed with the minimal C compiler optimization set (Constant Folding, Simple Access Optimizing and Jump Optimizing).

B. ROM Size Reduction

We evaluated the ROM size reduction, taking as examples three industrial applications. In Table III, we report the size of the method components of the applications and the space savings; the latter is defined as

$$space\ savings = \frac{S_{orig} - S_{compr}}{S_{orig}}$$

where S_{orig} is the size of the original method component and S_{compr} is the size of the method component after the compression process. The average of the methods space savings weighted by their relative method sizes is 4.0%.

For the completeness of the analysis, we also evaluated the impact of the introduction of the new instructions on the virtual machine ROM size. Table IV shows the increase in ROM size for the two implementations. It is evident that the second variant pays for the performance optimization with a higher ROM occupation of 2.0%. The space saved for an application is smaller than the additional ROM space needed for the integration of the new bytecodes.

The idea behind Java Card is to host several applications installed in a single card [4]; hence, the space saved by compressing all applications is likely to exceed the additional space needed for the Java Run-time Environment

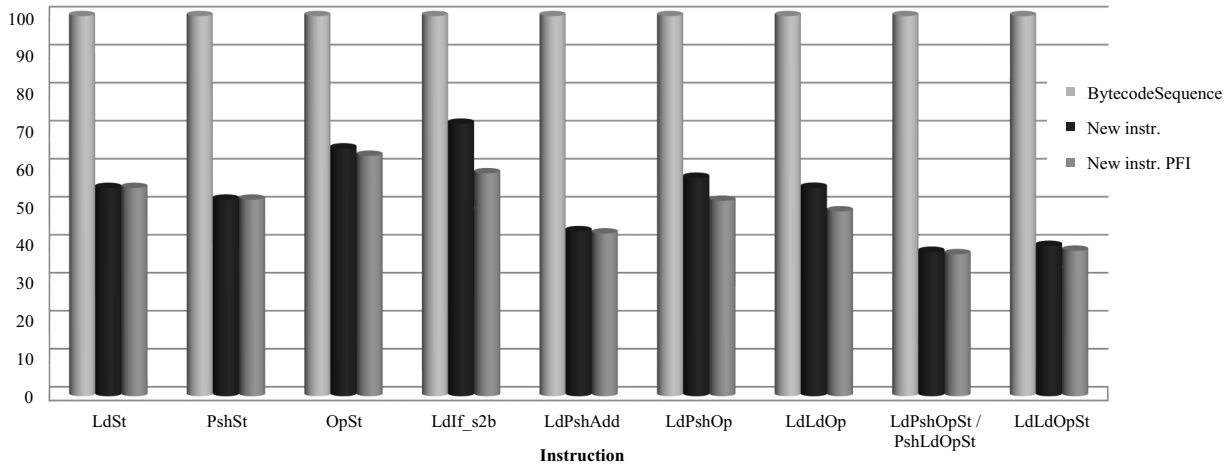

Figure 6. Execution time of the new instructions

Table IV
JAVA CARD RUN-TIME ENVIRONMENT ROM SIZES

JCRE Type	ROM Size [B]	Size incr. [%]
Standard	48950	-
W. Folding Instr.	53863	10.0
W. Folding Instr. PFI	54914	12.2

modification. If we consider the average applet size of 20 kB and the space savings owing to folding compression of 4%, we can estimate that the installation of six applets will balance the ROM size increase of the virtual machine.

C. Execution Speed-up

To evaluate the execution time performance we focused on the improvements regarding the single new instructions. For each new instruction we measured the time needed for its execution in the two variants of Java Card environment and we compared it with the time needed for the execution of the equivalent sequence of standard Java Card instructions. For example, in the case of the foldable sequence in figure 1, we compared the time needed for the execution of the four bytecodes (the two *SLOADs* the *SSUB* and the SSTORE) with the time needed for the execution of the new instruction *LdLdOpSt* in the two versions (normal implementation and PFI).

Figure 6 shows the results of this analysis; for each instruction there are three bars. The first bar represents the execution time of the normal Java Card sequence of bytecodes. The second bar is the time needed to execute the respective folded instruction implemented in normal C style, while the third relates to the folded instruction of the PFI. All execution times are shown as a percentage of that for the normal Java Card (hence, the normal Java Card is shown as 100% for each sequence). The graph shows a reduction in the execution time of 50% for the normal implementation and of 53% for the PFI. The reduction in the execution time of the new instructions is due to Java operand stack bypassing. Comparing these results to the ROM size results in the previous subsection, we point out that for the PFI, with an increase in the virtual machine ROM size of 2.0%, we have a reduction in execution time of 6% compared to the virtual machine with the normal implementation of the new instructions. Therefore, the choice between the two implementations depends on which of the aspects needs to be fostered between speed and ROM size.

To complete the assessment of the execution time, we evaluated a simple Java Card test-bench application. The assessment on the industrial applications was not possible, because of the use of proprietary libraries not available on the Java Card reference implementation. The test-bench application performs operations on big-integers using a custom big-integer class part of the test-bench. Applying the folding compression, we achieved space savings of 5.5% that is consistent with the results obtained with the industrial applications. The decrease in execution time of the test application amounts to 3.5%.

VI. CONCLUSIONS

In this work we have proposed a novel dictionary-based system for compressing Java Card applications that needs a small extension of the Java Card instruction set. The proposed framework consists of two main parts. The first part is the CAP file analysis and manipulation where we achieve space savings for the method component of 4.0%. The second part is the Java Card virtual machine modification where we implemented the functions needed for the interpretation of the new bytecode instructions. With the new

instructions we have a decrease in execution time of 50% compared to the interpretation of the equivalent sequence of standard bytecodes; the decrease of the time needed for the overall application execution is of 3.5%.

The drawback of the method is the increase in ROM memory occupation of the Java Card Runtime Environment. In a context like Java Card, where multiple applications are installed in a single card, the space saved with the compression of the installed applications can balance and exceed the increase in the Java Card Runtime Environment ROM size.

The combination of the presented technique with other compression techniques seems to be promising and feasible, and we are evaluating its investigation as part of future work.

Acknowledgment

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank their project partner NXP Semiconductors Austria GmbH.

References

[1] W. Rankl and W. Effing, *Smart Card Handbook*. Wiley, 2010.

[2] Oracle, *Java Card 3 Paltform. Virtual Machine Specification, Classic Edition. Version 3.0.4*. Oracle, September 2011. [Online]. Available: www.oracle.com

[3] Oracle, *Java Card 3 Paltform. Runtime Environment Specification, Classic Edition. Version 3.0.4*. Oracle, September 2011. [Online]. Available: www.oracle.com

[4] Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Professional, 2000.

[5] S. Debray and W. Evans, "Profile-guided Code Compression," *SIGPLAN Not.*, vol. 37, no. 5, pp. 95–105, May 2002.

[6] D. Salomon, *Data Compression: The Complete Reference*. Springer-Verlag New York Incorporated, 2004.

[7] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller, "Java Bytecode Compression for low-end Embedded Systems," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 3, pp. 471–489, May 2000.

[8] M. Zilli, W. Raschke, J. Loinig, R. Weiss, and C. Steger, "On the Dictionary Compression for Java Card Environment," in *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2013, pp. 68–76.

[9] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling Java Just in Time," *Micro, IEEE*, vol. 17, no. 3, pp. 36–43, 1997.

[10] A. Krall and R. Grafl, "CACAO - A 64-bit JavaVM Just-in-Time Compiler," *Concurrency Practice and Experience*, vol. 9, no. 11, pp. 1017–1030, 1997.

[11] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler," *IBM Systems Journal*, vol. 39, no. 1, pp. 175–193, 2000.

[12] T. A. Proebsting, "Optimizing an ANSI C Interpreter with Superoperators," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 322–332.

[13] M. A. Ertl, C. Thalinger, and A. Krall, "Superinstructions and Replication in the Cacao JVM interpreter," *Journal of .NET Technologies*, vol. 4, no. 1, pp. 31–38, 2006.

[14] D. Gregg, M. A. Ertl, and A. Krall, "A Fast Java Interpreter," in *Proceedings of the Workshop on Java optimisaton strategies for embedded systems (JOSES), Genoa*. Citeseer, 2001.

[15] B. Stephenson and W. Holst, "Multicodes: Optimizing Virtual Machines using Bytecode Sequences," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 328–329.

[16] K. Casey, D. Gregg, M. A. Ertl, and A. Nisbet, "Towards Superinstructions for Java Interpreters," in *Software and Compilers for Embedded Systems*. Springer, 2003, pp. 329–343.

[17] I. Piumarta and F. Riccardi, "Optimizing direct threaded code by selective inlining," *SIGPLAN Not.*, vol. 33, no. 5, pp. 291–300, May 1998.

[18] H. McGhan and M. O'Connor, "PicoJava: A Direct Execution Engine For Java Bytecode," *Computer*, vol. 31, no. 10, pp. 22–30, 1998.

[19] L.-R. Ton, L.-C. Chang, M.-F. Kao, H.-M. Tseng, S.-S. Shang, R.-L. Ma, D.-C. Wang, and C.-P. Chung, "Instruction Folding in Java Processor," in *Parallel and Distributed Systems, 1997. Proceedings., 1997 International Conference on*, 1997, pp. 138–143.

[20] C. Badea, A. Nicolau, and A. V. Veidenbaum, "A Simplified Java Bytecode Compilation System for Resource-Constrained Embedded Processors," in *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '07. New York, NY, USA: ACM, 2007, pp. 218–228.

[21] A. Azevedo, A. Kejariwal, A. Veidenbaum, and A. Nicolau, "High Performance Annotation-aware JVM for Java Cards," in *Proceedings of the 5th ACM international conference on Embedded software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 52–61.

[22] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2007, vol. 1009.

A light-weight compression method for Java Card technology

Massimiliano Zilli, Wolfgang Raschke, Reinhold Weiss and Christian Steger
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
{massimiliano.zilli;wolfgang.raschke; rweiss;steger}@tugraz.at

Johannes Loinig
NXP Semiconductors Austria GmbH
Gratkorn, Austria
johannes.loinig@nxp.com

ABSTRACT

Java Card is a Java running environment tailored for smart cards. In such small systems, resources are limited, and keeping application size as small as possible is a first order issue. Dictionary compression is a promising technique taken into consideration by several authors. The main drawback of this technique is a degradation in the execution speed.

In this paper we propose combining the dictionary compression with another compression technique based on the folding mechanism; the latter is less effective in terms of space savings, but has the advantage of speeding up the execution. A combination of the two techniques leads to higher space savings with a very low decrease in execution time compared with the plain dictionary compression.

Categories and Subject Descriptors

C.2.5 [**Special-purpose and application-based systems**]: Smartcards; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems; E.4 [**Coding and Information Theory**]: Data compaction and compression

General Terms

Languages, Experimentation, Measurements, Performance

Keywords

Smart card, Java Card, virtual machine, bytecode compression

1. INTRODUCTION

Smart cards are a very widespread technology, applied in the fields of banking, telecommunication and identification. Because of their large scale diffusion, these systems have to be cheap, hence with limited resources. Typical hardware configurations are based on a 8/16 bit processor, have some kilobytes of RAM and some hundreds of kilobytes of persistent memory. The applications running on these systems are often written in C or Assembly to keep the code size low and the performance high, but with the drawback of a low portability between different platforms. A virtual machine based system like Java resolves the portability problem, but the resources needed to run Java do not meet the smart card constraints. For this reason Java Card, a reduced set of the Java language specific for smart card applications, has been developed [8] [7]. Beyond the object-oriented programming language, Java Card offers a high security environment equipped with cryptographic functionalities and plays a role analogous to an operating system for the smart card.

The distribution of applications in Java Card takes place through the Java Card converted applet (CAP) file. The CAP file contains all the classes of the package application and it is organized in components. The Java Card environment uses the latter at installation time to install the application on the smart card.

Despite of the Java bytecode format being a compact instruction format, some research works based on dictionary compression go into the direction of compressing it, but at the price of a slower execution time. On the other hand research work regarding the speed-up of the bytecode execution does not usually take into consideration the ROM size as an issue, because they are applied to systems that are not as resource-constrained as smart cards.

In this work we focus on the compression of the method component by combining two techniques. The first one is based on the folding mechanism and substitutes foldable sequences of Java bytecodes with equivalent single superinstructions introduced as an instruction set extension of the virtual machine. The second one is based on the dictionary compression and substitutes repeated sequences of bytecodes with macros, whose definition is contained in a dictionary. Both techniques reduce the ROM size of the application, but while the second negatively affects the execution time of the application, the first speeds up its execution. Thanks to this approach we obtain better compression ratios paying a smaller price in terms of run-time performance compared to the plain dictionary compression.

The structure of the rest of the paper is as follows. Section 2 reviews the published research about code compression and execution speed-up that constitute the basis of this work. Section 3 provides a description of the new technique as a combination of the dictionary compression and the folding compression. Section 4 evaluates the proposed technique in terms of space savings and execution performance. Finally, in Section 5 we report our conclusions and outlooks on future work.

EWiLi'14, November 2014, Lisbon, Portugal.

2. RELATED WORK

In the context of embedded systems, keeping the application ROM size as small as possible is an important issue. It is even more important in the case of smart cards, where the memory size is smaller than in today's typical embedded systems. Compressing the executable code is one possible solution to overcome this problem, beyond following good programming practices.

Classic compression methods like Huffmann may demand resources that are not available in smart cards systems [11]. These methods usually need significant RAM memory for decompressing the entire information. Moreover, the decompression phase is time consuming and slows down the application execution, making the time constraints of the application domain hard to respect.

Dictionary compression does not have the limitations that prevent classical compression methods to be applied in low-end embedded systems [11]. It consists of the substitution of repeated sequences of information with a macro whose definition is stored in a dictionary. Claussen et al. introduce dictionary compression for low-end embedded systems running Embedded Java or Java Card [4]. The authors show that space savings up to 15% are achievable, but with an increase in execution time between 5% and 30%.

Systems based on virtual machines such as Java have a slower execution compared to systems where the applications are compiled in native machine instructions. The main approach present in most widely spread Java environments for speeding-up the execution is the *"Just In Time" (JIT)* compilation [5] [12]. It works by compiling sequences of frequently executed bytecodes directly into machine instructions during run-time. Throughout the compilation, the JIT mechanism performs optimizations within Java bytecodes sequences, making their execution faster compared to the plain Java bytecode interpretation. To store the temporary compiled code, JIT compilation makes use of remarkable quantities of RAM that are not available in smart cards.

One of the key-factors behind JIT compilation is the *superoperators* concept, introduced by Proebsting in [10]. According to it, a sequence of bytecodes can be reduced to a sequence of machine instructions where intermediate results are kept in registers instead of using the operand stack. A method for integrating superoperators in a low-end embedded system deploying Java consists of introducing superinstructions into the virtual machine instruction set [3] [9]. Thus, the new superinstruction can substitute the sequence of bytecodes equivalent to the superoperator. In addition to the advantages provided by the superoperators (e.g. less memory accesses), superinstructions need only one instruction fetch compared to the number of fetches needed during the execution of the sequence of bytecodes that they substitute.

A different approach to make Java environment faster is based on the use of a Java processor. The Java processor executes the Java bytecodes directly in hardware, given that the Java bytecodes constitute the machine instruction set. In [6], McGahm et al. propose *picoJava*, an example of a Java processor. Beyond the advantage provided by a direct hardware execution, picoJava has an optimization mechanism implemented in the *Instruction Folding Unit* [13]. This mechanism consists of analyzing the bytecodes to be executed next, and determining if they are foldable.

A software reproduction of the folding mechanism in a Java virtual machine is proposed in [2]. In that work, the virtual machine is able to recognize foldable instructions by means of Java annotations. Azevedo et al. introduce a similar approach for the Java Card environment [1]. The execution time improvement obtained within that work is up to 120%, but the class size increases up to 14%, because of the introduction of the annotations.

3. DESIGN AND IMPLEMENTATION

In this section we provide a brief description of the dictionary compression and the folding compression. Then, we present the light-weight compression technique analyzing how the two techniques interact. In the last subsection we discuss where the compression process can be inserted within the Java Card installation process.

3.1 Dictionary Compression

Dictionary compression consists of substituting repeated sequences of bytecodes with macros, whose definitions are stored into a dictionary. The dictionary can be static if it is used for every application, or dynamic if it is relative to a specific application. Dictionary compression can be plain, with wildcards or with generalized instructions [14]. The former case consists of completely substituting repeated sequences with a simple macro; in the latter cases the macros definitions are more general and can substitute similar sequences of bytecodes, keeping out of the definition the uncommon parts of the sequences as arguments of the macros. In this work we apply the plain dictionary compression, but the concept can be extended to the other two dictionary methods.

The sequences that can be substituted cannot be arbitrary, but they must respect the rule of being Single Entry Single Exit (SESE) blocks. Hence, no jumps are possible into the block except into the first instruction, and jumps are not possible from inside to outside the block but only within the block.

After all possible sequences have been found and grouped in sets of equal sequences that can be represented with the same macro, the most convenient superset of macros is selected. The number of elements of the superset is finite and corresponds to the number of undefined Java Card bytecodes reserved for the dictionary compression. In the Java Card standard 187 of the 256 possible values are defined bytecodes. In our experiments, we used only twelve undefined bytecodes for the dictionary compression; ten of them for one byte long macros (10 macros) and the remaining two for two byte long macros (2×255 macros), potentially allowing 522 macro definitions.

During bytecode execution, as sketched in Figure 1, when the Java Card virtual machine encounters a macro, it saves the return Java program counter (JPC_RET $\leftarrow$ JPC). In the second step, the virtual machine jumps through a look-up table into the corresponding dictionary definition (JPC $\leftarrow$ macroAddr). Afterwards, the execution of the Java bytecodes contained in the definition are performed. At the end of the macro definition, the execution of a special Java bytecode `ret_macro` will restore the Java program counter to the return value (JPC $\leftarrow$ JPC_RET).

3.2 Folding Compression

We developed the folding compression technique on the basis of the folding mechanism introduced for *picoJava*, a

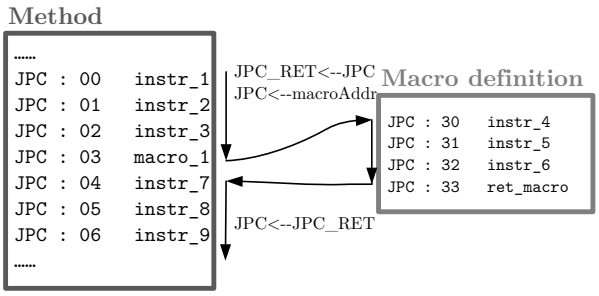

Figure 1: Execution flow in dictionary compressed code

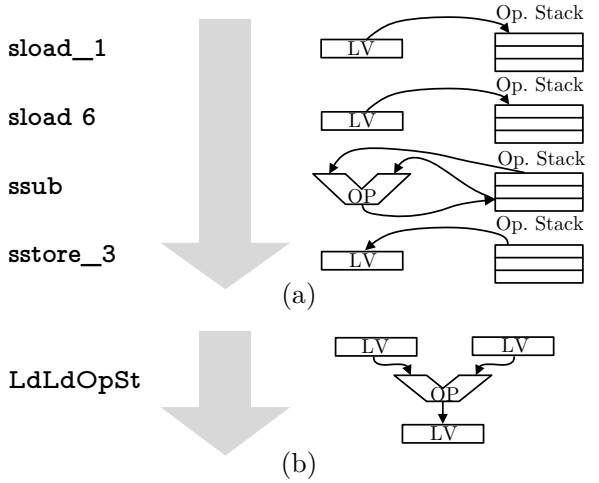

Figure 2: (a) Example of a foldable sequence; (b) Equivalent folded superinstruction

Java processor [13]. The bytecodes can be classified according to the usage of the Java Operand Stack as:

- *Producers* if they push an element onto the operand stack (e.g. `spush`, `sload`, ...)
- *Consumers* if they pop an element from the operand stack (e.g. `sstore`, ...)
- *Operands* if they pop one or two elements from the operand stack and they perform an operation (e.g. `sadd`, `sxor`, `ifeq`, `ifscmpeq`, ...)

Defined sequences of Java opcodes can be reduced to a single register machine like instruction. From now on, we will refer to these sequences as "foldable sequences". To clarify the concept, Figure 2 (a) reports a typical example of foldable sequence. The two initial load instructions (`sload_1` and `sload 6`) push the values of two local variables onto the stack, the subtract instruction (`ssub`) pops them, executes their addition and pushes the result onto the stack. Finally, the store instruction (`sstore_3`) pops the value on the stack and stores it in a local variable. The entire sequence can be substituted with a single register like instruction (Figure 2 (b)) that takes the values directly from the local variables,

Instruction	Argument	Opt. Arg.
LdSt(PC)	B1[St:Ld]	-
PshSt(PC)	B1[Op:Cnst]	B2[BPsh]B3[SPsh]
OpSt(PO)	B1[St:Op]	-
LdIf_s2b(PO)	B1[Op:Ld]	B2[Br]B3[Brw]
LdPshAdd(PPO)	B1[Cnst:Ld]	B2[BPsh]B3[SPsh]
LdPshOp(PPO)	B1[Cnst:Ld]B2[Op:Ord]	B3[BPsh]B4[SPsh]B5[Br]B6[Brw]
LdLdOp(PPO)	B1[Ld2:Ld1]B2[Op]	B3[Br]B4[Brw]
LdPshOpSt(PPOC)	B1[Cnst:Ld]B2[St:Op]	B3[BPsh]B4[SPsh]
PshLdOpSt(PPOC)	B1[Ld:Cnst]B2[St:Op]	B3[BPsh]B4[SPsh]
LdLdOpSt(PPOC)	B1[Ld2:Ld1]B2[St:Op]	-

Table 1: Instruction set extension

performs their addition and stores the results on the destination local variable. The use of such a register-like instruction saves three instruction fetches and avoids all the memory writes and reads to and from the operand stack.

Like dictionary compression, the folding compression also makes use of undefined Java Card bytecodes. In the folding compression case, ten undefined bytecodes are used for extending the Java Card instructions set with the new superinstructions. In Table 1, we report all the new folded superinstructions forming the instruction set extension. The superinstructions consist of an initial byte identifying the type of instruction, followed by a variable number of bytes constituting the argument. In the first column of the table, near the mnemonic of the superinstruction, there is the kind of sequence (in terms of (P)roducer, (C)onsumer, (O)perand classification) that the superinstruction substitutes.

The new superinstructions do not cover all the possible foldable sequences but only the most frequent ones. In fact, by means of the arguments, one superinstruction can represent many combinations (i.e. *LdLdOpSt* can represent the `sload_1 sload 6 ssub sstore_3` sequence as well as the `sload_3 sload_1 sxor sstore 10` sequence). Moreover, within the instructions belonging to the foldable sequences, the load and the store instructions are covered only for the first sixteen local variables. This allows to encode the local variable index with only four bits, thus we can express two load instructions with a one byte long argument. Covering only the first sixteen local variables allows to cover most of the cases anyway; the analysis on a set of three industrial applications (i.e. a statistically sound set of bytecodes combinations) points out a coverage of about 95% of all the foldable sequences.

To clarify how the space savings are obtained, we look again at the example of Figure 2, where we can compare the foldable sequence with the equivalent folded superinstruction. The first byte argument of the latter will be 0x61 whose digits indicate respectively the first and the sixth local variable for the load operations; the second byte will be 0x31 whose digits indicate respectively the subtraction operation and the third local variable for the store operation. Compared with the foldable bytecode sequence that occupies five bytes of ROM memory, the new superinstruction occupies only three bytes allowing space savings of two bytes.

3.3 The light-weight Compression

The combination of the folding compression and the dictionary compression constitutes the light-weight compres-

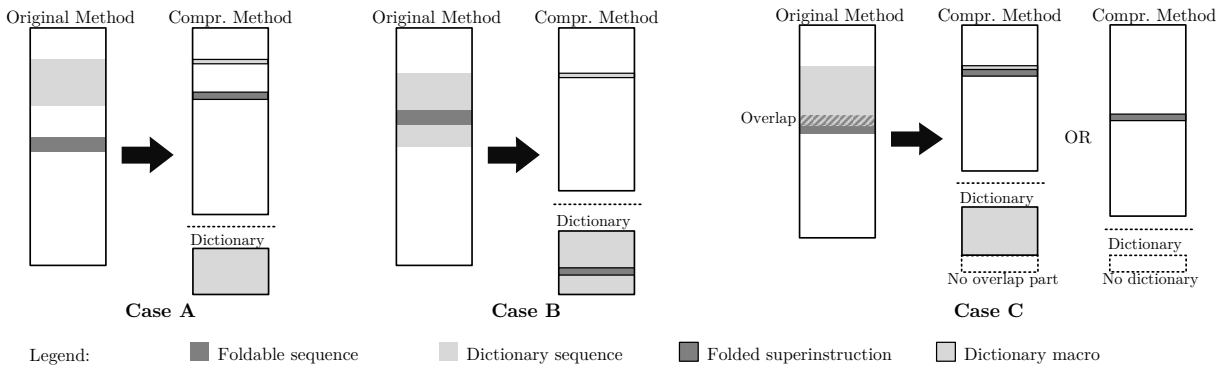

Figure 3: Possible cases in the combination of the compression techniques

sion. The first step consists of the compression with the folding method, while in the second step the application resulting from the first step is compressed with the dictionary method. The two techniques coexist with a small interference that affects the dictionary compression space savings. Figure 3 shows the three possible cases of interaction.

In case A the foldable sequence and the dictionary sequence are separated, hence there is no interference and the space savings due to the dictionary compression does not degrade. In Case B the foldable sequence is contained into the dictionary sequence, and it becomes part of the dictionary definition. It is also possible that the foldable instruction identifies with the dictionary definition, but the dictionary definition cannot be contained into the foldable sequence. Case C presents the case in which the foldable sequence and the dictionary sequence partially overlap. In this case, either the dictionary sequence is not substituted, or the dictionary definition is shortened, depending on the convenience for the overall space savings. However, in case C and B the space savings owing to the dictionary compression diminishes. We can express the overall space savings S due to the light-weight compression as:

$$S = S_f + (1 - k_1) \cdot S_d$$

where S_f and S_d are the space savings due to the folding compression and the dictionary compression respectively, and k_1 is the coefficient that expresses the degradation of the dictionary compression due to the interference with the folding compression. The coefficient k_1 ranges between 0 and 1; where a value of 0 means absence of interference, and hence the final space savings is the arithmetical sum of the two partial space savings.

The two techniques have opposite effects on the execution time. While the dictionary compression decreases the execution speed, the folding compression increases it. Also in this case, if there is interference from the folding compression on the dictionary compression, the slowing effect on the run-time provided by the dictionary compression is mitigated. The second option of case C in Figure 3 is the only case in which the execution performance effect due to the dictionary compression is reduced. Hence, the interference of the two techniques in the execution performance is lower than the interference they have in the space savings. The overall run-time effect R due to the application of the light-weight compression can be expressed with the formula:

$$R = R_f + (1 - k_2) \cdot R_d$$

where R_f and R_d are the effect on the execution time due to the folding compression and to the dictionary compression, respectively, and k_2 is the coefficient that accounts for the reduction of the dictionary compression due to the interference from the folding compression. In this case R_d is negative, because the dictionary compression slows the application execution; hence, a value of k_2 greater than 0 would positively affect the overall effect of the light-weight compression on the execution time.

Summing up, the combination of the two compression techniques leads to space savings that approximately equal the sum of the space savings due to the two techniques. Regarding the speed performance, the two techniques compensate each other.

3.4 Integrating the light-weight compression into the JCVM

The installation process in Java Card is different than in Java. It is split in two parts: one off-card and the other on-card. After the compilation and the creation of the class file, the off-card Java Card converts the class file into a CAP file, which is the distribution format of the application. The installation on the smart card starts with the verification of the CAP file. During this operation, the off-card Java Card checks the validity of the CAP file assuring a secure installation. At this point, the CAP file is handled by the off-card installer that establishes a communication channel with the on-card installer. The installer transfers and instantiates the application on the smart card. Once that the application is installed, the application exists until it is uninstalled; between a power-down and a power-on of the smart card, the application status is saved into non-volatile memory.

In this architecture, the most convenient point to perform the compression is after the verification and before the installation. In this way, the distributed CAP file is general for all Java Card systems, whether they are enabled with the light-weight compression or not. Moreover, the off-card Java Card is able to distinguish between an on-card Java Card enabled for the light-weight compression and one that is not only at installation time. Therefore, the Java Card verifier can be common for each Java Card and does not need to know the extended Java Card instruction set used

Application	Size [B]	Space Savings [%]	
		Dict. Compr.	Fold. Compr.
XPay	1784	12.28	6.73
MChip	23305	9.16	4.02
MChip Advanced	38255	10.52	3.72
BubbleSort	239	5.44	2.51
BigInteger	650	3.39	1.54

Table 2: Applications memory size and partial space savings

Application	Space Savings [%]
XPay	15.70
MChip	12.43
MChip Advanced	11.73
BubbleSort	6.70
BigInteger	4.46

Table 3: Space savings of the light-weight compression

for the compression.

4. RESULTS AND DISCUSSION

In this section we report the space savings and the run-time analyses obtained with the proposed technique. We first analyze the folding compression and the dictionary compression, separately; afterwards, we report the result of their interaction in the light-weight compression technique.

4.1 Space Savings

For the assessment of the space savings, we took into consideration a set of three industrial banking applications (MChip, MChip Advanced and XPay). We also developed two test-benches that we used for the evaluation of the execution performances. The first test-bench performs a "bubble sorting", while the second implements a basic big-integer class and performs a sequence of operations on big-integer variables. The space saving S_{Xcompr} owing to the compression technique $Xcompr$ is defined as

$$S_{Xcompr} = \frac{AppSize_{original} - AppSize_{Xcompr}}{AppSize_{original}}$$

where $AppSize_{original}$ is the application size of the original application and $AppSize_{Xcompr}$ is the application size after the compression with $Xcompr$.

Table 2 and Table 3 list all the space savings obtained during the evaluation. The third and the fourth column of Table 2 report the space savings obtained with the folding compression and with the dictionary compression, respectively. We see that the dictionary compression performs better. In the second column of Table 3 we can see the space savings for the light-weight compression. Comparing the two tables, we see that the effects of the two techniques are concordant and their combination (average space savings of 12%) is better than a pure dictionary compression (average space savings of 10%) with an improvement of 20%.

In the results reported above, we took into consideration the dictionary ROM space, but not the additional ROM space needed for the implementation of the extended instruction set needed for the folding compression. The additional ROM space needed for the implementation of the extended instruction set is about 5kB. Java Card environment is designed for hosting multiple applications in a single card. Hence, if we consider an average space savings of 12% and an average applet size of 20kB, we can estimate that the installation of two applets will balance the additional ROM space needed for the instruction set extension (to be more precise, the set of applications should occupy at least 42kB).

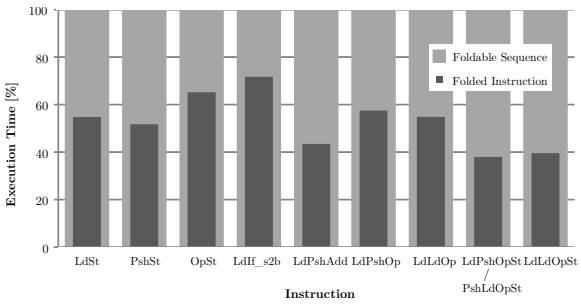

Figure 4: Execution speed-up for foldable sequences

4.2 Run-time performance

For the evaluation of the run-time performance, we took as a starting point the Oracle Java Card reference implementation, that we ported to the 8051 architecture, which is a plausible platform for a smart card. Afterwards, we added the instruction set extension for the folded instructions, and the mechanism for managing the dictionary compression.

Regarding the dictionary compression, we evaluated the increase in the execution time owing to the macro execution. For this purpose, we measured the time needed for the execution of a bytecode sequence whose length corresponds to the average number of bytecodes in the dictionary definitions of the test-set of industrial applications. We found that the execution time of a dictionary macro increases by about 50%, compared to the execution time of the average sequence contained in the dictionary definition. This increase is due to the jump through the look-up table to the dictionary definition and to the execution of the `ret_macro` instruction, as already discussed in Section 3.

To evaluate the folding compression mechanism, we compared the time needed for the execution of a foldable sequence of Java bytecodes with the time needed for the execution of the equivalent folded instruction belonging to the extended instruction set. Figure 4 shows the comparison; for each instruction of Table 1, the background bar (light gray) represents the time needed for the execution of the equivalent foldable sequence, whereas the foreground bar (dark gray) accounts for the execution time of the folded instruction. The execution of the folded instructions is about two times faster compared to the execution of the equivalent foldable sequences.

The industrial applets used for the assessment of the space savings make use of proprietary libraries that are not available in the reference implementation. For this reason we performed our test on our test-bench applications. We define the execution speed-up U_{Xcompr} for the generic compression

Application	Execution Time [%]		
	Dict. C.	Fold. C.	Light W. C.
BubbleSort	+3.2	-6.8	-3.8
BigInteger	+1.7	-4.0	-2.2

Table 4: Applications execution time

technique $Xcompr$ as

$$U_{Xcompr} = \frac{ExTime_{original} - ExTime_{Xcompr}}{ExTime_{original}}$$

where $ExTime_{original}$ is the execution time of the original applet, and $ExTime_{Xcompr}$ is the execution time of the applet compressed with the generic technique $Xcompr$. Table 4 reports the measurements on the execution time after the application of the different compression techniques; the results expressed in percentage are relative to the execution of the original applications. We point out that the dictionary compression slightly slows down the execution time, while the folding compression significantly speed it up. This behavior derives from the nature of the test-benches that have a small ROM size (dictionary compression is less effective in small application where there is a lower probability of repeated sequences) and a high computation level (folding compression is more effective in parts of code involved in computation). Considering the space savings of the industrial applications, we expect a slight slow-down of the execution after the application of the light-weight compression because of the dominance of the dictionary compression. The slow-down will be anyway lower compared to the case where only the dictionary compression is applied.

5. CONCLUSIONS

In this work we have proposed a novel compression technique for applications running on smart cards enabled with the Java Card System. The compression technique is the result of the combination of two compression methods: the dictionary compression and the folding compression. While the former pays for a good compression ratio with a higher application execution time compared to the original application execution time, the latter has lower space savings but offers at the same time a speed-up of the application execution. The final result is a light-weight compression method with an average space savings of 12% and a slight execution slow-down; compared to the plain dictionary compression, the light-weight compression has higher space savings and causes a lower slow-down in the execution of the application.

Providing a hardware support with an extension of the microcontroller instruction set specifically for the light-weight compression technique seems to be promising for resolving the execution slow-down, and it will therefore be the object of investigation in future research work.

6. ACKNOWLEDGMENTS

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank their project partner NXP Semiconductors Austria GmbH.

7. REFERENCES

[1] A. Azevedo, A. Kejariwal, A. Veidenbaum, and A. Nicolau. High Performance Annotation-aware JVM for Java Cards. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 52–61, New York, NY, USA, 2005. ACM.

[2] C. Badea, A. Nicolau, and A. V. Veidenbaum. A Simplified Java Bytecode Compilation System for Resource-Constrained Embedded Processors. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 218–228, New York, NY, USA, 2007. ACM.

[3] K. Casey, D. Gregg, M. A. Ertl, and A. Nisbet. Towards Superinstructions for Java Interpreters. In *Software and Compilers for Embedded Systems*, pages 329–343. Springer, 2003.

[4] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java Bytecode Compression for low-end Embedded Systems. *ACM Trans. Program. Lang. Syst.*, 22(3):471–489, May 2000.

[5] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java Just in Time. *Micro, IEEE*, 17(3):36–43, 1997.

[6] H. McGhan and M. O'Connor. PicoJava: A Direct Execution Engine For Java Bytecode. *Computer*, 31(10):22–30, 1998.

[7] Oracle. *Java Card 3 Paltform. Runtime Environment Specification, Classic Edition. Version 3.0.4*. Oracle, September 2011.

[8] Oracle. *Java Card 3 Paltform. Virtual Machine Specification, Classic Edition. Version 3.0.4*. Oracle, September 2011.

[9] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. *SIGPLAN Not.*, 33(5):291–300, May 1998.

[10] T. A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '95, pages 322–332, New York, NY, USA, 1995. ACM.

[11] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag New York Incorporated, 2004.

[12] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[13] L.-R. Ton, L.-C. Chang, M.-F. Kao, H.-M. Tseng, S.-S. Shang, R.-L. Ma, D.-C. Wang, and C.-P. Chung. Instruction Folding in Java Processor. In *Parallel and Distributed Systems, 1997. Proceedings., 1997 International Conference on*, pages 138–143, 1997.

[14] M. Zilli, W. Raschke, J. Loinig, R. Weiss, and C. Steger. On the Dictionary Compression for Java Card Environment. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, pages 68–76. ACM, 2013.

2014 17th Euromicro Conference on Digital System Design

A High Performance Java Card Virtual Machine Interpreter based on an Application Specific Instruction-Set Processor

Massimiliano Zilli, Wolfgang Raschke,
Reinhold Weiss, and Christian Steger
Graz University of Technology
Institute of Technical Informatics
Graz, Austria
Email: {massimiliano.zilli,wolfgang.raschke, rweiss,steger}@tugraz.at

Johannes Loinig
NXP Semiconductors Austria GmbH
Gratkorn, Austria
Email: johannes.loinig@nxp.com

***Abstract*—Java Card is a Java running environment specific for smart cards. In such low-end embedded systems, the execution time of the applications is an issue of first order. One of the components of the Java Card Virtual Machine (JCVM) playing an important role in the execution speed is the bytecode interpreter. In Java systems the main technique for speeding-up the interpreter execution is the Just-In-Time compilation (JIT), but this resource consuming technique is inapplicable in systems with as restricted resources available as in smart cards.**

This paper presents a hardware/software co-design solution for the performance improvement of the interpreter. In the software domain, we adopted a pseudo-threaded code interpreter that allows a better run-time performance with a small amount of additional code. In the hardware domain, we proceeded moving parts of the interpreter into hardware, giving origin to a Java Card interpreter based on an application specific instruction set processor.

***Keywords*-Smart card; Java Card; interpreter; application specific instruction set processor;**

I. Introduction

Smart cards are nowadays a widespread technology used mainly in the field of banking, e-government, and identification. Typical hardware configurations are based on 8/16 bit processors with some kilobytes of RAM and up to a few hundred kilobytes of non-volatile memory, distributed between ROM, Flash and EEPROM memory. In such small devices, the programming language for the applications development is often C or assembly in order to keep the code size small and the performance high. Issues that arise with these kinds of languages are the portability and the update of the applications. An interpreted language like Java alleviates these problems, also adding a high degree of security to the run-time environment.

Because of the resource constraints in smart cards, the adoption of a complete Java standard is unfeasible. Java Card standard is a reduced version of Java targeted for smart cards [1] [2]. Java Card inherits the main features of Java, easing object oriented programming and the *compile once run anywhere* feature. The applications for Java Card are distributed in form of CAP files for the executable binaries and of export files for the interface binaries. The CAP file contains the intermediate code in form of bytecodes that is the result of the Java compilation. In this paper we use the terms "Java bytecode" to indicate the result of the Java compilation and to distinguish the latter from the machine instructions (opcodes), which constitute the instruction set of the processor. Once the application has been installed on the smart card, its execution starts and continues until the application is uninstalled.

Like Java, Java Card Virtual Machine has an interpreter whose task is the interpretation of the Java bytecodes. The Java interpreter is a critical part of the Java Card virtual machine (JCVM), since it directly affects the execution time of the applications. This aspect is very important in industry, since applications often have very strict requirements. In standard Java, the interpreter has been subject of many optimization techniques aimed to improve the execution speed, but these techniques are not applicable on smart cards, because of the limited resources available.

In this paper, we propose a hardware/software co-design solution that improves the performances of the Java Card interpreter. In the software domain we adopted a pseudo-threaded code solution for the interpreter; in the hardware domain, we integrated parts of the interpreter into the microcontroller architecture. We investigated three solutions with different degrees of integration of the interpreter into the microcontroller architecture and compared them to a classic interpreter implementation.

The structure of the rest of the paper is as follows. Section II reviews the previous literature related to this research. Section III provides a functional description of the hardware aided interpreter architectures. Section IV gives an overview of the design and the implementation for the software and hardware part of the Java Card interpreter. Section V presents the results on the performance evaluation of the proposed interpreters. In Section VI we report our

978-1-4799-5793-4/14 $31.00 © 2014 IEEE
DOI 10.1109/DSD.2014.47

conclusions and an outlook on future work.

II. Related Work

The interpreter of Java Card Virtual Machine is usually a classic interpreter ideally based on a huge switch wrapped by a while loop [3]. This solution is simple and compact in terms of ROM code size, but suffers from low speed performance. Nevertheless, the classic interpreter is a clean solution that executes hardware independent code, and hence, complies with the *compile once run anywhere* model. An alternative interpreter is the direct threaded interpreter (DTI) [4] [5]. This interpreter is based on a compiler that produces a machine depended code. In fact, the executable code consists of a sequence of subroutine addresses that have to be handled; the presence of machine specific addresses into the executable code makes the latter not portable. The portability problem is solved in [5] separating the compilation of the application in two phases: the first produces a preliminary code that is portable, while the second creates the threaded code.

An application of the DTI to the Java System is proposed in [6]. Analogously to [5], the problem of the portability is overcome by adopting a pre-execution phase that transforms the Java bytecode into threaded code. This solution hardly fits into a Java Card environment, because of the remarkable increase of the executable code size (i.e. in an architecture with 16-bit wide address space, the executable threaded code would be about twice the size of the original bytecode).

The main approach to reduce execution time present in most widely spread Java environments is the *Just-In-Time* (JIT) compilation [7] [8] [9]. Instead of being intepreted, the bytecodes are compiled into machine code performing optimizations that make their execution faster than a normal interpretation. Although this mechanism is very effective in general purpose systems and high-end embedded systems, it is not applicable in smart cards because of the high amount of RAM it needs for storing the compiled code.

To overcome the issue of limited resources in smart cards, some authors have proposed alternative methods. Azevedo et al. introduced an *Annotation Aware Virtual Machine* able to recognize annotations that indicate foldable sequences [10]. Hence, during the execution, when the virtual machine encounters an annotation, it executes the superoperator relative to the foldable sequence instead of the normal bytecode sequence. Since the superoperator is an optimized form of the bytecode sequence, the execution is faster than in the plain interpretation.

Another direction in which research has gone for enhancing performance in Java is the hardware implementation of the Java Virtual Machine. The hardware acceleration can be achieved in two main ways, a direct Java bytecode execution or a Java bytecode translation. McGham et al. proposed picoJava [11], a Java processor that executes Java bytecodes directly on hardware. The Java virtual machine is completely implemented in hardware and the Java bytecodes constitute the instruction set of the processor. In this model there is no longer an interpreter, because the processor executes the Java bytecode natively, with outstanding runtime performance. A problem of this architecture is the integration with established operating systems or existing applications, since the processor is not able to execute programs written in other programming languages. An example of bytecode translation into native machine op-code sequences is ARM Jazelle [12]. The integration of the Jazelle with existing operating systems and the concurrent execution of other applications written in other programming languages is possible using the extended instruction set.

The instruction set extension is common practice in application specific instruction set processor (ASIP) [13] [14] [15]. With the new opcodes of the instruction set, it is indeed possible to activate hardware functionalities added for the purpose of enhancing the performance of a specific application. In the solution that we propose in this paper we extended the instruction set of a microcontroller to support a pseudo-threaded interpreter whose fetch-decode part is executed in hardware. The execution phase of the Java bytecodes is kept in software for taking advantage of its flexibility. The latter is indeed necessary, for example, to add security checks inside the Java bytecode functions [16].

III. Interpreter

This section presents the evaluated solutions for the interpreter in a functional view. The first subsection regards software interpreters running on standard architecture, while the last two regard two interpreters, where parts of the functionality are moved to hardware.

A. Software Interpreter

An interpreter is a software entity that executes instructions without compiling them into machine instructions. In the case of Java language, the interpreter executes an intermediate representation of the Java source code. The intermediate representation is a sequence of Java bytecodes and is the product of the Java compiler.

The interpreter activity can be subdivided in three main phases: fetch, decode and execute. The fetch phase consists in reading the instruction (the Java bytecode). The decode phase recognizes the actual Java bytecode and makes the execute flow of the microcontroller start the right sequence of machine instructions that constitute the Java bytecode. The execution of the machine instructions constituting the Java bytecode represents the execute phase of the interpreter. The three phases are repeated sequentially for each Java bytecode of the application, analogously to what happens in a hardware processor that executes the machine instructions of an application compiled natively.

Figure 1 shows a plain example of a Java Card interpreter written in pseudo-code. The while loop allows to reiterate

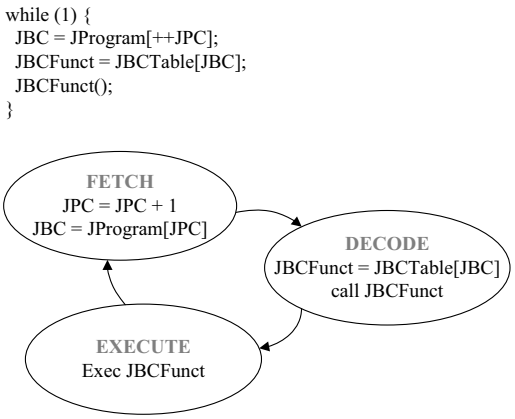

Figure 1. Pseudo-code and state machine of the Java Card interpreter

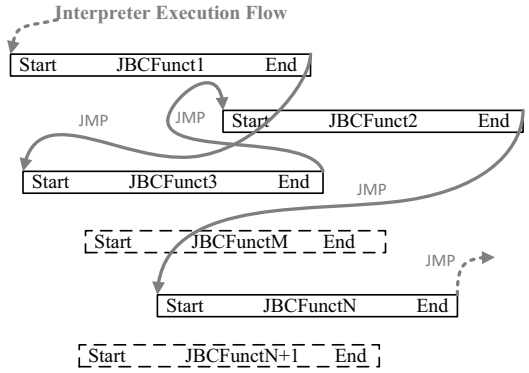

Figure 2. Representation of the behavior of the pseudo-threaded interpreter

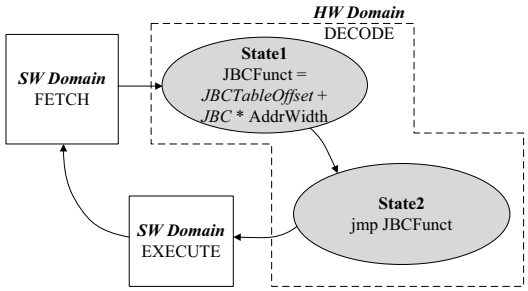

Figure 3. State machine for the interpreter with decode phase in hardware

the fetch-decode-execute phases cyclically. The first line of code inside the while loop represents the fetch phase; JPC is the Java program counter and JProgram is the bytecode to be executed. In the second line, the interpreter extracts from a look-up table (JBCTable) the address corresponding to the Java bytecode function. The third line consists in the execution of the actual bytecode function. In literature this kind of interpreter is also known with the name of token-threaded interpreter [17]. In the finite state machine of the figure, the call of the function is part of the decode phase while the execution of the function represents the execute phase of the interpreter. In this interpreter we count a call to the Java bytecode function with the relative return instruction and a final jump at the beginning loop.

To improve the execution speed, we propose an interpreter that is not based on a loop and that we will call pseudo-threaded interpreter. While in the classic interpreters the fetch-decode phase is centralized in the main loop, in the pseudo-threaded interpreter each Java bytecode function ends with the fetch and decode phase and jumps to the next Java bytecode function to execute. The name of the interpreter derives from its similarity to the threaded interpreter, where the execution flow jumps from each Java bytecode function to the next one, as shown in Figure 2. With the exception of this similarity, the pseudo-threaded interpreter maintains the look-up table like the classic interpreter and performs the fetch and decode phase in the same manner. The drawback of this solution is the growth of the code size of the interpreter, because of the repetition of the fetch-decode code in each Java bytecode function. On the other hand, the execution flow does not return to a main loop, using a jump instead of a call and saving the use of the return instruction and of a jump instruction compared to the classic interpreter.

B. ASIP based Interpreter

In the previous subsection we presented the pseudo-threaded interpreter. In the latter, the final part of each Java bytecode function is dedicated to the fetch-decode of the next Java bytecode, increasing dramatically the code size of the interpreter. Moreover, since the fetch and decode phases of the interpreter are performed at every bytecode interpretation, the improvement of the two phases is a key factor for the execution speed. In this context, to improve the run-time performance, we adopted a hardware/software co-design to introduce application specific hardware features into the microcontroller architecture. We developed two solutions: one executes directly in hardware the decode phase, and the other executes directly in hardware both the fetch and the decode phases.

1) Hardware Decode: Within this design approach, the decode phase is situated in the hardware domain, while the fetch phase remains in the software domain. The microarchitecture has to be aware of the position of the Java bytecode functions table. The argument of the functionality is the Java bytecode to be decoded. Figure 3 reports the finite state machine representing the functionality. After the fetch of the Java bytecode in the software domain, the first step of the hardware decode fetches from the look-up table the address

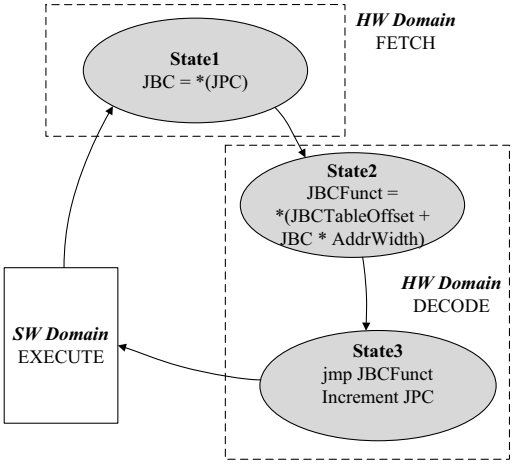

Figure 4. State machine for the interpreter with fetch-decode phase in hardware

of the Java bytecode function, and calculates the absolute address within the look-up table with the formula

$$JBC_{Funct} = JBCTable_{StrtAddr} + JBC * AddrWidth$$

where $JBCTable_{StrtAddr}$ is the starting address of the look-up table containing the addresses of the Java bytecode functions, JBC is the actual Java bytecode fetched by the interpreter, and $AddrWidth$ is the ROM memory address bus width expressed in bytes (i.e. 2 bytes for a standard 8051 architecture). In the second step, the state machine jumps to the address fetched. Henceforth, the interpreter continues in the software domain and executes the Java bytecode function. The state machine has a behavior similar to a jump instruction that has its target depending on the Java bytecode it has to elaborate.

2) Hardware Fetch-Decode: While in the previous approach we moved to hardware only the decode part, here also the fetch phase is executed in hardware. In the first step of the finite state machine in Figure 4, the Java bytecode is fetched according to the value of the variable JPC. Once the bytecode is fetched, the state machine proceeds as the state machine of the previous subsection with the calculation of the absolute address in the bytecode table, the fetch of and the jump to the address of the next operand function, but with the additional increment of the JPC. Every time the hardware part of the interpreter is active, its functionality depends on the status of the interpreter at the last activation, status represented by the JPC. After the hardware decode phase, the interpreter switches in the software domain for the execution of the Java bytecode.

IV. Design and Implementation

Due to the fact that Java Card is a subset of Java designed for smart cards, we chose to implement our system in a microcontroller architecture that is typically used for smart card application. The 8051 microcontroller fits well into the smart card context, where cheapness and low power consumption are requirements of the first order. The main characteristics of the architecture are an 8-bit arithmetic and a 16-bit addressable space. The considered 8051 core is distributed by Oregano under LGPL license [18]. The design is fully synchronous, allowing the core to be up to ten times faster than conventional 8051 architecture.

For the software development, we used the Keil μVision4 IDE, whose compiler supports the Oregano 8051 microcontroller.

A. Classic Interpreter

Compared to the classic version in Figure 1 used for the sake of a clear explanation, the implemented version is more compact and avoids time consuming passages into intermediate variables. The variable *JBC* and the function pointer *JBCFunct* are not used anymore and the internal of the while loop consists on the single line instruction $JBCTable[JProgram[++JPC]]()$. The actual Java bytecode pointed by the Java program counter (*JPC*) is fetched from the Java program (*JProgram*); the Java bytecode is used as the index of the look-up table (*JBCTable*) to select the corresponding Java bytecode function to execute.

B. Pseudo-threaded Interpreter

As anticipated in Section III-A, for implementing an interpreter with pseudo-threaded code, the final part of every Java bytecode function has to be able to fetch the next Java bytecode, decode it, and perform the jump to the next Java bytecode function. A direct jump to a function is possible with the GCC compiler using the "labels as value" feature [19]. The latter is neither available in ANSI C, nor in the Keil Compiler. Hence, we added a portion of assembly code at the end of each Java bytecode function to implement the threaded code as Figure 5 reports in pseudo-code for the 8051 processor. The pseudo-code does not take into consideration the register and address length for the sake of a better readability. In the 8051 architecture, this portion of assembly code requires 49 bytes for every Java bytecode instruction. In Java Card technology, where 188 Java bytecodes are defined, the code threaded feature has a code overhead of about 9kB. In smart cards, where the resources are limited and a typical Java Card implementation has a ROM size in the order of 100kB, such an amount of ROM memory overhead could be an issue.

C. Interpreter with Decode in Hardware

For the realization of the decode phase in hardware, we already claimed in Section III that the system needs to know the address of the look-up table where all the addresses of the Java bytecode functions are stored. In the hardware implementation, we added a special function register (SFR)

```
void JavaBytecodeFunctionN (void) {
  /*
  Execution part of the bytecode
  */
#pragma asm
  ;;Fetch
  MOV R0, JPC
  INC @RO
  MOV A, JProgram
  MOV DPTR, A
  MOV A, @R0
  MOV A, @A+DPTR
  ;;Decode
  MOV B, #02
  MUL AB
  MOV DPTR, A
  MOV A, @DPTR
  MOV DPTR, A
  JMP DPTR
#pragma endasm
}
```

Figure 5. Pseudo-code for the interpreter with pseudo-threaded code

that allows to store the address of the Java bytecode table and to access it quickly. The additional information needed to extract the right function address from the look-up table is represented by the Java bytecode and is supplied to the microcontroller by means of the accumulator register. The state machine of Figure 3 has been integrated in the main finite state machine of the 8051 microcontroller. The main additions for the integration of the described functionality consist of the adder and the multiplier needed for the calculation of the absolute address within the Java bytecode table.

To interface the new functionality with the software layer, we extended the instruction set of the microcontroller using undefined opcode A5. More in detail, in the executable code, the byte after the A5 opcode tells the microcontroller which of the new interpreter specific machine functions to execute. In fact, beyond the main opcode representing the jump to the next Java bytecode function, we added an instruction for the initialization of the SFR containing the address of the Java bytecode functions table.

In the software domain, the assembly part at the end of the Java bytecode functions is reduced to the sequence in Figure 6. The final assembly instruction JMPJTABLE is the mnemonic for the new jump opcode discussed above. The ROM space needed for the new sequence of assembly instructions at the end of each Java bytecode function amounts to 26 bytes. The ROM space overhead for the 188 Java bytecodes is about 4.8kB.

```
void JavaBytecodeFunctionN (void) {
  /*
  Execution part of the bytecode
  */
#pragma asm
  ;;Fetch
  MOV R0, JPC
  INC @RO
  MOV A, JProgram
  MOV DPTR, A
  MOV A, @R0
  MOV A, @A+DPTR
  ;;Decode
  JMPJTABLE
#pragma endasm
}
```

Figure 6. Pseudo-code for the interpreter running on an architecture with decode phase in hardware

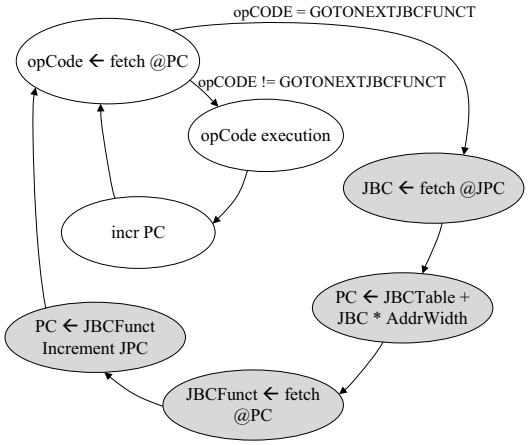

Figure 7. Execution flow in the 8051 architecture for the new instruction performing the fetch and decode phase of the Java interpreter

D. Interpreter with Fetch and Decode in Hardware

The interpreter with fetch and decode in hardware is an extension of the system described in the previous subsection. Beyond the introduction of an SFR for storing the address of the Java bytecode function table, we introduced also an internal register for storing the Java Program Counter (JPC). In the internal architecture, the JPC and the microcontroller Program Counter (PC) have a similar behavior, and interact during the fetch and decode phase of the interpreter. To explain the model, Figure 7 sketches the execution flow resulting from the combined use of the two registers within the modified 8051 microcontroller. When the hardware interpreter is activated, the PC is inhibited and the next processor fetch is based on the content of the JPC that points to the

```
void JavaBytecodeFunctionN (void) {
  /*
  Execution part of the bytecode
  */
#pragma asm
  ;;Fetch and Decode
  GOTONEXTJBCFUNCT
#pragma endasm
}
```

Figure 8. Pseudo-code for the interpreter running on an architecture with fetch-decode phase in hardware

next Java bytecode; the PC is then used again for fetching the address of the next Java bytecode function from the look-up table. Finally, the address of the Java bytecode function is loaded into the PC and the JPC is incremented. Henceforth, the control returns to the standard microcontroller mode, until the next activation of the state machine.

As in the previous subsection, to make the activation of the fetch-decode state machine from the software domain possible, we extended the instruction set of the microcontroller with interpreter specific machine instructions. With the availability of the new instruction, the final part of the Java bytecode functions looks like in Figure 8. The instruction GOTONEXTJBCFUNCT is the mnemonic for the new opcode. The new opcode needs only two bytes; thus the ROM memory needed to implement pseudo-threaded code for the 188 Java bytecodes functions of Java Card amounts to 376 bytes. Furthermore we added set and get opcodes for the manipulation of the JPC. As for the interpreter with hardware decode there is an instruction for setting the SFR with the address of the look-up table containing the addresses of all the Java bytecode functions.

In Java Card standard, there are also bytecodes provided with arguments. The latter have to be accessible within the Java bytecode function, but their location is bound to the information contained in the JPC. Therefore, we provided an additional opcode, whose functionality is to fetch the byte pointed by the JPC, to copy it in the accumulator, and to increment the JPC by one unit.

1) Bounds Check: To counteract control-flow attacks like EMAN4 [20], the interpreter have to perform checks on the JPC against specific bounds [21]. Since in our implementation the fetch-decode is completely executed in hardware, we introduced the JPC bounds check in hardware. For this purpose, we added two 16-bit registers for storing the values of the upper and lower limits of the JPC with the relative set opcodes. If the JPC has a value out of the bounds, an interrupt is activated and, eventually, an interrupt service routine can be executed.

```
#pragma ___JBytecodeThreadFunct___

void JavaBytecodeFunctionN (void) {
  /*
  Execute part of the bytecode
  */
}
```

Figure 9. Example of pragma usage for marking Java bytecode functions

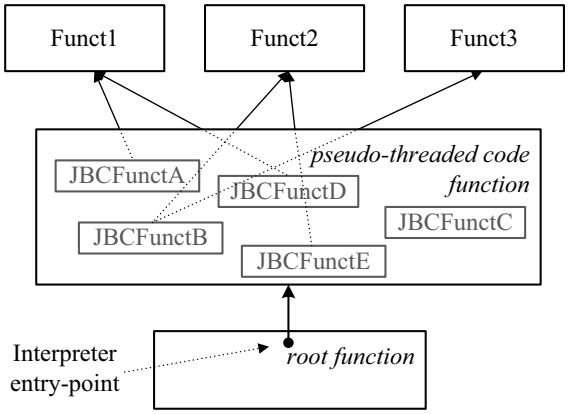

Figure 10. Schematic of the call-tree for the pseudo-threaded interpreter

E. Outlook to the Compiler Integration

In this subsection we spend a few words on the feasibility of the integration of the new feature into the compiler. In Figure 9 we report a template of Java bytecode function written in C. The compiler will use the information given by the *pragma* statement to identify the functions implementing Java bytecodes, and to deploy the latter in the call-tree as schematized in Figure 10. In fact, the threaded code functions are never called, but each function belonging to this set ends jumping to another function of the same set; based on this behavior the compiler will consider all the threaded functions as a single function in the call tree.

Moreover, the compiler will insert the ad-hoc fetch-decode sequence discussed in the previous subsections at the end of each Java bytecode function. As a consequence, all the functions called from each of the Java bytecode functions belonging to the threaded set have to be considered as called from the same single function, in turn called from the interpreter root function. To identify the latter, the compiler will expect a unique, special named or marked function that it will interpret as the interpreter entry point.

V. Results

In this Section we report the results regarding the different software interpreters that we took into consideration. We evaluated the classic interpreter based on while loop (CWI),

Table I
ROM MEMORY SIZE OF THE INTERPRETER FETCH AND DECODE PART

Interpreter	ROM Size [B]	
	w/o JPC Check	w/ JPC Check
CWI	241	283
PTCI	9812	17108
PTCHwD	4888	12784
PTCHwFD	376	376

the interpreter based on pseudo-threaded code (PTCI), the interpreter based on pseudo-threaded code with decode in hardware (PTCHwDI), and the interpreter based on pseudo-threaded code with both fetch and decode in hardware (PTCHwFDI).

At the same time, we evaluate the different hardware architectures where the interpreters run. These are the standard Oregano 8051 microcontroller (Std8051), the Oregano 8051 with the addition for the decode phase (DI8051), and the Oregano 8051 extended for the fetch and the decode phase (FDI8051). The first subsection briefly resumes the ROM size needed for the software layer; in the remaining sections we reported some performance analysis in terms of additional hardware, execution time, and power and energy estimation. The results are relative to the standard Oregano 8051 architecture implemented on a Virtex-5 FXT FPGA ML507 Evaluation Platform. For the software compilation we used the Keil μVision4 IDE tool chain.

A. ROM Size of the Software Layer

As seen in Section IV each version of the interpreter has a different implementation in the software layer. Table I resumes the ROM size of the software part performing the fetch and decode of the different interpreters. The two columns report the sizes in terms of bytes of the fetch and decode phase, but the second one includes also the check for the JPC minimum and maximum limits. We observe that the CWI and the PTCHwDFI have the lowest overhead in terms of code size, while the TCI has the highest overhead.

B. Additional Hardware

The architectures proposed in this paper were synthesized on FPGA, thus the estimation of the chip area was done in terms of FPGA utilization (used flip-flops (FFs) and used look-up tables (LUTs)). The two architectures (DI8051 and FDI8051) derived from the standard Oregano 8051 VHDL (Std8051) do not utilize a great amount of additional hardware as Table II shows; with a very rough estimation, the average additional utilization is about 3.2% for the DI8051 and about 7% for the FDI8051 compared to the Std8051. In fact, the modification consists of a few additional registers and little logic, with which we integrated the additional functionalities in the existing 8051 state machine.

Table II
FPGA UTILIZATION FOR THE DIFFERENT ARCHITECTURES

Architecture	FPGA Util.			
	FFs	Diff. %	LUTs	Diff. %
Std8051	582	-	2623	-
DI8051	597	2.6	2721	3.7
FDI8051	614	5.5	2885	10.0

Table III
RUN-TIME PERFORMANCE OF THE INTERPRETERS FOR THE FETCH AND DECODE PART

Interpreter	Run-time [Clk Cycles]			
	w/o JPC Check	Diff. %	w/ JPC Check	Diff. %
CWI	76	-	138	-
TCI	60	-21	122	-12
TCHwD	38	-50	100	-28
TCHwFD	6	-92	6	-96

C. Run-time Performance

The assessment of run-time performance was done by counting the clock cycles needed for the fetch-decode phase of the interpreter. The CWI and PTCI only make use of standard 8051 instructions; the difference between their run-time performances is due to the pseudo-threaded code technique present in PTCI as already discussed in Section III. PTCHwDI and PTCHwFDI are based on the non-standard 8051 instructions introduced in DI8051 and FDI8051, respectively. Table III shows a clear improvement in terms of clock cycles reduction for the PTCHwDI and the PTCHwFDI compared to the two interpreters running on the standard architecture. The improvement is due to the execution of the fetch and the decode functionalities directly in hardware instead of executing a sequence of many standard 8051 instructions. The fourth column of Table III is relative to the run-time performance for interpreters with the check on the JPC bounds. The trend of the improvement is similar to the second column where no check is done; but for the PTCHwFDI the improvement is impressive and due to the fact that the bounds check is done in hardware.

For the sake of a more complete investigation of the improvement of the run-time performance, we measured and averaged the execution times of a set of eight often used Java Card bytecode instructions (sconst_n, bspush, sspush, sstore_n, sload_n, sadd, ifeq, ifcmpeq). Figure 11 refers to the interpreter without bounds check and shows how the fetch and decode time reduction influences the overall Java Card bytecode interpretation time. The TCHwFD implementation presents an overall time reduction of 41% compared to the CWI implementation.

D. Power and Energy Analysis

We ran and tested our system on a Virtex 5 FPGA; hence, the only estimation that we made is related to this technology. To estimate the power consumption of the three

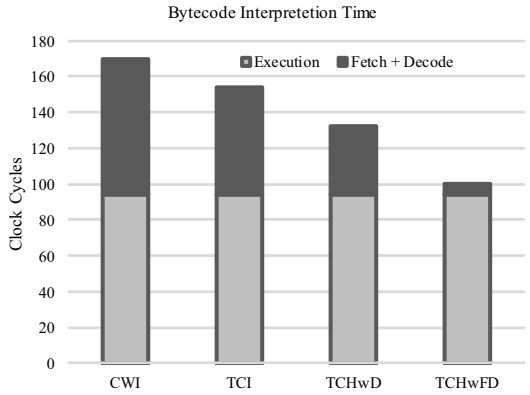

Figure 11. Run-time evaluation for the bytecode interpretation

Table IV
DYNAMIC POWER OF THE ARCHITECTURE CORE

Architecture	Dynamic Power [mW]			
	w/o JPC check	Diff. %	w/ JPC check	Diff. %
Std8051	25	-	25	-
DI8051	21	-16	22	-12
FDI8051	18	-28	18	-28

architectures in use, we based our analysis on the XPower Analyzer tool in the Xilinx design Suite [22]. For the sake of a more precise estimation of the dynamic power, the tool can use the switching activity data of the circuit nodes obtained during a testbench simulation. To get an even better result, we used the switching activity data extracted from the testbenches simulations of the fetch and decode phases of the interpreter. With this setting the power information is strictly correlated to the fetch-decode activity. From the XPower Analyzer we took the power relative to the switching activity (dynamic power) into consideration, that means without taking the leakage contribution into account. Looking at Table IV, we observe how the power consumption decreases from a maximum value in the Std8051 architecture to a minimum value in the FDI8051 architecture. The nature of the test-bench explains the power trend. In fact, in the Std8051, a wide set of various instructions are involved in the fetch and decode of the interpreter (and hence, a spread part of the circuits has switch activity); on the other hand, in the FDI8051, only the new instruction is involved (and hence, a focused part of the circuits has switch activity). We counter-check the results running the test-bench of the CWI on the FDI8051 architecture. The XPower Analyzer estimates the same power as for the Std8051 architecture. This demonstrates that also in the modified architecture, a set of various instructions has the same dynamic power as the STD8051 architecture. With the power analysis and the run-time analysis it is possible to calculate the dynamic energy needed for the fetch-decode phase with the well-

Table V
DYNAMIC ENERGY CONSUMPTION OF THE INTERPRETERS IN PERFORMING THE FETCH AND DECODE PHASES

Interpreter	Energy Cons. $[mJ/T_{ck}]$			
	w/o JPC Check	Diff. %	w/ JPC Check	Diff. %
CWI	1900	-	3450	-
PTCI	1500	-21	3050	-12
PTCHwD	798	-58	2200	-36
PTCHwFD	108	-94	108	-97

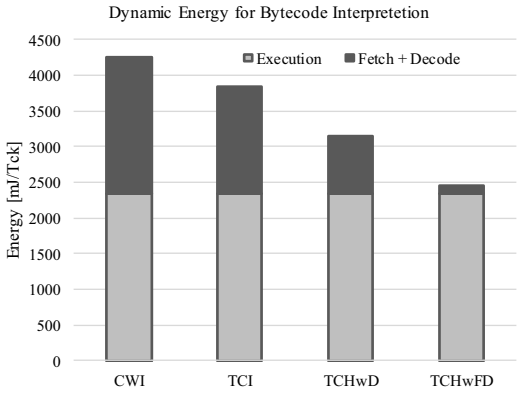

Figure 12. Dynamic energy consuption evaluation for the bytecode interpretation

known formula

$$E = \int p(t)dt = p_m \cdot \Delta t = p_m \cdot N \cdot T_{ck}$$

where $p(t)$ is the power at a point in the time, p_m is the average over the time of the power, Δt is the time interval taken into consideration, N is the number of clock periods of the time interval, and T_{ck} is the clock period. Table V reports the energy consumption for the fetch and decode phases in the different interpreters. It is expressed in mJ/T_{ck}, because the run-time of Table V is expressed in the number of clock cycles. The second column regards the basic interpreters, while the fourth regards interpreters performing also bounds check. The trend shows a consistent decrease of the energy consumption for all the interpreters compared to the CWI. This behavior is the result of the combination between a lower run-time and a lower power consumption.

As previously in the run-time assessment, we analyzed the energy needed for a complete Java bytecode interpretation (i.e. fetch, decode and execution). Figure 12 presents the energy consumption for the interpreter without bounds check. Considering the CWI and the TCHwFD, the overall energy consumption for the complete Java bytecode interpretation presents a reduction of 42%.

VI. CONCLUSIONS

In this paper, we proposed the modification of a standard architecture generally used in embedded systems for

enhancing the performance of the Java Card interpreter. We designed the interpreter with the pseudo-threaded code architecture, and we integrated in two different hardware architectures the decode phase, and both the fetch and decode phases of the interpreter, respectively. The interpreter with both fetch and decode performed in hardware has a reduction in the bytecode interpretation time of 41%, and a reduction in the energy consumption of 42%, compared to a classic interpreter implemented on a standard platform. The price for the improvement is a slight increase of the FPGA utilization (about 7%) and a negligible increase of the ROM size for the implementation of the software part.

Considering Java bytecode engineering techniques like dictionary compression promising , we see an opportunity for future work in the integration of such techniques with the Java Card interpreter based on the introduction of application specific functionalities into the microcontroller.

ACKNOWLEDGMENT

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank their project partner NXP Semiconductors Austria GmbH.

REFERENCES

[1] Oracle, *Java Card 3 Paltform. Runtime Environment Specification, Classic Edition. Version 3.0.4*. Oracle, September 2011. [Online]. Available: www.oracle.com

[2] Oracle, *Java Card 3 Paltform. Virtual Machine Specification, Classic Edition. Version 3.0.4*. Oracle, September 2011. [Online]. Available: www.oracle.com

[3] P. Klint, "Interpretation Techniques," *Software: Practice and Experience*, vol. 11, no. 9, pp. 963–973, 1981.

[4] J. R. Bell, "Threaded Code," *Communications of the ACM*, vol. 16, no. 6, 1973.

[5] I. Piumarta and F. Riccardi, "Optimizing Direct Threaded Code by Selective Inlining," *SIGPLAN Not.*, vol. 33, no. 5, pp. 291–300, May 1998.

[6] D. Gregg, M. A. Ertl, and A. Krall, "A Fast Java Interpreter," in *Proceedings of the Workshop on Java optimisaton strategies for embedded systems (JOSES), Genoa*. Citeseer, 2001.

[7] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling Java Just in Time," *Micro, IEEE*, vol. 17, no. 3, pp. 36–43, 1997.

[8] A. Krall and R. Grafl, "CACAO - A 64-bit JavaVM Just-in-Time Compiler," *Concurrency Practice and Experience*, vol. 9, no. 11, pp. 1017–1030, 1997.

[9] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler," *IBM Systems Journal*, vol. 39, no. 1, pp. 175–193, 2000.

[10] A. Azevedo, A. Kejariwal, A. Veidenbaum, and A. Nicolau, "High Performance Annotation-aware JVM for Java Cards," in *Proceedings of the 5th ACM international conference on Embedded software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 52–61.

[11] H. McGhan and M. O'Connor, "PicoJava: A Direct Execution Engine For Java Bytecode," *Computer*, vol. 31, no. 10, pp. 22–30, 1998.

[12] S. Steel, "Accelerating to Meet the Challenges of Embedded Java," *Whitepaper, ARM Limited*, 2001.

[13] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man, "Instruction Set Definition and Instruction Selection for asips," in *Proceedings of the 7th international symposium on High-level synthesis*. IEEE Computer Society Press, 1994, pp. 11–16.

[14] M. Arnold and H. Corporaal, "Designing domain-specific processors," in *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, ser. CODES '01. New York, NY, USA: ACM, 2001, pp. 61–66.

[15] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2008, pp. 209–220.

[16] A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet, "Checking the Paths to Identify Mutant Application on Embedded Systems," in *Future Generation Information Technology*, ser. Lecture Notes in Computer Science, T.-h. Kim, Y.-h. Lee, B.-H. Kang, and D. Slezak, Eds. Springer Berlin / Heidelberg, 2010, vol. 6485, pp. 459–468.

[17] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Trans. Archit. Code Optim.*, vol. 4, no. 4, pp. 2:1–2:36, Jan. 2008.

[18] Oregano Systems, "8051 Soft IP CORE," 2013. [Online]. Available: www.oreganosystems.at (Oregano Systems website)

[19] GNU GCC, "Compiler Collection," 2013.

[20] G. Bouffard and J.-L. Lanet, "The next smart card nightmare," in *Cryptography and Security: From Theory to Applications*, ser. Lecture Notes in Computer Science, D. Naccache, Ed. Springer Berlin Heidelberg, 2012, vol. 6805, pp. 405–424.

[21] M. Lackner, R. Berlach, M. Hraschan, R. Weiss, and C. Steger, "A defensive java card virtual machine to thwart fault attacks by microarchitectural support," in *Risks and Security of Internet and Systems (CRiSIS), 2013 International Conference on*, Oct 2013, pp. 1–8.

[22] Xilinx, "ISE Design Suite," 2013.

An Application Specific Processor for Enhancing Dictionary Compression in Java Card Environment

Massimiliano Zilli[1], Wolfgang Raschke[1], Johannes Loinig[2], Reinhold Weiss[1] and Christian Steger[1]
[1]*Institute for Technical Informatics, Graz University of Technology, Inffeldgasse 16/I, Graz, Austria*
[2]*Business Unit Identification, NXP Semiconductors Austria GmbH, Gratkorn, Austria*
{*massimiliano.zilli, wolfgang.raschke, rweiss, steger*}*@tugraz.at, johannes.loinig@npx.com*

Keywords: Smart Card, Java Card, Hardware-supported Interpreter, Compression

Abstract: Smart cards are low-end embedded systems used in the fields of telecommunications, banking and identification. Java Card is a reduced set of the Java standard designed for these systems. In a context of scarce resources such as smart cards, ROM size plays a very important role and dictionary compression techniques help in reducing program sizes as much as possible. At the same time, to overcome the intrinsic slow execution performance of a system based on interpretation it is possible to enhance the interpreter speed by means of specific hardware support. In this paper we apply the dictionary compression technique to a Java interpreter built on an application specific processor. Moreover, we move part of the decompression functionalities in hardware with the aim of speeding up the execution of a compressed application. We obtain a new interpreter that executes compressed code faster than a classic interpreter that executes non-compressed code.

1 INTRODUCTION

Entering a building with restricted access without a metal key, calling someone with a mobile phone, and paying at the supermarket without physical money are all activities based on the use of smart cards. With the increase of informatization in many sectors of society, these systems are destined to become even more widespread.

Smart cards are low-end embedded systems consisting of a 8/16 bit processor, some hundreds kilobytes of persistent memory and some kilobytes of RAM. The applications running on smart cards are often developed in C and in assembly to maximize execution time and minimize ROM size. Even following good programming practices, developing the applications in C and assembly has the problem of portability and demands a great amount of time for porting the applications from one platform to another. A programming language based on an interpreter like Java would resolve the problem of portability and also introduce security mechanisms included in the Java run-time environment.

A complete Java run-time environment requires hardware resources two orders of magnitude higher than the typical smart card hardware configuration. Java Card standard is a reduced set of the Java standard tailored for smart cards (Oracle, 2011a) (Oracle, 2011b). Moreover, with the "sandbox model", the Java Card run-time environment offers a secure and sound environment protecting against many types of security attacks. In contrast to standard Java environments, Java Card virtual machine is split in two parts: one off-card and the other on-card. The off-card Java Card Virtual Machine consists of the converter, the verifier, and the off-card installer. The converter transforms the *class* file into a CAP file, which is the shipment format of Java Card applications. The verifier checks the CAP file for legitimacy so that the code can be safely installed. After verification, the off-card installer establishes a communication channel with the on-card installer to transfer the content of the CAP file to the smart card. The on-card installer proceeds with the installation of the application, so that afterwards the application execution is possible.

In smart cards, where the persistent memory size is a first order issue, keeping the ROM size of the application as small as possible is a prominent issue. Compression techniques based on the dictionary mechanism fit very well in a software architecture based on a "token" interpreter like Java. The compression phase is performed off-card between the verification and the installation processes. The on-card Java Card virtual machine provides for the decompression during run-time. The drawback of this approach is the slow-down of the application execution

due to the decompression phase. As a consequence, time performance issues could prevent the adoption of the compression system especially in contexts where time constraints are strict.

The most popular technique for speeding-up the Java interpreter is the Just In Time (JIT) compilation. Unfortunately, this approach is not compliant with the typical smart card hardware configurations. A different approach based on an interpreter with hardware support is more suitable for smart cards.

In this paper we integrate the dictionary decompression functionality on an interpreter with hardware support. As final result, we obtain an interpreter able to execute compressed applications faster than a standard software interpreter executing a non-compressed application.

The structure of the rest of this paper is as follows. Section 2 reviews the previous work that forms the basis of this research. Section 3 analyzes the dictionary compression and its application to the interpreter with hardware support. In section 4 we evaluate the proposed models with particular attention to the execution time. Finally, in section 5 we report our conclusions and present suggestions for future work.

2 RELATED WORKS

Java Card is a Java subset specifically created for smart cards that allows developers to use an object-oriented programming language and to write applications hardware independently (Chen, 2000). The virtual machine in the run-time environment represents a common abstraction layer between the hardware platform and the application, and makes it possible to compile the application once and run it in each platform deploying a compliant Java Card environment. The issuing format of Java Card applications is the CAP file, whose inside is organized in components (Oracle, 2011b). All the methods of the classes are stored in the method component; consequently the latter is usually the component with the highest contribution to the overall application ROM size. For this reason, a reduction of the size of the method component would mean a significant reduction of the ROM space needed to install the application on the smart card.

Alongside following good programming practices, the main solution for reducing the ROM size of an application is the compressing of said application. The drawback of common compression techniques based on Huffmann and LZ77 algorithms is their need of a considerable amount of memory to decompress the application before its execution (Salomon, 2004).

Dictionary compression is a technique based on a dictionary containing the definitions of new symbols (macros) (Salomon, 2004). Each definition in the dictionary consists of a sequence of symbols that is often repeated in the data to compress. In the compression phase the repeated sequences are substituted with the respective macros (the macro definition and the substituted sequence have to be the same), while in the decompression phase the macros are substituted with their definition. Claussen et al. applied dictionary compression to Java for low-end embedded systems (Clausen et al., 2000). Applied to interpreted languages like Java, the main advantage of this compression technique, when compared to the traditional techniques, resides in the possibility to decompress the code on-the-fly during run-time. In fact, when the interpreter encounters a macro in the code, it starts the interpretation of the code in the macro definition, not needing the decompression of the entire code. Claussen et al. could save up to 15% of the application space, but this had also the disadvantage of a slower execution speed quantifiable between 5% and 30%. In (Zilli et al., 2013), we explored the extensions of the base dictionary technique. In that work, we evaluated the static and dynamic dictionary as well as the use of generalized macros with arguments.

The main disadvantage of interpreted languages compared with native applications is the low execution performance. The system commonly used to improve the execution speed in standard Java environments is the “Just In Time” (JIT) compilation (Suganuma et al., 2000) (Cramer et al., 1997) (Krall and Grafl, 1997). JIT compilation consists of the run-time compilation and optimization of sequences of bytecodes into native machine instructions. The disadvantage of this technique is the amount of RAM memory required to temporary store the compiled code. For low-end embedded systems such as smart cards, the amount of RAM memory needed for JIT compilations does not comply with the memory configurations.

Another solution for overcoming the low execution speed is the hardware implementation of the Java virtual machine. Previous works can be categorized into two main approaches: the direct bytecode execution in hardware, and the hardware translation from Java bytecode to machine instructions. An example of the first case is picoJava (McGhan and O’Connor, 1998), a Java processor that executes the bytecodes directly in hardware. This approach reaches high execution performance, but has the disadvantage of a difficult integration with applications written in native code. An example of hardware bytecode translation is the ARM Jazelle technology (Steel, 2001). In

this case the integration with native code is possible through an extension of the machine instruction set, but the performances are not as good as in the Java processor.

In the context of Java Card, we proposed a Java interpreter with hardware support (Zilli et al., 2014). In this work, we re-designed the interpreter as a "pseudo-threaded" interpreter where each bytecode provides for the jump to the next one. Moreover we moved the part of the Java interpreter responsible for the fetch and decode of the bytecode into the hardware. With this approach, we obtain a time reduction on the single bytecode execution of 40%.

The base of this research is the work in (Zilli et al., 2014). We extend the interpreter proposed for the handling of the dictionary decompression and present two solutions. One is in software, while, in the second, we implement part of the dictionary functionalities in hardware. For the evaluation of our work, we compare the two solutions with a software implementation on a standard hardware platform.

3 DESIGN AND IMPLEMENTATION

3.1 Dictionary Compression

In the context of Java Card, dictionary compression is an off-card process and consists of the substitution of repeated sequences of bytecodes with a macro whose definition is stored in a dictionary (Clausen et al., 2000) (Zilli et al., 2013). Given that 68 of the possible bytecode values are not defined by the standard, part of these can be used to extend the virtual machine instruction set and to represent the macros.

While the compression phase is performed in the off-card part of the Java Card virtual machine, the decompression phase is done on-card during the runtime. The decompression phase adapts well to the interpreter architecture, because every dictionary macro can be interpreted similarly to a "call" instruction. In Figure 1 we convey the structure of the dictionary, where two main components can be found. The first one consists of the look-up table containing the addresses of the macro definitions. The second component of the dictionary is the set of the macro definitions. The latter consists of sequences of Java bytecodes with a final Java bytecode being specific to the dictionary compression, whose mnemonic is `ret_macro`.

The realization of the decompression module in the Java Card virtual machine requires the implementation of the Java bytecode functions for the dictionary macro (`macro_jbc`) and for `ret_macro` (`ret_macro_jbc`). These two Java bytecodes are similar to the call and the return opcodes of a microcontroller, but they act on the program counter of the Java Card virtual machine (*JPC*). Figure 2 shows pseudo-assembly code for the implementation of the two bytecodes on a standard architecture. For the

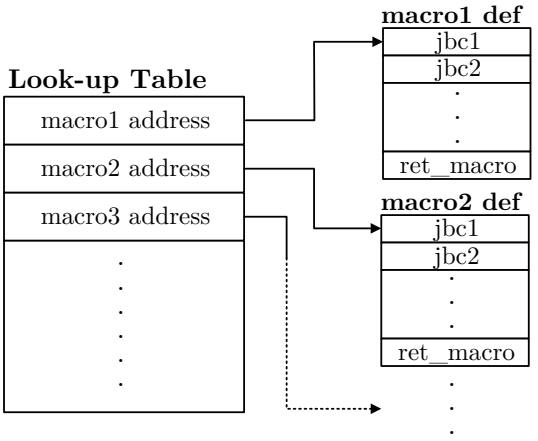

Figure 1: Organization of a dictionary

```
macro_jbc:
  MOV A, JPC
  MOV JPC_RET, A
  MOV A, #LOOKUP_TABLE
  ADD A, JBC
  MOV DPTR, A
  MOV A, @DPTR
  MOV JPC, A
  RET

ret_macro_jbc:
  MOV A, JPC_RET
  MOV JPC, A
  RET
```

Figure 2: Macro function and `ret_macro` function in pseudo-assembly code for the standard architecture

sake of easier readability, the pseudo-code does not take the real address width into consideration. In the first part of the `macro_jbc` function, the actual *JPC* is stored into a "return" variable. Afterwards, the actual Java bytecode (e.g. the macro value) is used to calculate the offset in the look-up table from which the address of the corresponding macro definition is fetched. At this point, the JPC is loaded with the address of the macro definition. From this point on, the Java virtual machine interprets the bytecodes contained in the macro until it encounters the

ret_macro Java bytecode. Figure 2 shows the implementation of the ret_macro bytecode in pseudo-assembly code. As can be seen, the Java program counter value contained in the "return" variable is restored within the function. After the interpretation of the ret_macro bytecode, the Java Card virtual machine continues with the execution of the bytecodes following the macro instruction.

3.2 Decompression with the Hardware Supported Interpreter

In a context such as that of smart cards where the resources are limited, a solution to speed up the virtual machine is represented by an interpreter that uses hardware extensions specific for the Java Card interpretation (Zilli et al., 2014). Figure 3 explains, by means of a state machine, how the fetch and the decode phase of the interpretation are performed in hardware.

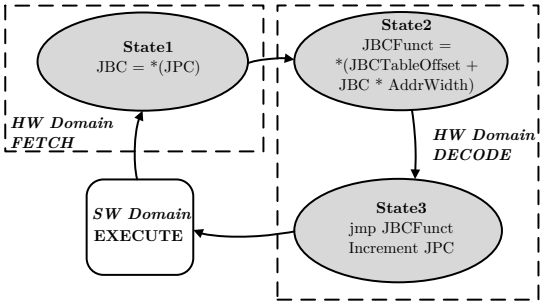

Figure 3: State machine of the intepreter with the fetch and the decode phase performed in hardware

To do this, the authors added the JPC register to the hardware architecture of the microcontroller, and extended the instruction set of the latter to enable access to the new functionalities. Moreover, they modified the interpreter from a classical "token" model to a "pseudo-threaded" model where every Java bytecode function has at its end the instructions to fetch and to decode the next bytecode.

We implement the dictionary decompression functionality for this enhanced architecture. In Figure 4 we show the pseudo-code for the implementation of the Java bytecode functions macro_jbc and ret_macro_jbc. In this implementation the dictionary decompression is completely performed in software and is analogous to the implementation for the normal interpreter, except for the use of the extended functionalities for manipulating the JPC that is now an internal register of the hardware architecture. The macro implementation is the same as the standard case except for the fact that we use the new machine

```
macro_jbc:
  GET_JPC_IN_A
  MOV JPC_RET, A
  MOV A, #LOOKUP_TABLE
  ADD A, JBC
  MOV DPTR, A
  MOV A, @DPTR
  SET_JPC_FROM_A
  GOTONEXTJBCFUNCT

ret_macro_jbc:
  MOV A, JPC_RET
  SET_JPC_FROM_A
  GOTONEXTJBCFUNCT
```

Figure 4: Macro function and ret_macro function in pseudo-assembly code for the interpreter with the fetch and the decode phase in hardware

instructions and that, at the end of the function, instead of a normal RET instruction, we have the activation of the hardware fetch and decode of the next Java bytecode. The same considerations can be made for the ret_macro bytecode.

3.3 Hardware Extension for Dictionary Decompression

The architecture extension of the hardware aided interpreter represents the core of this work. In a hardware/software co-design context, we moved parts of the software implementation of the dictionary decompression to hardware. Referring to Figure 1, it is necessary for the hardware architecture to have access to the dictionary look-up table base address and to the actual value of the macro. Moreover, we added a register for storing the return address of the Java program counter (*JPC*) and an internal register to temporary store the value of the processor program counter (*PC*).

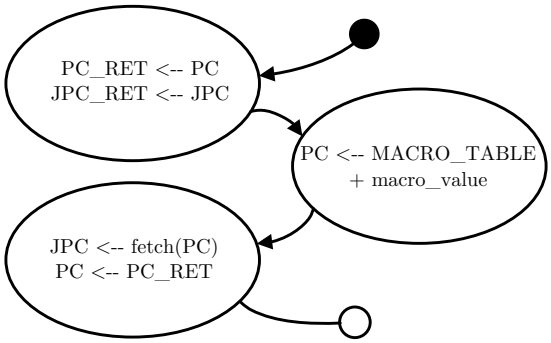

Figure 5: State machine of the PRECALL_MACRO machine opcode

Figure 5 sketches the finite state machine taking charge of the macro decodification. In the first step, the PC and the JPC are stored in the respective "return" register. In the second step, the *PC* is loaded with the value resulting from the sum of the macro look-up table base address (*MACRO_TABLE*) and the offset relative to the actual macro being decoded (*macro_value*). At this point, the value fetched from the ROM at the address pointed by the PC corresponds to the address of the macro to be executed. The fetched value is then stored into the JPC and the PC is restored.

The hardware functionality just described is activated by means of an additional machine instruction (whose mnemonic is *PRECALL_MACRO*) part of the extended instruction set. In Figure 6 we report the implementation of the `macro_jbc` and *ret_macro_jbc* bytecode functions. After the modification of the JPC, the `GOTONEXTJBCFUNCT` instruction starts the execution of the bytecodes of the macro definition. When the interpreter executes the `ret_macro` bytecode, the `REST_DICT_JBC` machine instruction is executed to restore the *JPC* value previously stored into the internal return register within the `PRECALL_MACRO` machine instruction. Afterwards, the instruction `GOTONEXTJBCFUNCT` continues the execution flow from the first bytecode after the macro bytecode.

```
macro_jbc:
  MOV A, #LOOKUP_TABLE
  SET_DICTLOOKUP_TABLE
  MOV A, JBC
  PRECALL_MACRO
  GOTONEXTJBCFUNCT

ret_macro_jbc:
  REST_DICT_JBC
  GOTONEXTJBCFUNCT
```

Figure 6: `macro_jbc` and `ret_macro_jbc` functions in pseudo-assembly code for architecture with hardware support for dictionary decompression

4 RESULTS AND DISCUSSION

The results of this work can be subdivided into two parts: one related to the compression phase and the other to the decompression phase. For the first part we consider the space savings that we obtained by applying the dictionary compression to a set of industrial applications. For the decompression we analyze the run-time improvements due to the use of the new microcontroller architecture.

4.1 Compression

The main aspect regarding the compression phase is the space savings that can be obtained. We did not evaluate the compression speed performance because it is performed off-card, hence in a platform with no particular hardware constrains. For the assessment of the space savings, we applied the dictionary compression method to a set of three banking applications (XPay, MChip and MChip Advanced).

Table 1: Space savings obtained with the dictionary compression

Application	Size [B]	Space Savings [%]
XPay	1784	12.2
MChip	23305	9.2
MChip Advanced	38255	10.5

Table 1 summarizes the space savings obtained for the three applications. The second and third columns show the sizes of the method component and the space savings over the method component expressed in percentage. The reported space savings also account for the ROM space needed for the storage of the dictionary.

4.2 Decompression

To evaluate the decompression, we implemented the proposed architectures before building the relative interpreters onto them . The 8051 architecture is a low-end microcontroller consistent with hardware configurations of typical smart cards. As a starting point we took the 8051 implementation provided by Oregano and we added to it the extensions described in Section 3. We created two different architectures: one has the fetch and decode phase of the interpreter realized in hardware (FDI8051); the other, in addition to the fetch and decode of the interpreter, also has the hardware extension for the dictionary decompression (FDI8051DEC).

4.2.1 Additional Hardware

We synthesized the proposed hardware architectures on a Virtex-5 FPGA (FXT FPGA ML507 Evaluation Platform). In this way we are able to quantify the FPGA usage in terms of flip-flops (FFS) and look-up tables (LUTs). Table 2 shows the necessary hardware for the three available architectures. Both architectures have an increment of the FPGA usage with an higher increment for the FDI8951DEC. The higher

Table 2: FPGA utilization for the different architectures

Architecture	FPGA Util.			
	FFs	Diff. %	LUTs	Diff. %
Std8051	582	-	2623	-
FDI8051	614	5.5	2885	10.0
FDI8051DEC	666	14.4	2946	12.3

FPGA usage in the FDI8051DEC is due to the introduction of the decompression mechanism in hardware.

4.2.2 Execution Speed-up

For the evaluation of the run-time performance we proceeded with the evaluation of the execution of a dictionary macro. For the sake of this assessment we proceeded in two steps. The first step consists of the execution time measurement of a sequence of bytecodes running on a completely software-based interpreter (on the Std8051 architecture) and on the interpreter running on the architecture with the fetch and the decode phase of the interpretation in hardware (FDI8051 architecture). In the second step we evaluated the increment of the execution time due to the encapsulation of the sequence under test into a macro definition.

The analysis coming from the off-card compression shows that macros have an average length of three bytecodes. In a second instance, we measured and averaged the interpretation time of a set of frequently used bytecode instructions (`sconst_n`, `bspush`, `sspush`, `sstore_n`, `sload_n`, `sadd`, `ifeq`, `ifcmpeq`) to calculate the "average bytecode interpretation time". For the assessment, we took the implementation of the bytecodes from the Java Card reference implementation provided by Oracle. The measurement of the interpretation time was performed on two architectures: one with the interpreter completely in software (Std8051), and the other with the fetch and the decode phase of the interpreter performed in hardware (FDI8051).

Table 3: Execution time of a bytecode sequence

Architecture	Exec. Time [Clk Cycles]	Diff [%]
Standard 8051	680	-
FDI8051	400	-41%

At this point it is possible to evaluate the time needed for the execution of a sequence composed of general bytecodes with a length equal to the average length of a macro definition. Table 3 lists the execution time for the two architectures.

To assess the influence on the execution time due to the sequence being encapsulated into a macro definition, we took into consideration the same "average" sequence previously examined. In this case we completed the evaluation using the three hardware architectures available, because each of them displays different behavior during the execution of a macro. The

Figure 7: Execuiton time of a macro on three architectures

graphic of Figure 7 summarizes the results regarding the interpretation of the macro. The execution time (vertical coordinate) is expressed in clock cycles. In each bar the light-gray part represents the time needed for the interpretation of the sequence. The dark gray part accounts for the overhead due to the macro encapsulation, which means the interpretation of the macro bytecode and the `ret_macro` bytecode. The entire bar represents the time needed to execute a macro. At the top-right corner of each bar there is a percentage number representing the macro overhead compared to the overall macro execution.

4.3 Discussion

Dictionary compression allows space savings of about 10% of the applications ROM footprint. With a standard architecture, the execution overhead due to the macro encapsulation would take 20% of the overall macro interpretation (Figure 7).

The architecture with the fetch and the decode phase of the interpreter in hardware (FDI8051) provides a significant acceleration of the macro execution because of the increase in speed of the single bytecode execution. Compared to the macro execution on the standard 8051 architecture, the FDI8051 architecture permits a decrease in the execution time of 45%.

The new architecture with the support for the dictionary compression (FDI8051DEC) further improves the execution performance, reducing the time overhead owed to the macro encapsulation of the se-

quence. Compared to the FDI8051 architecture, the overhead time is reduced by about 50%. Compared to the standard 8051 architecture, the FDI8051DEC architecture allows a speed-up of 2 in the execution of a dictionary macro.

5 CONCLUSIONS

In this paper we combined the dictionary compression technique with a Java Card interpreter based on an application specific processor. Moreover, we moved part of the decompression functionalities to the hardware architecture, further extending the application specific processor. The result of this design is a new Java Card interpreter able to execute compressed code twice as fast as a standard interpreter. Although the new hardware architecture needs a little additional hardware, the compressed Java Card applications need about 10% less memory footprint than the non-compressed ones.

Beyond plain dictionary compression, dictionary compression techniques that make use of general macros definitions with arguments are available. The integration of these dictionary compression techniques in the application specific processor provides opportunities for future research.

ACKNOWLEDGEMENTS

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank their project partner NXP Semiconductors Austria GmbH.

REFERENCES

Chen, Z. (2000). *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Professional.

Clausen, L. R., Schultz, U. P., Consel, C., and Muller, G. (2000). Java Bytecode Compression for low-end Embedded Systems. *ACM Trans. Program. Lang. Syst.*, 22(3):471–489.

Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., and Wolczko, M. (1997). Compiling Java Just in Time. *Micro, IEEE*, 17(3):36–43.

Krall, A. and Grafl, R. (1997). CACAO - A 64-bit JavaVM Just-in-Time Compiler. *Concurrency Practice and Experience*, 9(11):1017–1030.

McGhan, H. and O'Connor, M. (1998). PicoJava: A Direct Execution Engine For Java Bytecode. *Computer*, 31(10):22–30.

Oracle (2011a). *Java Card 3 Paltform. Runtime Environment Specification, Classic Edition. Version 3.0.4*. Oracle.

Oracle (2011b). *Java Card 3 Paltform. Virtual Machine Specification, Classic Edition. Version 3.0.4*. Oracle.

Salomon, D. (2004). *Data Compression: The Complete Reference*. Springer-Verlag New York Incorporated.

Steel, S. (2001). Accelerating to Meet the Challenges of Embedded Java. *Whitepaper, ARM Limited*.

Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., and Nakatani, T. (2000). Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193.

Zilli, M., Raschke, W., Loinig, J., Weiss, R., and Steger, C. (2013). On the dictionary compression for java card environment. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, M-SCOPES '13, pages 68–76, New York, NY, USA. ACM.

Zilli, M., Raschke, W., Loinig, J., Weiss, R., and Steger, C. (2014). A high performance java card virtual machine interpreter based on an application specific instruction-set processor. In *Digital System Design (DSD), 2014 Euromicro Conference on*. in print.

Embedding Research in the Industrial Field: A Case of a Transition to a Software Product Line

Wolfgang Raschke
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
wolfgang.raschke@tugraz.at

Massimiliano Zilli
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
massimiliano.zilli@tugraz.at

Johannes Loinig
NXP Semiconductors Austria GmbH
Gratkorn, Austria
johannes.loinig@nxp.com

Reinhold Weiss
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
rweiss@tugraz.at

Christian Steger
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
steger@tugraz.at

Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
christian.kreiner@tugraz.at

ABSTRACT

Java Cards [4][5] are small resource-constrained embedded systems that have to fulfill rigorous security requirements. Multiple application scenarios demand diverse product performance profiles which are targeted towards markets such as banking applications and mobile applications. In order to tailor the products to the customer's needs we implemented a Software Product Line (SPL). This paper reports on the industrial case of an adoption to a SPL during the development of a highly-secure software system. In order to provide a scientific method which allows the description of research in the field, we apply Action Research (AR). The rationale of AR is to foster the transition of knowledge from a mature research field to practical problems encountered in the daily routine. Thus, AR is capable of providing insights which might be overlooked in a traditional research approach. In this paper we follow the iterative AR process, and report on the successful transfer of knowledge from a research project to a real industrial application.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.13 [**Reusable Software**]: Domain Engineering; K.6.1 [**Project and People Management**]: Systems analysis and design; K.6.1 [**Project and People Management**]: Systems development

WISE'14, September 16, 2014, Vasteras, Sweden.

ACM 978-1-4503-3045-9/14/09 ...$15.00.
http://dx.doi.org/10.1145/2647648.2647649 .

Keywords

Knowledge Transfer, Action Research, Software Reuse

1. INTRODUCTION

As the field of Software Product Line Engineering [3][6] is now maturing, it is of growing importance to provide industrial cases. In the industrial context, many challenges exist in the establishment of a SPL which are not obvious to academia. Nevertheless, it is important to provide such an experience to an audience who is interested in establishing a SPL but has no prior real-life experience. For such an audience, it is beneficial to draw on a catalog of documented experiences. In this paper we strive to provide such a catalog.

In a matured research field, when the scientific results are about to be transfered to real applications, it is important to leave the proverbial research lab and apply research in the field. The classical scientific model requests to first state a problem, solve it, then evaluate the solution. In an industrial context, the evaluation of a process improvement is often not bound to a single and consistent problem. Rather, an evaluation is more oriented towards improving an existing solution. Action Research (AR) is an established research method which can be applied in such a context [2][7]. In order to assess the improvement a specific action has to a solution, this research method is cyclic and iterative. At the end of each iteration, the outcome and experiences are digested into lessons learned. In addition we describe how AR helps to elicit field experience in a knowledge transfer project. We show that the applied method can provide different insights to the traditional research approach.

2. RESEARCH METHOD

The more a science field grows and matures, the more it loses relevance for practitioners. This difficulty has been ob-

served for different fields [7]. Action research is an approach that bridges a highly developed research field and industrial practice. Thusly, as action researchers we are interested in the way a working system was established and how problems have been resolved. The research is accomplished en route [7]:

> "objectives, the problem, and the method of the research must be generated from the process itself, and that the consequences of selected actions cannot be fully known ahead of time".

We are interested in the process of the transition itself, and thus the iterative AR process is beneficial.

2.1 The Process of Action Research

As can be seen in Fig. 1, the AR process is iterative. Each iteration incorporates the following process steps [7]: diagnosing, action planning, action taking, evaluating and specifying learning.

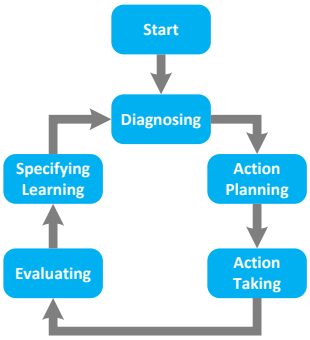

Figure 1: The Action Research process [7] is iterative and encompasses the following steps: *Diagnosing*, *Action Planning*, *Action Taking*, *Evaluating* and *Specifying Learning*. Each iteration starts with the *Diagnosing* process step.

- **Diagnosing:** An iteration starts with a problem. The problem is further elaborated and described. It states the requirements (research question) for the current iteration.
- **Action Planning:** In this phase the possible solutions for the diagnosed problem are investigated. If there are alternative solutions, they need to be compared with each other. In the end, a specific action has to be selected.
- **Action Taking:** Action needs to be taken. In the scope of this report this an increment of a prototype towards a running solution.
- **Evaluation:** The actions are evaluated against the diagnosed problems.
- **Specifying Learning:** Comparing the actions and the evaluation leads to lessons learned (LL). Moreover, reflecting on the lessons learned leads to newly diagnosed industrial problems and research questions.

3. FIRST ITERATION: DESIGN

The first iteration was accomplished mainly by the research partner. The industrial partner was inolved via interviews. In this phase we strived to create an initial big picture of a Software Product Line approach applied to the existing software system.

3.1 Diagnosing

In the first iteration the diagnosing step is a general elicitation of requirements for a Software Product Line. The initial adoption of a SPL is expensive because appropriate tools are needed. Moreover, there is a large amount of effort required for refactoring. Thusly, a decision needs to be taken whether the SPL shall be implemented on an industrial scale or not. For this reason, we started with a research project with the intention of gradually increasing the industrial participation. We can then also regard each iteration of the AR cycle as an evaluation if the transition to a SPL shall be continued. Basically, the problem set by the first iteration is: The SPL approach is usually not known to everyone who is involved in the software development. Before starting a big implementation (which is expensive) the responsible people need to be convinced that the approach works. Moreover, there is no way to alter the software system to the needs of an SPL. It has to be minimally invasive in the sense that the existing software system shall continue working as it is without much refactoring.

3.2 Action Planning

For an initial action plan we listed the most problematic pain points in the design and configuration process:

- **Requirements consistency:** There are many interrelations between requirements (product features and security features). Requirements may demand the inclusion or exclusion of other requirements. Due to the high number of requirements and corresponding dependencies, it is hard to maintain the consistency across the selected requirements of a certain product.
- **Mapping features to source code:** Features and source code are two distinct worlds with different roles involved: product managers, security engineers, test engineers and developers. It is important to bridge these two worlds with a mapping between the high-level features to existing source code and tests.
- **Build configuration:** The build configuration shall be consistent with the selected product and security features. This is only possible, if there is an automatic derivation of build information from the features mentioned.
- **Test selection:** The test selection shall be consistent with the product and security features. Again, the test selection shall be derived automatically from the high-level information.

For providing a feasibility study, we sought to rapidly create a first prototype. The design of this prototype is given in Fig. 2. The design states that the inputs for a configuration are the product and security features. Based on their selection, the appropriate set of components is calculated. Since unit tests and integration tests are bound to components, the component set also determines the set of these tests.

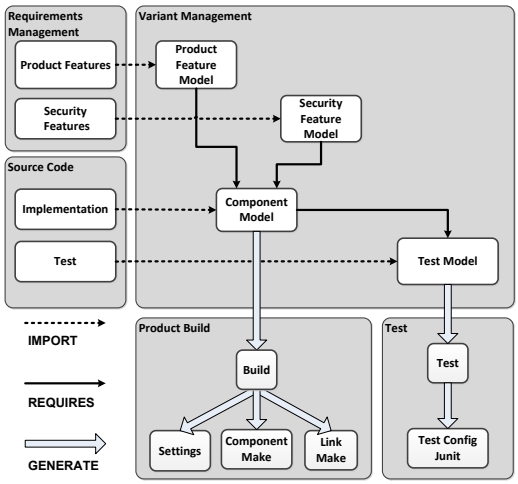

Figure 2: Variability models are imported from a requirements management system and from the source code. The user selects a set of product and security features. The resulting configuration is calculated. As a result, a test and a product build configuration is generated.

3.3 Action Taking

As a proof of concept, we implemented a first prototype of the design. The first prototype is not an appropriate implementation of the industrial problem. The industrial implementation would take lots of expert knowledge for the exact mapping of: features to components, and features to system tests (which has been omitted in the prototype).

3.4 Evaluation

We have modeled four main artifacts: product features, security features [1], components and tests. Thus, it is necessary to demonstrate the approach involving all of them. Moreover, all pain points shall be covered by the evaluation. We designed a use case for this purpose: First, select a specific feature and generate the build and test configuration. Demonstrate that the test passes in this case. Second, deselect a specific feature and generate build and test configuration. Demonstrate that the test is not executed. All the other previously executed tests should deliver the same results as before. Although the use case seems trivial at a first glance, it is in fact a real improvement in an industrial setting and has made considerable impression.

3.5 Specifying Learning

LL1: Don't start with a big implementation. It is not feasible to start with a big implementation for several reasons: First, the SPL approach is usually not commonly known within a company. Before starting a big implementation (which is expensive) the responsible people need to be convinced of the feasibility of the approach. This is one of the most important issues, because a transition of real-world source code is extremely expensive. Moreover, a first prototype can be demonstrated and encourages reflection processes within and in-between individuals and influences the requirements for the SPL. In the following, we provide a short example: Variant management in an industrial context is usually associated with source code configuration. People are not aware that more artifacts come into play, such as documentation and tests. After the demonstration of the prototype, people regarded the test selection and configuration as the most valuable asset of such an SPL.

LL2: It is not apparent where the product features come from. The first apparent source of features are high-level requirements in requirements management systems and tables from product management. Nevertheless, the number of features from such sources are in the range of several hundred. A consistent mapping of so many features to source code artifacts would require too much effort from domain experts. It is hard to justify so much effort for building a prototype solution. Although, it makes sense to address these points later in the transition, there needs to be a smaller feature set.

LL3: We learned also, that the mining of component dependencies is not facile. For the first prototype, we retrieved these dependencies from *include* statements. However, this approach is just an approximation: includes may mask other inclusions of dependencies. Such nested *ifdef* constructs occurred frequently but could be avoided with adhere coding guidelines and appropriate refactoring.

LL4: One of the most surprising lessons learned was that the complexity of the Software Product Line was underestimated. We, as research partners did underestimate the size of the software, the complexity of the configuration and the dynamics of an industrial project. Moreover, the complexity of the problems that come with a SPL were underestimated. This is due to the reason that some mechanisms for variant management were still implemented in the software project. These mechanisms worked fine but with the rising number of configuration switches they grew too complex.

4. SECOND ITERATION: BUILD

The second iteration was accomplished by the research partner and the industrial partner. As a research partner we participated in the daily routine of the industrial software project. Together, we sought to create an applicable solution for a sophisticated build management.

4.1 Diagnosing

The purpose of the second iteration was the improvement of the existing build system for variant management. At the time of diagnosing, variant management was still accomplished to a certain degree. The problems with the existing approach were the following: First, nested *ifdef* constructs were becoming increasingly unreadable. There were no concise coding guidelines for such constructs. Second, the granularity was not consistently defined: sometimes makefiles were split into more files and sometimes not. There was no evident splitting criterion. Third, configuration switches were spread over the code: they exist in makefiles, scripts, xml files, source code and the like.

4.2 Action Planning

The rationale of this iteration is to show that a considerable number of features can be managed systematically. However, the outcome of this iteration shall not only be a feasibility study, but also the development of appropriate tools, guidelines and practices. It is not possible to evaluate these improvements in an active project because the risk of

interruption and the resulting cost is too high. Thus, we decided to implement the build iteration in a small team.

4.3 Action Taking

We started with refactoring the existing makefiles and writing a proposal for the respective makefile coding guidelines. The refactoring of the makefiles was only possible to a very limited degree: each change of a makefile had to be tested for several product configurations. This diligent testing was necessary, because the refactoring was considered risky for the product development. Before the refactored makefile was tested fully, the makefile had been changed by the industrial team and needed adjustments again. Summarizing, a makefile refactoring is only feasible, if the entire team focuses on this objective.
The definition of a common vocabulary was a major issue. A variant management existed beforehand with different vocabulary to that used in academia. This was the source of several misconceptions. Another issue was the development of a naming scheme for compiler flags and constant definitions.
It was important to discuss the configuration of the high-level switches (features). The existing set of several hundred product features was considered as too complex to be linked to source code artifacts. This is not a technical limitation. It is a limitation of complexity. In an industrial context it is nearly impossible to map several hundred features to source code, because the required domain expertise is rare. Thusly, we decided to limit the number of product features to 30. Nevertheless, handling such a number of high-level switches in a complex industrial project is a challenge and deemed appropriate as a starting point.
In order to connect the existing makefiles with the software product line tooling, it was necessary to find the existing switches within makefiles and source code files. During this examination we found that there are several high-level switches available in the makefiles.We planned to use them for the first product feature set because their number did not exceed the mentioned 30. With these rare existing product features it was feasible to build another increment of the variant management prototype and to demonstrate the feasibility of our approach.

4.4 Evaluation

The most relevant achievement was the agreement on the concept of mapping features, makefiles and compiler flags. We decided to start with up to 30 features in order to keep complexity low. In fact, controlling up to 30 features in an industrial environment is a challenge: the knowledge of the mapping between features and source code artifacts is not apparent. It is spread over many developers and domain experts whose time is limited. For this purpose we first mined this knowledge from the source code and makefiles. We have shown the feasibility of the approach when the number of features is up to 30. We managed to agree on a makefile concept that satisfies all parties such as developers, configuration managers and the like. We could elicit a set of coding guidelines for makefiles. Unfortunately, the makefiles could not be refactored in the active project, because of the frequently altering dependencies.
We agreed on a concept of storing all features in a string. This string is then parsed in the makefiles and conditions are evaluated according to the feature set in the string. The makefiles are then responsible for further configuration of the source code.

4.5 Specifying Learning

LL5: Configuration switches were scattered across many different locations. A goal of this transition is to centralize the logic of the configuration. As a rule of thumb, avoiding nested ifdefs is a good coding practice because this is an indicator of there being no hidden configuration knowledge.
LL6: Regarding the number of product features there is a difference between technical feasibility and the actual feasibilty (which is limited by complexity and effort). At the beginning of a SPL establishment it is necessary to demonstrate the feasibility with a reduced feature set.
LL7: The sources of product features are often not so apparent. Initially, we mined them in tables from product management and in Requirement Management Systems (RMS). Later we found, that makefiles and source code are a better starting point for retrieving features.
LL8: Building up a common vocabulary is difficult. Because there were many involved parties (industry, tool vendor, academia), many different phrases were used to denote the same meaning. Since this vocabulary was not consistent, it lead to many misunderstandings. It took a long time and many discussions to build up a vocabulary that was consequently used by all involved parties.

5. THIRD ITERATION: INDUSTRIAL DEVELOPMENT

Before the start of the third iteration much knowledge was transfered to the industrial partner who was then excellently prepared for establishing a SPL. Moreover, a tool vendor was involved in order to tailor the existing tool to the specific needs.

5.1 Diagnosing

The main issues are: build and test configuration on an industrial scale. The approach must be minimally invasive: the tooling must create configuration artifacts which can be used without the tool. This is due to the fact that many software developers work on the project: First, not everyone shall be able to change the variability model. We agreed that it is better to a few well trained developers working on the model and generating configuration artifacts with the tool. These configuration artifacts are then submitted to the version management system. Second, the transition to a product line with dedicated tooling is costly in terms of software licenses and resources. Thus, a possible termination of the transition shall not result in a system which no longer works.
As mentioned above, the makefiles will be controlled by configuration artifacts. As an addition, it is necessary to refactor the makefiles. Such a refactoring has failed in the previous iteration. Therefore, co-ordination needs to be planned more effectively.
Again,we want to address the problem of test configuration. There are actually two dimensions of test case selection: First, tests are selected regarding their functionality, which means that the tests shall cover the functionality defined by the product feature set. The respective tests are unit tests, integration tests and system tests. Second, the tests are also

selected regarding the degree of coverage: smoke tests are very few tests which shall detect possible errors. The full test set strives to cover each line of code. Moreover, the test selection also depends on the test platform: simulator, emulator, real hardware.
Currently, we utilize a test selection mechanism, based on JUnit[1] *pre-conditions*. However, these pre-conditions shall be removed and deployed to the variant model.
At present, there is no single point of test selection. In the underlying industrial context, this is the major argument for the introduction of a variant management tool.

5.2 Action Planning

Initially, there was a phase of requirements engineering and many interactions between the industrial partner and the tool vendor. This phase was intended to gather information and demands regarding the SPL project.
We decided to first start with the build configuration and proceed with the configuration of unit and integration tests. Afterwards, we intend to manage the selection of system tests. For each of these objectives, we defined three different phases: a development phase, a pilot phase and a transfer phase. In the development phase, the software and the tool are developed in order to meet the requirements. It is not used in the active software project. In the pilot phase, the finished solution is evaluated in order to find its flaws and possible corrections. If the solution is regarded as mature enough, it passes to the transfer phase. In the transfer phase the solution is used in the industrial project and developers are trained to apply it.

5.3 Action Taking

The new variant model is similar to the model of the first iteration (see Fig. 2). It contains product features, components and tests. It has been taken over by the industrial partner, for the bigger part. Also, the mapping of unit tests and integration tests to components is similar to the first prototype.
System tests cause the test selection to be problematic: several thousand of them cannot be mapped to components. Thus, they need to be mapped to features. This can only be done by experienced domain experts whose time is limited. Thus, we still strive to find the dependencies in a different way.

5.4 Evaluation

The industrial project was successful, so far. With the limited feature set, the mapping between features and source code is feasible. At the moment, the pilot phase of the solution seems to work as desired. What is missing is experience from the broad transfer of the solution to the daily routine. academic partner's previous work could be reused in the industrial setup. So, it can be concluded that the knowledge transfer was successful to a considerable degree.

5.5 Specifying Learning

LL9: What we especially learned is that such a transition to a SPL can only be successful, if there is one *insider* responsible for it. It is usually much easier for such an insider to get information and support. This is especially the case, when an academic project starts to become industrial.

[1]http://junit.org/

LL10: A very important point is the following: such a project must deliver some early successes to get enough support. For this reason and to minimize risk, the industrial partner fostered a staged transition to a SPL. After each stage the solution is evaluated with the possibility of the project being canceled.

6. CONCLUSION

In Table 1 the lessons learned are summarized. As can be seen we gathered 10 lessons learned in 3 iterations. In the third iteration we could only identify 2 lessons learned: this is due to the fact that this iteration is not yet fully completed. We rated the experience of each lesson learned regarding the degree of technical and field experience. The more the experience is field-related the less it can be reproduced in a traditional research lab setting. Field-related experience may have a technical facet but usually involves more aspects. The relation between pure technical experience and field experience shows to which degree action research can enhance traditional research methods. We rated each dimension on a scale between zero and three plus: No related experience results in a zero rating. A rating of three plus denotes a highly related experience.

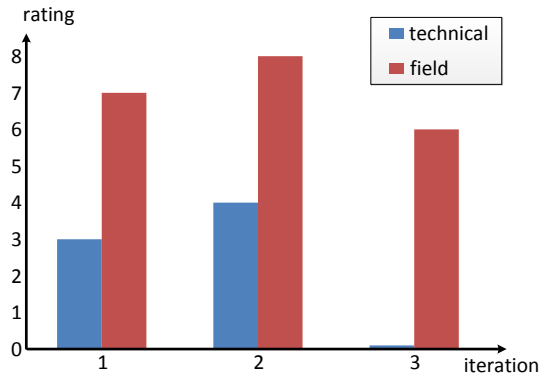

Figure 3: The rated technical and field experience of the lessons learned for all three iterations.

Fig. 3 shows the evolution of the learning experience over the three iterations. In the first iteration, the focus of the SPL transition was on the research partner. Following the rated experience is lower than in the second iteration. There, the research partner and the industrial partner were closely working together on a solution. In the third iteration the rated experience is lower than in the previous iterations. That is because this iteration is not yet fully completed. It is interesting to see that the field experience is always rated higher than the technical experience. This is due to the fact that we were participating in a large and dynamic software project. In such a setting, the research conditions are different to an isolated research project. However, the research method also helps to gather insights which are not possible to elicit with a traditional research method. Such a method would only reveal the same technical experience, in the best case.

Table 1: Summary of all lessons learned.

Lesson Learned	Description	Iteration	Technical Experience	Field Experience
LL 1	don't start big	1		+++
LL 2	unclear source of features	1	+	+
LL 3	mining component dependencies	1	++	+
LL 4	underestimated complexity	1		++
LL 5	avoid nested *ifdefs*	2	++	+
LL 6	limit number of features	2		+++
LL 7	retrieve features from source	2	++	+
LL 8	build up a common vocabulary	2		+++
LL 9	insider responsibility	3		+++
LL 10	early successes	3		+++

In this paper we reported on a successful knowledge transfer from academia to industry. We described this transfer with an established research method which fosters the transfer of science to real and practical applications. This research method has enabled the extraction of several lessons learned which are likely to appear in a similar setting. The lessons learned have demonstrated that the action research approach is capable of providing more insight to field experiences than the traditional research method which focuses solely on problem solving.

7. ACKNOWLEDGMENTS

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank pure systems GmbH for support.

8. REFERENCES

[1] *Common Criteria. Java Card Protection Profile - Open Configuration. Version 3.0 (May 2012).*

[2] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen. Action research. *Communications of the ACM*, 42(1):94–97.

[3] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, 2002.

[4] Oracle. *Runtime Environment Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[5] Oracle. *Virtual Machine Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[6] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[7] G. Susman and R. Evered. An assessment of the scientific merits of action research. *Administrative science quarterly*, Jan. 1978.

Bibliography

[1] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," *Proceedings of the IEEE*, vol. 100, pp. 1411–1430, May 2012.

[2] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriquez-Rosell, "A virtual machine emulator for performance evaluation," *Commun. ACM*, vol. 23, pp. 71–80, Feb. 1980.

[3] H. M. Deitel, P. J. Deitel, and D. R. Choffnes, *Operating systems*. Pearson/Prentice Hall, 2004.

[4] W. Rankl and W. Effing, *Smart Card Handbook*. Wiley, 2010.

[5] Oracle, *Java Card 3 Paltform. Runtime Environment Specification, Classic Edition. Version 3.0.4*. September 2011.

[6] Oracle, *Java Card 3 Paltform. Virtual Machine Specification, Classic Edition. Version 3.0.4*. September 2011.

[7] J. Susser, M. Butler, and A. Streich, "Techniques for implementing security on a small footprint device using a context barrier," Jan. 13 2009. US Patent 7,478,389.

[8] J. Susser, M. Butler, and A. Streich, "Techniques for permitting access across a context barrier on a small footprint device using an entry point object," Oct. 20 2009. US Patent 7,607,175.

[9] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller, "Java Bytecode Compression for low-end Embedded Systems," *ACM Trans. Program. Lang. Syst.*, vol. 22, pp. 471–489, May 2000.

[10] J. Staunstrup and W. Wolf, *Hardware/Software Co-Design*. Springer US, ISBN: 1441950184, September 2010.

[11] D. Salomon, *Data Compression: the Complete Reference*. Springer, 2004.

[12] M. Nelson and J.-L. Gailly, *The Data Compression Book 2nd Edition*. 1995.

[13] D. A. Huffman *et al.*, "A method for the construction of minimum redundancy codes," *proc. IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[14] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[15] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *Information Theory, IEEE Transactions on*, vol. 24, no. 5, pp. 530–536, 1978.

[16] P. Katz, "String searcher, and compressor using same," Sept. 24 1991. US Patent 5,051,745.

[17] I. Pavlov, "LMZA," 1999.

[18] P. W. Katz, "PKZIP," *Commercial compression system, version 1*, 1990.

[19] Oracle, "JAR File Specification," 2014.

[20] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, vol. 1009. Pearson/Addison Wesley, 2007.

[21] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler Techniques for Code Compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, pp. 378–415, Mar. 2000.

[22] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, and S. Lucco, "Code Compression," *SIGPLAN Not.*, vol. 32, pp. 358–365, May 1997.

[23] S. Debray and W. Evans, "Profile-guided Code Compression," *SIGPLAN Not.*, vol. 37, pp. 95–105, May 2002.

[24] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, "Improving code density using compression techniques," in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pp. 194–203, Dec 1997.

[25] A. Orpaz and S. Weiss, "A Study of CodePack: Optimizing Embedded Code Space," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, (New York, NY, USA), pp. 103–108, ACM, 2002.

[26] D. R. Ditzel, H. R. McLellan, and A. D. Berenbaum, "The Hardware Architecture of the CRISP Microprocessor," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, ISCA '87, (New York, NY, USA), pp. 309–319, ACM, 1987.

[27] G. Bizzotto and G. Grimaud, "Practical Java Card bytecode compression," in *Proceedings of RENPAR14/ASF/SYMPA*, Citeseer, 2002.

[28] T. A. Proebsting, "Optimizing an ANSI C Interpreter with Superoperators," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '95, (New York, NY, USA), pp. 322–332, ACM, 1995.

[29] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*. Newnes, 2013.

[30] A. Krall and R. Grafl, "CACAO - A 64-bit JavaVM Just-in-Time Compiler," *Concurrency Practice and Experience*, vol. 9, no. 11, pp. 1017–1030, 1997.

[31] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling Java Just in Time," *Micro, IEEE*, vol. 17, no. 3, pp. 36–43, 1997.

[32] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler," *IBM Systems Journal*, vol. 39, no. 1, pp. 175–193, 2000.

[33] J. Aycock, "A Brief History of Just-In-Time," *ACM Comput. Surv.*, vol. 35, pp. 97–113, June 2003.

[34] A. Gal, C. W. Probst, and M. Franz, "HotpathVM: An Effective JIT Compiler for Resource-constrained Devices," in *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, (New York, NY, USA), pp. 144–153, ACM, 2006.

[35] D. Gregg, M. A. Ertl, and A. Krall, "A Fast Java Interpreter," in *Proceedings of the Workshop on Java optimisaton strategies for embedded systems (JOSES), Genoa*, Citeseer, 2001.

[36] K. Casey, D. Gregg, M. A. Ertl, and A. Nisbet, "Towards Superinstructions for Java Interpreters," in *Software and Compilers for Embedded Systems*, pp. 329–343, Springer, 2003.

[37] B. Stephenson and W. Holst, "Multicodes: Optimizing Virtual Machines using Bytecode Sequences," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, (New York, NY, USA), pp. 328–329, ACM, 2003.

[38] M. A. Ertl, C. Thalinger, and A. Krall, "Superinstructions and Replication in the Cacao JVM interpreter," *Journal of .NET Technologies*, vol. 4, no. 1, pp. 31–38, 2006.

[39] H. McGhan and M. O'Connor, "PicoJava: A Direct Execution Engine For Java Bytecode," *Computer*, vol. 31, no. 10, pp. 22–30, 1998.

[40] A. Azevedo, A. Kejariwal, A. Veidenbaum, and A. Nicolau, "High Performance Annotation-aware JVM for Java Cards," in *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, (New York, NY, USA), pp. 52–61, ACM, 2005.

[41] C. Badea, A. Nicolau, and A. V. Veidenbaum, "A Simplified Java Bytecode Compilation System for Resource-Constrained Embedded Processors," in *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, (New York, NY, USA), pp. 218–228, ACM, 2007.

[42] M. Rossi and K. Sivalingam, "A Survey of Instruction Dispatch Techniques for Byte-Code Interpreters," tech. rep., Seminar on mobile code, Number TKO-C-79, Laboratory of Information Processing Science, Helsinki University of Technology, 1995. Bibliography 92 [77], 1996.

[43] I. Piumarta and F. Riccardi, "Optimizing Direct Threaded Code by Selective Inlining," *SIGPLAN Not.*, vol. 33, pp. 291–300, May 1998.

[44] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual Machine Showdown: Stack Versus Registers," *ACM Trans. Archit. Code Optim.*, vol. 4, pp. 2:1–2:36, Jan. 2008.

[45] J. R. Bell, "Threaded Code," *Communications of the ACM*, vol. 16, no. 6, 1973.

[46] P. Klint, "Interpretation Techniques," *Software: Practice and Experience*, vol. 11, no. 9, pp. 963–973, 1981.

[47] C. Porthouse, "High performance Java on embedded devices," *ARM Limited, Oct*, 2005.

[48] S. Steel, "Accelerating to Meet the Challenges of Embedded Java," *Whitepaper, ARM Limited*, 2001.

[49] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man, "Instruction Set Definition and Instruction Selection for asips," in *Proceedings of the 7th international symposium on High-level synthesis*, pp. 11–16, IEEE Computer Society Press, 1994.

[50] M. Arnold and H. Corporaal, "Designing Domain-specific Processors," in *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, CODES '01, (New York, NY, USA), pp. 61–66, ACM, 2001.

[51] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," in *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 209–220, Springer, 2008.

[52] R. Radhakrishnan, R. Bhargava, and L. K. John, "Improving java performance using hardware translation," in *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, (New York, NY, USA), pp. 427–439, ACM, 2001.

[53] D. Wu, L. Wu, and X. Zhang, "Design and fpga implementation of java card coprocessor for emv compatible ic bankcard," in *ASIC, 2009. ASICON '09. IEEE 8th International Conference on*, pp. 971–974, Oct 2009.

[54] J. He, L. Wu, and X. Zhang, "Design and implementation of a low Power Java Coprocessor for dual-interface IC Bank Card," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, pp. 965–969, Oct 2011.

[55] P. Capewell and I. Watson, "A RISC hardware platform for low power Java," in *VLSI Design, 2005. 18th International Conference on*, pp. 138–143, Jan 2005.

[56] J. O'Connor and M. Tremblay, "picoJava-I: the Java virtual machine in hardware," *Micro, IEEE*, vol. 17, pp. 45–53, Mar 1997.

[57] Sun microsystems, *picoJava-II^{TM} Microarchitecture Guide*. March 1999.

[58] Sun microsystems, *picoJava-II^{TM} Programmer's Reference Manual*. March 1999.

[59] Y. Tan, C. Yau, K. Lo, W. Yu, P. Mok, and F. A.S., "Design and implementation of a Java processor," *IEE Proceedings - Computers and Digital Techniques*, vol. 153, pp. 20–30(10), January 2006.

[60] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54, no. 1–2, pp. 265 – 286, 2008.

[61] M. Zilli, W. Raschke, J. Loinig, R. Weiss, and C. Steger, "On the Dictionary Compression for Java Card Environment," in *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, pp. 68–76, ACM, 2013.

[62] M. Zilli, W. Raschke, R. Weiss, J. Loinig, and C. Steger, "Instruction folding compression for java card runtime environment," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pp. 228–235, Aug 2014.

[63] M. Zilli, W. Raschke, R. Weiss, J. Loinig, and C. Steger, "A light-weight compression method for java card technology," in *Proceedings of the 4th Embedded Operating Systems Workshop, 2014*, EWiLi'14, Nov 2014.

[64] M. Zilli, W. Raschke, R. Weiss, J. Loinig, and C. Steger, "A high performance java card virtual machine interpreter based on an application specific instruction-set processor," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pp. 270–278, Aug 2014.

[65] M. Zilli, W. Raschke, R. Weiss, J. Loinig, and C. Steger, "An application specific processor for enhancing dictionary compression in java card environment," in *Proceedings of the 5th International Conference on Pervasive and Embedded Computing and Communication Systems, 2015*, PECCS'15, Feb 2015.

[66] W. Raschke, M. Zilli, J. Loinig, R. Weiss, C. Steger, and C. Kreiner, "Embedding research in the industrial field: A case of a transition to a software product line," in *Proceedings of the 2014 International Workshop on Long-term Industrial Collaboration on Software Engineering*, WISE '14, (New York, NY, USA), pp. 3–8, ACM, 2014.

[67] G. Bouffard and J.-L. Lanet, "The next smart card nightmare," in *Cryptography and Security: From Theory to Applications* (D. Naccache, ed.), vol. 6805 of *Lecture Notes in Computer Science*, pp. 405–424, Springer Berlin Heidelberg, 2012.

[68] M. Lackner, R. Berlach, M. Hraschan, R. Weiss, and C. Steger, "A defensive Java Card virtual machine to thwart fault attacks by microarchitectural support," in *International Conference on Risks and Security of Internet and Systems (CRiSIS)*, pp. 1–8, Oct 2013.

[69] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko, "Heap compression for memory-constrained java environments," *SIGPLAN Not.*, vol. 38, pp. 282–301, Oct. 2003.

[70] M. Kato and C.-T. Lo, "Impact of java compressed heap on mobile/wireless communication," in *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, vol. 2, pp. 2–7 Vol. 2, April 2005.