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

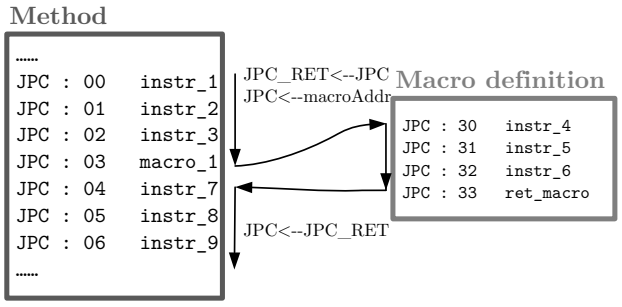

Figure 1: Execution flow in dictionary compressed code

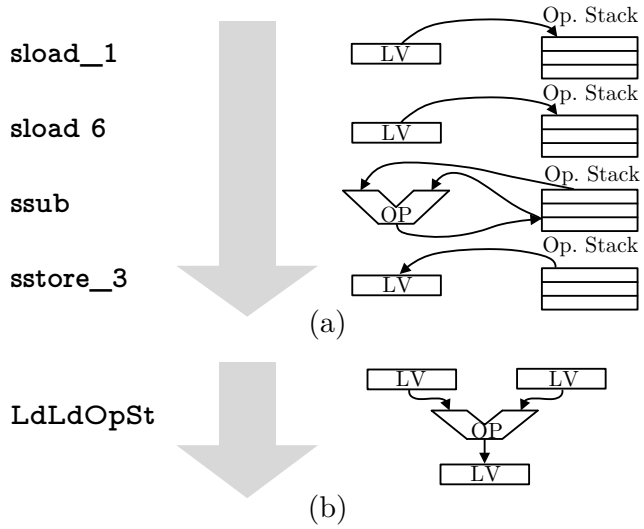

Figure 2: (a) Example of a foldable sequence; (b) Equivalent folded superinstruction

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-

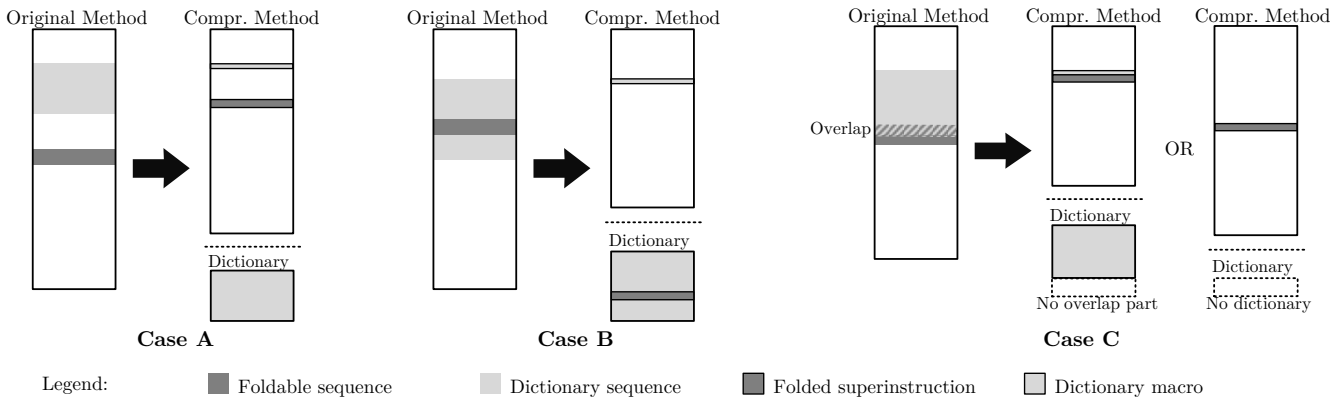

Figure 3: Possible cases in the combination of the compression techniques

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

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

need to know the extended Java Card instruction set used 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).

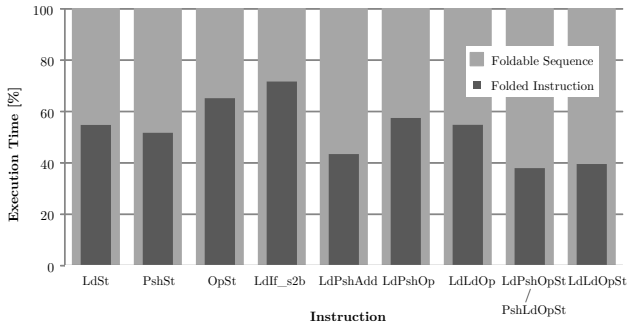

Figure 4: Execution speed-up for foldable sequences

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.