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*

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

Zilli M., Raschke W., Loinig J., Weiss R. and Steger C..
An Application Specific Processor for Enhancing Dictionary Compression in Java Card Environment.
DOI: 10.5220/0005318703050311
In *Proceedings of the 5th International Conference on Pervasive and Embedded Computing and Communication Systems* (PECCS-2015), pages 305-311
ISBN: 978-989-758-084-0

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

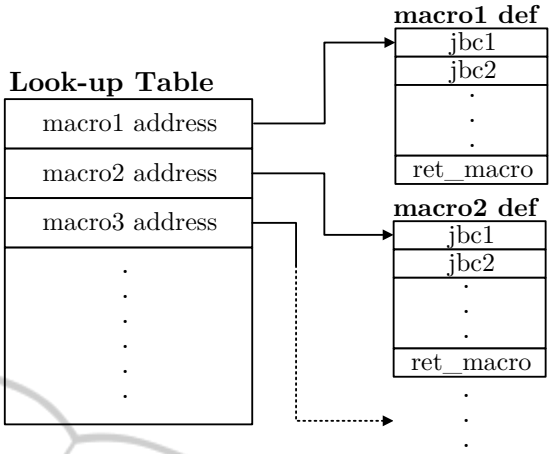

Figure 1: Organization of a dictionary.

```
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 im-

plementation 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.

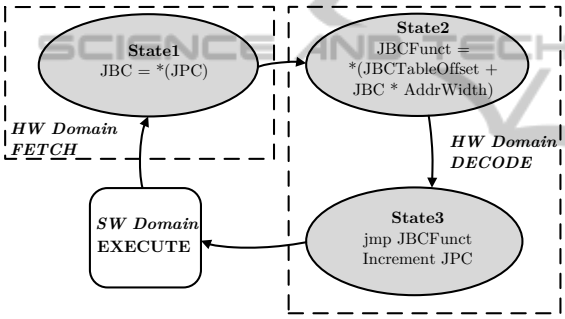

Figure 3: State machine of the intepreter with the fetch and the decode phase performed in hardware.

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 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.

```
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.

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 sketches the finite state machine taking charge of the macro decodification. In the first step, the PC and the JPC are stored in the respec-

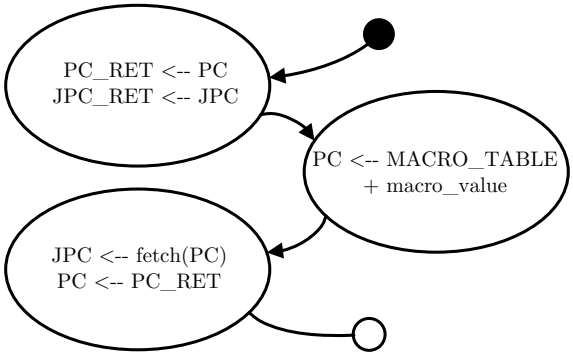

Figure 5: State machine of the PRECALL_MACRO machine opcode.

tive "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	613	5.3	2921	11.4
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

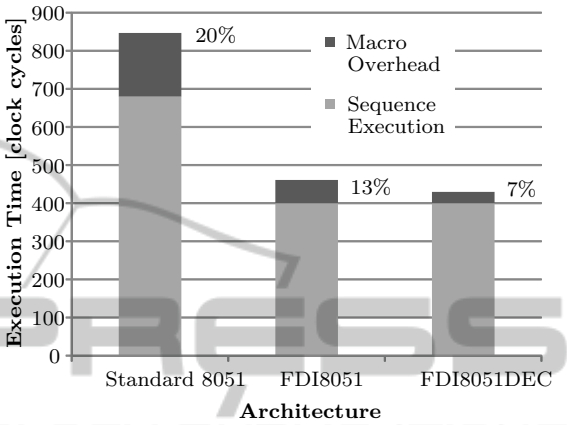

Figure 7: Execuiton time of a macro on three architectures.

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.