A Java Card Virtual Machine Design Based on Off-card/On-card Co-design Pre-processing

Jiaxin Hong(✉), Jianguo Hu, and Ge Lin

School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China
hongjx3@mail2.sysu.edu.cn, hujguo@mail.sysu.edu.cn

Abstract. The design of Java Card Virtual Machine (JCVM) is the critical part in Java Card development. One of the evaluation standards on Java card is the fast response rate. Embedded a high performance JCVM on the memory constrained devices such as smart card is a great challenge. This paper presents an implementation of JCVM and an off-card/on-card co-design pre-processing approach to speed up the interpreter in JCVM. In the off-card domain, we propose moving part of instruction interpreting off card, performing a static analysis on applet files before downloaded. In the on-card domain, a dynamic analysis for external items reference is adopted with a small amount of addition code. The experiment result shows that our proposed scheme has an improvement of 36.3% on execution rate, therefore it is effective to speed up JCVM and it is available for Java card to raise its responsive efficiency.

Keywords: Java Card Virtual Machine (JCVM) · Interpreter
Off-card/on-card co-design · Pre-processing

1 Introduction

Java card is nowadays a development trend of smart cards. It is widely used in the field of finance, identification, transportation and mobile communication. Compared with the native card, Java card can support for multiple applications and provide higher security. Java card specification is a reduced version of Java because of its memory constrained [1–3]. One of the important component of Java card is Java card virtual machine (JCVM), which bridges between the underlying hardware and card applications [4]. The causes for the slow development of Java card include a lack of a large memory chip and a high efficiency JCVM. Therefore, realizing a JCVM with higher execution rate is significant.

Many researches on high performance JCVM have applied JVM optimization algorithms to JCVM. The interpreter in traditional JCVM uses switch-case approach to invoke instructions' handler functions. This method is simple and clean but greatly slows down the execution speed of the interpreter. The most common optimization method for virtual machine is direct threaded interpreter proposed by Ertl in [5]. It utilizes a threaded array to record a series of instructions' handler address, which are converted from executed bytecode and remain operand unchanged. The interpreter now can just execute threaded array without switching and the execution speed can be

W. Xu et al. (Eds.): NCCET 2017, CCIS 600, pp. 11–21, 2018.
https://doi.org/10.1007/978-981-10-7844-6_2

improved greatly. In [6] Gregg et al. uses this method into Java system and it presents a amazing result. However, a handler address needs 32-bit memory in ARM microprocessor which is four times larger than an instruction. It is not suitable for smart card to implement.

Jin et al. in [7] propose reducing times of writing EEPROM to raise execution rate. The authors provide two methods to realize it. One is to implement transaction processing in RAM and the other is to set a Java card object buffer with high locality.

In [8] Zilli et al. design a hardware and software co-design scheme to improve execute rate of JCVM. They integrate part of the interpreter to the microprocessor. Their optimal JCVM architecture is to fetch and decode on hardware and execute with software.

The author Liu et al. in [9] work on instruction folding on Java card. They use shift-reduce method to recognize and fold several instructions into one new instruction defined by authors. The result they presents that this method can reach a gain of 1.24 on financial applet.

The off-card approach is not only instruction folding algorithm, CAO and Ying in [10] research on instruction pre-scheduling algorithm. They analyzed executed instruction flow from applets and then proposed a code arrangement method to realize the pre-scheduling of interpreter. This method has took the program locality into consideration and has increased efficiency of interpreter by improving cache hit rate.

The solutions proposed above can have a good improvement on JCVM. However, using a buffer in RAM memory needs a large amount of memory overhead. Implementing on hardware has high hardware requirement such that it will limit the application and extension. The result of the off-card optimization methods such as instruction folding and instruction pre-scheduling are vary from applet to applet. Thus their running JCVM should be adjusted according to the processing result of each applet such that these approaches are not suitable for multiple applets. Thus, a method that can balance memory cost and execution time for multiple applets in Java card is rarely found.

In the solution that we propose in this paper we combine the off-card and the on-card pre-processing method based on applet file to realize a high efficiency JCVM. In terms of the off-card aspect, we perform a static analysis on constant pool array and the size of the output file will not be changed. As for the on-card aspect we perform a dynamic analysis in order to reference to another applet in card. After optimization, the instructions executing can directly get the offset or address by constant pool array rather than a long time linking process, which can greatly decrease the running time of JCVM.

This paper is organized as follows. Section 2 introduces the framework of the proposed JCVM design and the implementation of each module. Section 3 gives the detail description of the optimization scheme based on applet file. Section 4 shows the experiment result of comparison before and after optimization. Section 5 makes a conclusion for this work.

2 JCVM Implementation

The virtual machine of Java card is divided into the off-card part and the on-card part. In this paper, we focus on the on-card JCVM, which is responsible for interpreting and executing. The optimization of JCVM is based on the implementation of JCVM. In this section, we first introduce the system architecture of JCVM and the design of each modules.

2.1 System Architecture

The core of JCVM is the interpreter whose duty is to fetch, decode and execute instructions. The interpreter has four modules, including stack management module, heap management module, register management module and execution module. The execution module works as a core and the rest are auxiliary modules. RAM is used to store Java stack, register and other runtime data structure. EEPROM is used to store object of different card applets and some other persistent objects. Figure 1 shows the complete framework of the designed interpreter in JCVM.

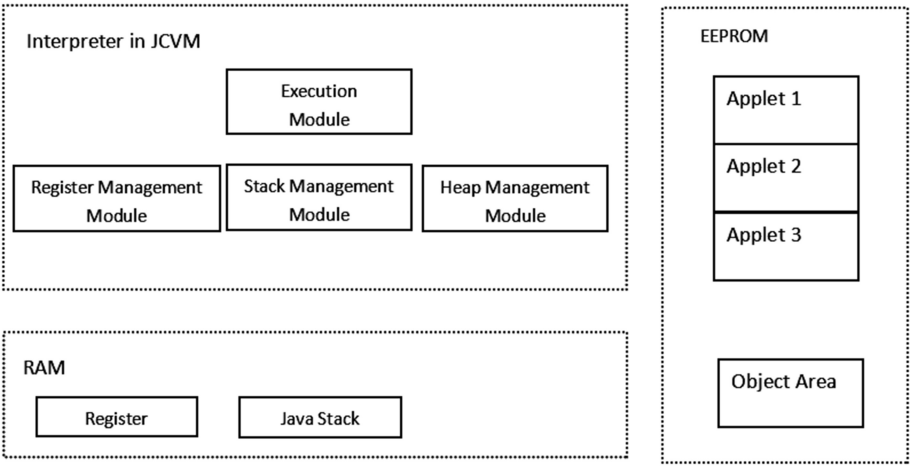

Fig. 1. Overview of the designed interpreter in JCVM

2.2 Execution Module

Traditional interpreter in JCVM uses switch-case to decode and execute instruction after the fetch step. An instruction handler should be found from case zero to the target case. When next instruction was fetched, the program jumped back to the beginning of the loop and find the next target case. This mechanism is time consuming since it has accessed memory twice and jumped once. First memory access is to fetch instruction and the second is to find handler in switch-case.

Unfortunately, direct threaded interpreter is not suitable in smart card, thus in this paper, we use a handler table as a tool for interpreter. The handler table and the instructions are one-to-one relationship. Each element in table stores the instruction handling function address. The instructions provided by Java card specification are numbered from 0 to 184 as operation code such that they can be easily corresponded to the array index. When executing bytecode, we use instruction's operation code as index to get its handler address. This method can obviously reduce the instruction matching time.

2.3 Runtime Data Area

During the process of executing bytecode, the interpreter stores internal data in stack. JCVM runs only one thread thus it only needs one stack in RAM. Java stack is consisted of several frames. Each frame is corresponding to a method invoked. The stack management module is responsible for frame creating, frame pushing and popping. The structure of frame contains three parts: operand stack, local variable area and runtime environment. When VM executes a method invoked by an instruction, it will create a frame and push it into the Java stack. The frame of this method will be pop until this method returns. Figure 2 shows the runtime data area of JCVM and the relationship between frame and register.

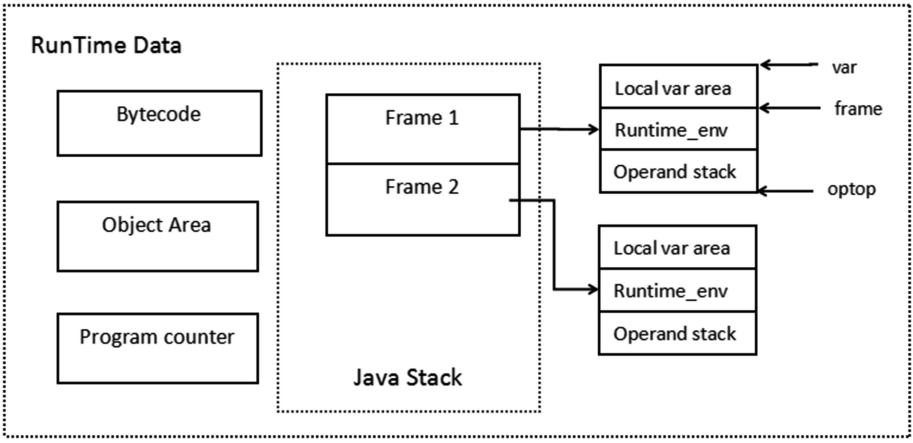

Fig. 2. Runtime data of JCVM

As shown in Fig. 2, there are all four registers needed in JCVM. Three of them record address of data area in frame. Register optop stores the address at the top of the operand stack in the current frame; Register var stores the starting address of local variable area in the current frame; Register frame stores the starting address of runtime environment in the current frame. The rest one register is program counter (pc). It records the address of the executing instruction. Register management module is in charge of these four registers, including initializing, getting and setting registers' value.

Each method invoked by interpreter uses a frame to store internal data. Local variable area is used for passing parameters. Operand stack is used to store calculating data. For example, before the arithmetic computations, the interpreter will first execute two instructions that load two variables to operand stack. Then it comes to an arithmetic instruction, interpreter will pop two values from stack and then push back the computing result. Runtime environment in a frame is used to store information of the caller method.

The duties of stack management module are initializing frame, creating frame, and pushing or popping frame. When a method is called, method arguments will be popped from caller's operand stack and passed to the callees storing in the local variable area of callee. In Java stack, callee's frame adjoins caller's frame and caller's operand stack is next to callee's local variable area. Therefore, in this paper we use Frame Memory Sharing approach to remove the argument passing process. The method makes part of caller's operand stack overlap with part of callee's local variable area such that they share part of data and need not to pass argument. Figure 3 shows the detail of Frame Memory Sharing design.

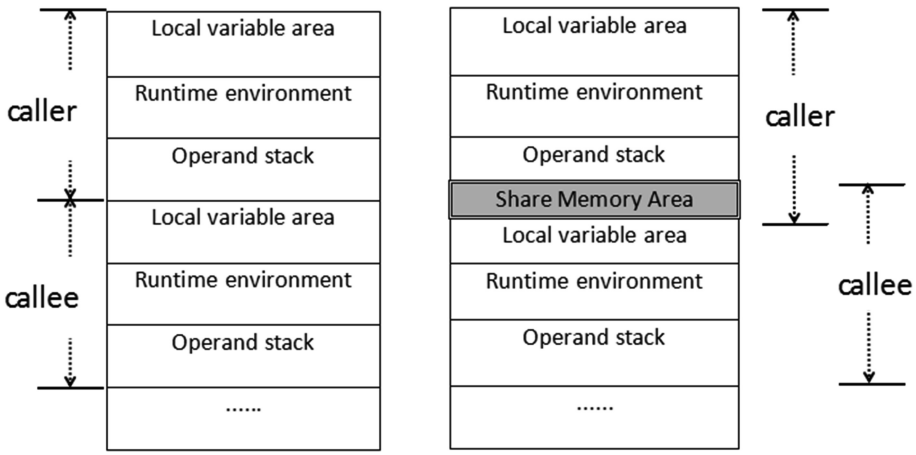

Fig. 3. The design of Frame Memory Sharing

Using this approach, the caller only need to push the passing argument into operand stack and the frame created for the callee can start at the sharing area. It is realized by moving registers. When a new frame create, the registers can be set by following formulas.

$$\text{cur\_var} = \text{pre\_optop} - \text{cur\_methodParaSize}; \tag{1}$$

$$\text{cur\_frame} = \text{pre\_optop} + \text{max\_locals}; \tag{2}$$

$$\text{cur\_optop} = \text{cur\_frame} + \text{FRAME\_SIZE}; \tag{3}$$

The Frame Memory Sharing approach can save not only space of Java stack but also time of passing arguments. When the callee returns, the frame of callee is deleted including sharing area, and then the return value of callee will push into caller's operand stack.

2.4 Implementation of Instruction Handlers

Each method of card applet is stored in method component. It is consist of instructions specified in JCVM specification. An instruction has a single byte operating code, numbers from 0 to 184. After operating code is several bytes of operand. There are seven kinds of instructions: load, store, branch, field operating, object creating and method invoking. A load instruction is to push a value into operand stack. The value may access from local variable area or instruction operand. A store instruction is to popping value from operand stack and store into local variable area. An arithmetic instruction will pop the value to be calculated from operand stack and pushing back the computing result. The types of value used in calculating are short and int. Reference is not allowed to be involved. A branch instruction is to make a comparison and set the program counter to specified location. A field operating instruction deals with fields of objects. An object creating instruction is to new an object or new an array. When comes to a method invoking instruction, it should be analyzed to get method information and then creating frame, set program counter, execute method.

Implementing 184 instructions handling functions is one of the difficult parts in JCVM design. It concerns how fast can JCVM execute an applet. Thus improving interpretation of instructions can influence the execution rate of JCVM. The next section will describe how can the off-card/on-card co-design pre-processing method can reduce the time of interpretation.

3 Pre-processing Based on CAP File

The interpretaion speed of JCVM is one of the most important determinate factors for response rate of Java card. Thus to decrease card response time, one access is to improve performance of the interpreter. Converted Applet (CAP) file is an executable binary file represented for card applet. Our optimization approach for JCVM is a pre-processing method is based on CAP file. In this section we present a detail implementation of the off-card/on-card pre-processing. Before that, we introduce some basic knowledge of linking process scheme in order to have a better understanding of the whole optimization approach.

3.1 Static Analysis and Dynamic Analysis

In Java card, there is a token-based linking scheme such that card applet can be linked to Java card API embedded on the card. When applet converted, every external visible item will be assigned a public token which can be referenced from another card applet. One applet is related to one package. In a package there are three types items can be assigned public tokens, including classes, methods and fields. When JCVM executes

bytecode in method component, some instructions require interpreter to change these tokens into corresponding address or offset, and this process is called instruction analysis. There are two kinds of instruction analysis: static analysis (SA) and dynamic analysis (DA). It is according to whether the analysis is involved external package or just local package.

SA only needs local component. The instruction operand in method component refer to constant pool component, and then refer to class component or static field component according to the specified instruction. DA needs information from other packages, such as linking to API. It can be seen that DA rely on resources in Java card so it must be done on the card. Instruction analysis is the key module in JCVM execution therefore improving efficiency of interpreter will have a considerable result for the whole Java card performance.

3.2 Implementation of Pre-processing

In CAP file, constant pool array in constant pool component contains an element for each class, method, field referenced by items in method component of this CAP file [2]. These reference information link to entry in class component, method component, static field component and import component. When executing some instructions such as method invoking instruction which need linking processing, JCVM can get reference information by accessing constant pool array.

The CAP file pre-processing proposed in this paper is based on analyzing of constant pool array, including off-card SA and on-card DA.

Off-card SA Implementation. Off-card SA is performed by a program. The input is CAP file and output is also a CAP file with the same size. Usually, those instructions that need to be analyzed have one or two bytes operand as an index to constant pool array and get an entry. This entry has 4-byte information. The first byte is a tag, showing that if this entry is a class, a method or a field. If it is a class, then the following are 2-byte class index and 1-byte padding. If it is a method, then the following are 2-byte class index and 1-byte method token. Using these reference information JCVM can locate the position of method, class or field and then replace into constant pool array. Take method invoke instruction as example.

1. When executing an instruction, taking its operand as an index to constant pool array and get a 4-byte information cp;
2. Assume that cp_0 shows the entry is a method, then the next 2-byte token is a class token.
3. Located in class component according to the class token and get the method offset with method token in cp_3;
4. Replace the entry in constant pool array with this method;

Figure 4 shows the off-card pre-processing operation.

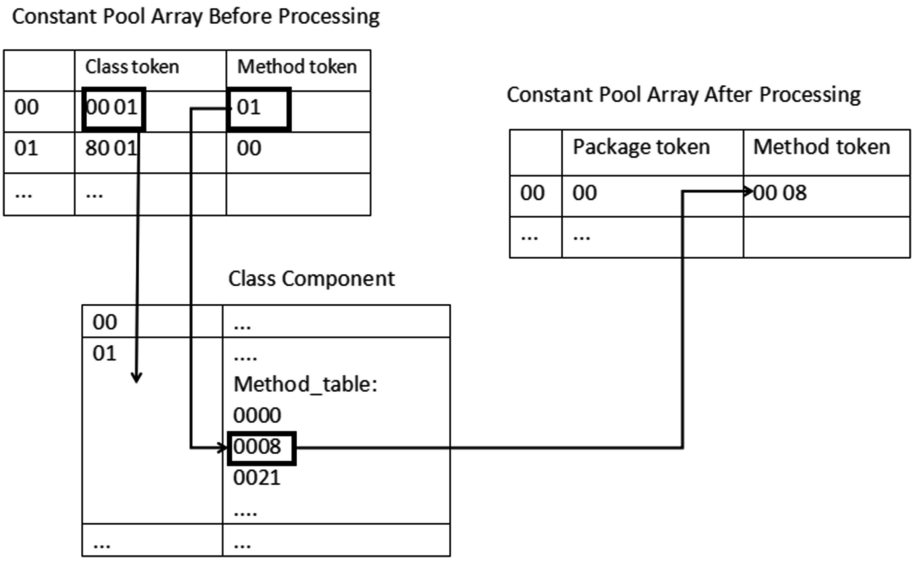

Fig. 4. Overview of the off-card pre-processing operation

Without pre-processing, bytecode in method component should reference to several component to get required information. Every time when JCVM accesses a component, it needs several times memory reading from EEPROM. After pre-processing, bytecode in method component can only reference to constant pool component to get a direct offset to method, class or field. Thus pre-processing can increase executing rate by reducing number of accessing memory.

On-card DA Implementation. On-card DA is performed right after install CAP file onto card. The process is showed as below.

1. When executing an instruction we use its operand as a reference to constant pool array and will get a 4-byte information cp. The first byte cp_0 is also a tag identified this item. For example it is an item for external method.
2. Then the second byte cp_1 is a package token that the highest bit is 1 and the rest 7 bits as an index in import component. In the import component we can find the external package AID with the package token cp_1.
3. Through this AID and applet register table stored in memory, we can get the address of that external package. We need information of its export component.
4. We use cp_2, a class token, as an index to export component to get the external class offset.
5. Find class in external applet class component with class offset and use cp_3, a method token, to find the offset of method.
6. Referred to method component, get the address of this method and replace into constant pool array.

Figure 5 takes method invoke operation as example to show the process of DA.

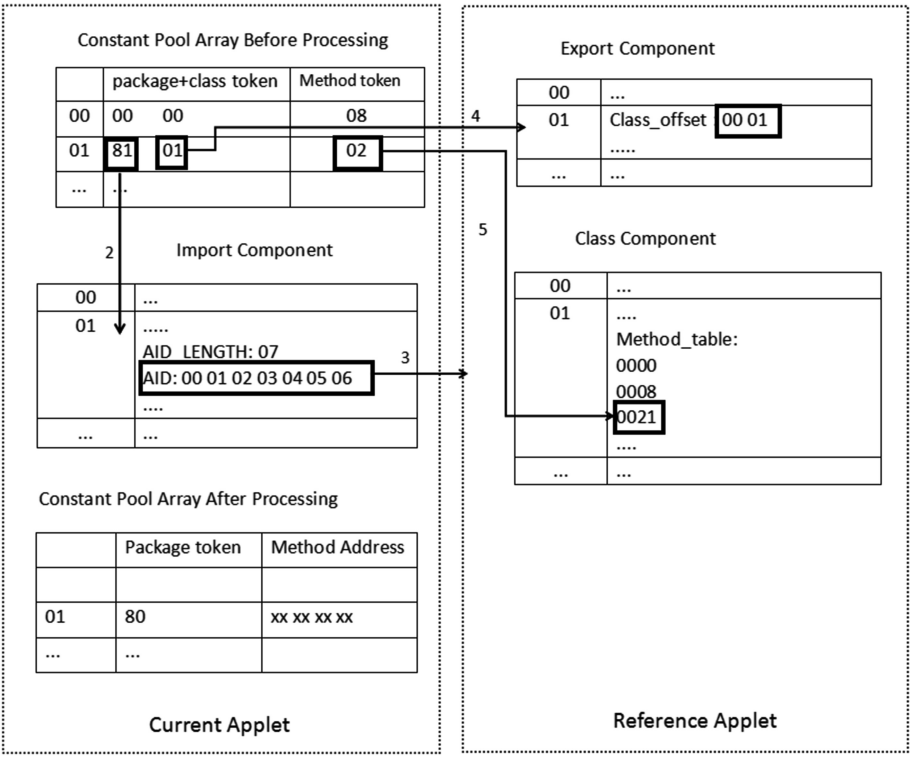

Fig. 5. Overview of the on-card pre-processing operation

In Fig. 5 shows the operation from step 2 to step 5. It is a time-consuming process if it is analyzed in JCVM when executing instruction. The dynamic pre-processing will have an amazing performance if the applet has a great number of external package invoking, or when circularly call the same external method. The pre-processing method makes constant pool array expanded from 4 byte to 6 byte, but considering the improvement to execution rate, this overhead can be accepted.

4 Experiment Result

In this paper, we proposed an implementation of the interpreter in JCVM and an optimization that exploited off-card/on-card co-design pre-processing method to CAP file such that it can reduce the time for the instruction analysis and raise the whole execution rate. In this section we report the experiment result regarding different Java card applets that we took into consideration. To verify the efficiency improvement of the proposed design, we conduct two experiments. First running different functions of Wallet on JCVM and comparing execution time before and after system optimization. Next several typical Java card applets are selected to execute on our designed JCVM with the optimization method proposed and to compare the difference of efficiency

improvement on different applets. Our designed JCVM system is transplanting to the ARM development board. The experiment result are shown in Tables 1 and 2.

Table 1. Performance comparison on different functions of Wallet before and after optimization

Function	Executing time/ms		Efficiency improvement/%
	Original	Optimization	
getBalance	131.2	80.1	38.9%
Debit	573.1	362.2	36.9%
Credit	575.4	370.3	35.6%
Verify	152.1	100.8	33.7%

Table 2. Performance comparison on different Java card samples before and after optimization

Applet	Executing time/ms		Efficiency improvement/%
	Original	Optimization	
JavaPurse	1704.2	1223.3	28.2%
Wallet	367.8	237.6	35.4%
Photocard	3230.3	2276.5	29.5%
JavaLoyalty	61.5	40.1	34.8%
RMIDemo	110.2	80.4	27.0%

From Table 1, we can find that the designed JCVM with the pre-processing approach presents an overall time reduction of 36.3% compared to the design without optimization. In Table 2 as well, the Java card efficiency improvement reaches 30.9% average.

5 Conclusion

The response time is one of the key criterion for Java card. The research we carried out in this paper is about JCVM, a key component in Java card relevant to applets executed. In this paper, we presented the design architecture of JCVM and the implementation of each module. The proposed optimization method combined an off-card static analysis and an on-card dynamic analysis to speed up the execution rate of JCVM. The pre-processing of CAP file is separated from the instruction interpreting process thus it can have a considerable improvement since it has reduced the time in interpreting. The experiment result demonstrated that the pre-processing method can gain an improvement on efficiency of 36.3%.

Acknowledgments. The research was supported by the National Natural Science Foundation of China (No. 61402546). The project has also been supported by two Technology Projects of Guangzhou (No. 201604016126) and (No. 201604010110).

References

1. Oracle, Java Card 3 Platform: Runtime Environment Specification, Classic Edition. Version 3.0.4. Oracle, September 2011
2. Oracle, Java Card 3 Platform: Virtual Machine Specification, Classic Edition. Version 3.0.4. Oracle, September 2011
3. Oracle, Java Card 3 Platform: Application Programming Interface Specification, Classic Edition. Version 3.0.4. Oracle, September 2011
4. Chen, Z.: Java Card™ Technology for Smart Cards. Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison-Wesley Longman Publishing Co. Inc (2000)
5. Ertl, M.A.: Threaded code variations and optimizations. In: EuroForth 2001 Conference Proceedings, pp. 49–55 (2001)
6. Gregg, D., Ertl, M.A., Krall, A.: A fast Java interpreter. In: Proceedings of the Workshop on Java Optimization Strategies for Embedded Systems (JOSES), Citeseer, Genoa (2001)
7. Jin, M.-S., Choi, W.-H., Yang, Y.-S., Jung, M.-S.: A study on fast JCVM with new transaction mechanism and caching-buffer based on Java card objects with a high locality. In: Enokido, T., Yan, L., Xiao, B., Kim, D., Dai, Y., Yang, Laurence T. (eds.) EUC 2005. LNCS, vol. 3823, pp. 91–100. Springer, Heidelberg (2005). https://doi.org/10.1007/11596042_10
8. Zilli, M., Raschke, W., Weiss, R., et al.: Hardware/software co-design for a high-performance Java card interpreter in low-end embedded systems. Microprocess. Microsyst. **39**(8), 1076–1086 (2015)
9. Liu, T., Zhang, D., Jiang, Y.: Research and Implementation of Bytecode Instruction Folding on Java Card (2014)
10. Cao, X., Ying, L.I.: Feedback-based JCVM instruction prescheduling scheme. Comput. Eng. **40**(1), 78–82 (2014)