# A High Performance Buffering of Java Objects for Java Card Systems with Flash Memory †

Min-Sik Jin[1], Min-Soo Jung[2]

[1] Dept of Computer Engineering, Kyungnam University, Masan, Korea,
comsta6@kyungnam.ac.kr
[2] Dept of Computer Engineering, Kyungnam University, Masan, Korea,
msjung@kyungnam.ac.kr

**Abstract.** Java Card technology provides a secure, vendor-independent, ubiquitous Java platform for smart cards and other memory constrained devices such as SIM technology. It is also an open standard in SIM and UIM technology for 3G environment. However, the major point of criticism with regard to Java for smart cards and SIM cards is its low execution speed, aside from its memory demands. We found out that the most long-time work during the execution is to write data to non-volatile memory such as Flash memory. In this paper, we make a suggestion to improve the execution speed by buffering effectively Java Card Objects in order to reduce the number of flush() method. With our approach, the total number of flash writing and the execution speed of applications reduced by about 50% and 38% separately.

## 1 Introduction

In the smart card world, Java Card has been one of the most hyped products around for years. The main reason for the hype is Java Card's potential. Not only would it let all Java programmers develop smart card code, but also such code could be downloaded to cards that have already been issued to customers. This flexibility and post-issuance functionality would significantly extend smart card possibilities. For instance, on the majority of cellular telephone networks, a subscriber uses a smart card commonly called a SIM and UIM card to activated the telephone [1, 3, 8].

Although Java Card is increasingly popular for its predominant traits and potential, now there is an obstacle related to slow execution speed for its fully growth. Aside from its memory demands like small embedded devices, the major point of criticism with regard to Java for smart cards is its low execution speed [3]. Even with a 32-bit processor, the execution speed of Java bytecode executed by JCVM is still 20 to 40 times slower than program code written in C. Several fabulous advantages that Java Card provides such as a dynamic downloading of application called post-issuance and

† **This work is supported by Kyungnam University Research Fund, 2006**
1 Interim Full-Time Instructor of Kyungnam University
2 Professor of Kyungnam University

a platform independency caused Java Card to be used widely by customers in spite of its slow execution speed [3, 4].

A Java Card is essentially an Integrated Circuit Card(ICC) with an embedded JCVM. The Central Processing Unit(CPU) can access three different types of memory: a persistent read-only memory(ROM) which usually contains a basic operating system and the greatest part of the Java Card runtime environment, a persistent read-write memory(EEPROM) which can be used to store code or data even when the card is removed from the reader, and a volatile read-write memory(RAM) in which applications are executed [4]. The JCVM controls the access to all smart card resources, such as memory and I/O. Especially, during the execution of an application, data and Java Card Objects created usually are stored in persistent memory such as EEPROM or flash memory to prevent a serious data loss against a power-down [2, 3].

We found out that there are too many flash writings while Java Card downloads a small application called *an applet* from a terminal and executes an installed applet with Application Protocol Data Unit(APDU). We also discovered that it makes the execution speed of Java Card much slowly.

In this paper, we make a suggestion to improve the execution speed by using a high performance object buffer that was very effectively implemented in order to reduce the number of flush() method that means one block writing into flash memory. Our approach came from the writing time per each memory cell. Namely, writing operation of RAM memory cell is approximately 2,000 times faster than that of flash memory. Flash memory is also more common than EEPROM memory in the embedded system, because EEPROM can be written by a byte unit, but flash can be written by a block unit.

This paper is organized as follows. Section 2 describes the feature of each memory in a typical Java Card, Java Card objects and the method that writes data to flash. Section 3 explains about object's high locality and how to write objects into flash in a traditional Java Card. Section 4 describes the possibility of our approach with some data and outlines our approach about new object buffer based on a high locality of Java Card objects. Section 5 discusses the evaluation between a traditional one and our approach. Finally, we present our conclusions in Section 6.

## 2 Java Card Environment

### 2.1 Java Card with Flash Memory.

A early smart card system including Java Card has three kinds of memory: ROM, RAM, and EEPROM. However, the use of flash memory, not EEPROM as a non-volatile memory is dramatically increasing in a smart card system for several reasons. EEPROM can be electrically programmed one byte at a time, but flash can be programmed a large group of bytes or words called a block, sector, or page. The difference of this writing process makes a card with flash memory faster than a card with EEPROM [1, 3, 4, 5].

In the Java Card system with flash memory, the JCRE code including JCVM, API and COS is placed in ROM memory. Persistent data such as post-issuance applet classes, applet instances and longer-lived data are stored in flash memory. RAM is used for temporary storage. The Java Card stack is allocated in RAM. Intermediate results, method parameters, and local variables are put on the stack [3,5].

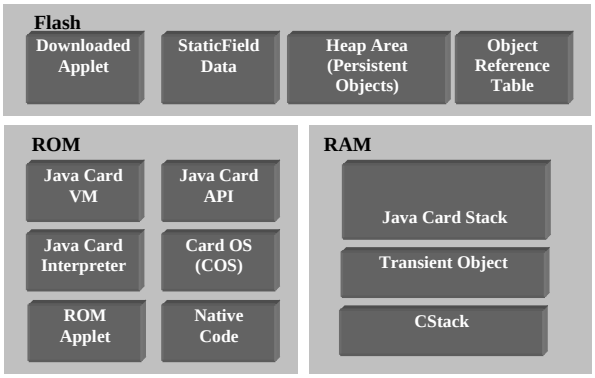


**Fig. 1**. Java Card Memory Model with Flash memory and the components that each memory has.

As illustrated in Figure 1, Java Card technology supports both persistent and transient objects. Java Card Objects are inherently created as persistent objects in flash when they are generated using the new() method. Persistent objects survive both the end of a session and a sudden loss of power, without losing data or consistency [3].

The applet instance and associated persistent objects of an application must survive a session. Therefore they are placed in the non volatile storage on a card, usually flash. Flash provides similar read and write access as RAM does. However, The difference of both memory is that writing operations to flash memory are typically over 2,000 times slower than to RAM and the possible number of flash writing over the lifetime of a card is physically limited [4, 16].

**Table 1.** Comparison of memory types used in Smart Card microcontrollers [4, 16]

| Type of Memory | Number of write/erase cycles | Erase time per a cell | Writing time per a cell |
|---|---|---|---|
| **RAM** | **unlimited** | **-** | **≈70ns** |
| EEPROM | 100,000 – 1,000,000 | ≈1.5 ~ 2ms | ≈1.5 ~ 2ms |
| **Flash** | **10,000** | **4ms** | **≈20≠s** |

### 2.2 How to write data into Flash Memory

A single flash write operation consists of an erase and a write operation. Erase and write operation is performed in a page unit. The size of a page unit depends on a chip manufacturer such as ARM, Philips and Samsung. Its size is generally between 128-byte and 256-byte. Figure 2 shows how to write data into flash memory with one buffer in RAM. First, a page in flash memory should be saved into the buffer in RAM

for backup. Only the data to be written or changed in the buffer in RAM is updated. after updating, the target page in flash is erased and the buffer in RAM is written to the target page in block [14, 16].

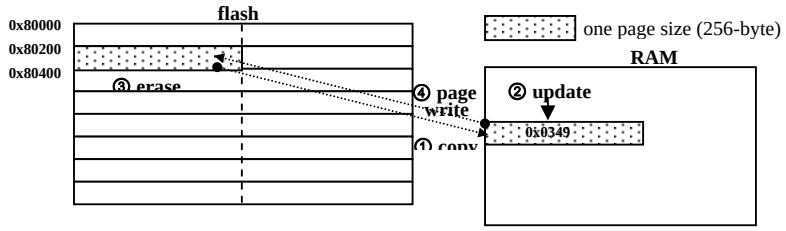


**Fig. 2.** Flash memory writing using a buffer in RAM and the procedure for writing 0x0349 into the target address, 0x80103.

### 2.3 Data Writing mechanism in Java Card

The JCRE has a flash writing mechanism to store consecutive bytes by providing an abstraction of flash as a stream. This mechanism also provides atomic operations of data to be written into flash. Namely, the Java Card platform should guarantee that creating and updating to a persistent object is atomic. The reason is that there is a high risk of failure at any time during applet execution with smart cards.. Failure can easily happen due to a computational error or more often, a user of the smart card may accidentally remove the card from the CAD [1, 4]. For this reason, the Java Card system uses the buffering of consecutive data to keep an data integrity.

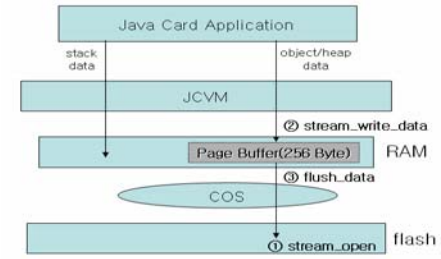


**Fig. 3**. How to write consecutive bytes to destination address by using stream_open() and flush() method.

Figure 3 shows how to write data by using a stream method in a traditional Java Card. This stream form is used to gather consecutive data to reduce the number of flash writing. This also allows the low-level layer to use a buffer and wait until one full page is completed before writing it to flash. Here is a whole process to write some data to flash memory; First, the stream of the target address to be written is opened, and then, one or more data such as byte, short, boolean and array are consecutively written into the buffer in RAM. Finally, after gathering some data into the buffer, these data in the buffer are stored with flush() method.

In this mechanism, if the addresses of flash writing are not consecutive during the execution of an applet, or the size of data written into flash is generally small, the performance of this buffer mechanism is very inefficiency. In other words, the main purpose of this buffer in RAM is simultaneously to write 1-byte up to consecutive 256-byte data into flash at a time.

If this buffer in RAM has this simple function as well as a buffering and caching function based on a nearness of flash writing addresses caused by the feature of Java Card Objects, the number of flash write operations that spends more time than other operations are reduced. It makes the Java Card much faster.

## 3 Object writing of a traditional Java Card

### 3.1 Object Representation in Java Card

The structure of objects is defined by a sort of fields that a class has; primitive types, primitive arrays, static types and reference types. If a class has some primitive type fields, the value of this primitive fields such as byte, short and boolean is in the object. If a class has fields as another class like a API class, the object of this class only includes the object id of another class. If a class has static variables and static arrays as a field, the object of this class does not have any information about this static data. It is managed by the JCVM in a special area, static field area, in flash memory.

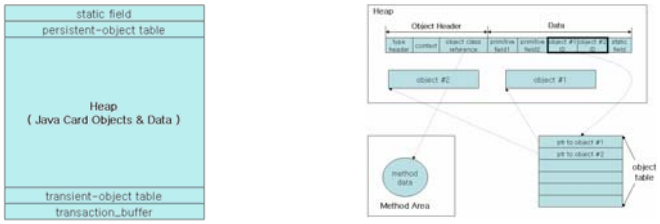


**Fig. 4**. The whole structure of flash memory consisting of 5 parts and the format of Java Card objects stored in heap area.

### 3.2 Object's high locality of Heap area

As mentioned earlier, a traditional Java Card System has only one buffer in RAM to write data into flash. The buffer has a function for the buffering of just consecutive bytes. In this paper, we suggest an object buffer that perform a buffering and caching to improve the execution speed of Java Card. The most important and considerable point in order to add caching function to Java Card is a high hitting rate of the caching buffer.

```
public class wallet extends Applet{
    int balance;                  ┐
    int withdraw;                 ├ global variables
    OwnerPIN pin;                 → reference class

    wallet (){     // constructor
      pin = new OwnerPIN(3, 8);  // create OwnerPIN(trylimit, Pinsize) object
    }
    initialize(){
       balance = 90;
    }
    withdraw(){  // method
        withdraw = 50;
        balance = balance – withdraw;
    }
}
```

**Fig. 5**. wallet applet that has 3 methods and 3 fields; when the wallet applet is created by install() method, OwnerPIN object also is created in wallet() constructor.

When the wallet class is created by install() method, the wallet object (2011C3A600000000) that have 3 fields is first written in flash, and then, OwnerPIN object (20111E69000308) that assigned 0045 as an objectID is created and written in EEPROM. After the OwnerPIN object created, Java Card writes the objectID (0045) as pin reference field of the wallet object (2011C3A60000**0045**). After the wallet applet is created, a method such as initialize() and withdraw() generally would be invoked. In figure 4, initialize() method is to change the value of balance field into 100. After this operation, the content of the wallet object is 2011C3A6**90**000045. withdraw() method also changes the field value of withdraw and balance into 50 and 40 separately. At this time, the content of the wallet object is 2011C3A6**4050**0045.

Figure 5 and 6 showed several flash writing processes from the creation of wallet applet to the execution of methods such as initialize() and withdraw(). If Java Card just performs these processes by using one buffer above-mentioned, it might spends much time in writing and changing localized-data like above example.

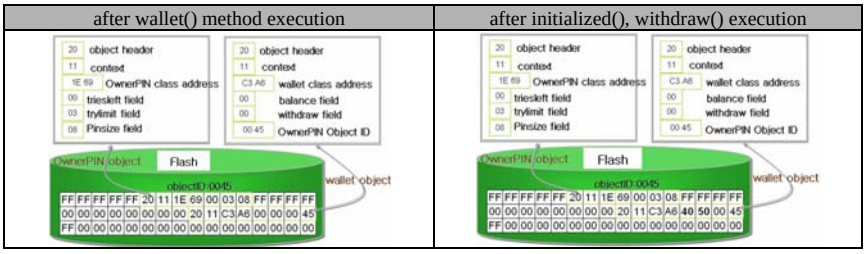


**Fig. 6**. The creation process of the wallet applet and the OwnerPIN object in flash and the process of the changing localized-fields and rewriting them.

## 4 Our changed Java Card with a high performance buffering

### 4.1 high locality of Java Card Objects

In chapter 2, we explained how to gather data into the buffer by using a stream method in a traditional Java Card. The purpose of this mechanism is to reduce the number of Flash writing by buffering only consecutive data in address and to keep the

data integrity of Java Card. However, this buffer was made without any consideration about a high-locality of heap area caused by the feature of Java Card Objects.

For a logicality of our study, we investigated the size of all data written into flash memory and how the addresses of flash writing are close to each other in a traditional Java Card.

**Table 2.** Comparison of the size of all data written into flash memory during the downloading and execution of applications.

| Size / Applet | 1 byte | 2 bytes | 3 bytes | 4 bytes | over 4 bytes | Total number of flash writing |
|---|---|---|---|---|---|---|
| HelloWorld | 2481 | 920 | 8 | 30 | 1127 | 4566 |
| ChannelDemo | 4305 | 1428 | 10 | 4305 | 1942 | 11990 |
| Wallet | 3049 | 1085 | 7 | 28 | 1373 | 5542 |
| PackageA | 5505 | 1784 | 7 | 30 | 2426 | 9752 |
| SampleLibrary | 1929 | 754 | 6 | 27 | 859 | 3575 |
| Demo2 | 24399 | 6838 | 68 | 142 | 10857 | 42304 |
| EMVL | 3022 | 1044 | 8 | 29 | 1394 | 5497 |
| EMVL_Applet | 3591 | 1162 | 20 | 36 | 1676 | 6485 |

As illustrated in table 2, the rate of flash writing less than 4 bytes in size is more than 70%. We found also out how many percentages next flash writing is in the current page. Flash writing in only heap area is checked to figure out the locality of a heap that Java Card Objects are saved. As shown in table 3, there is very high probability that the next target address will be in the current page during the downloading and execution of application.

**Table 3.** Comparison of total number of flash writing and the number of next flash writing in the current page.

| Applet | Total number of flash writing | Next flash writing in the current page |
|---|---|---|
| HelloWorld | 4566 | 2235 |
| ChannelDemo | 11990 | 7665 |
| Wallet | 5542 | 2737 |
| PackageA | 9752 | 4876 |
| SampleLibrary | 3575 | 1750 |
| Demo2 | 42304 | 20748 |
| EMVL | 5497 | 2683 |
| EMVL_Applet | 6485 | 3095 |

### 4.2 Our changed Java Card with an efficient Object buffering

In a traditional Java Card, the buffer is used to gather data to RAM by using a stream method in a traditional Java Card. The purpose of this mechanism is to reduce the number of flash writing and to keep the data integrity of Java Card. However, as illustrated in chapter 3, this method is not efficient and enough to reduce flash write operations, because only consecutive data is gathered. To improve the performance of this buffer, we investigated all addresses that JCVM writes during the execution. After that, we discovered that all objects and data that the Java Card creates during the execution have a high locality. It means that an additional caching and buffering function makes the number of flash writing go down. For these reasons, we developed new Java Card with two buffer in RAM; one is the existing buffer for non-heap area, another is for heap area in flash. The heap area is where objects created by Java Card is allocated.

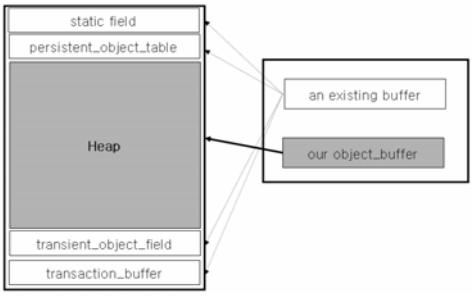


**Fig. 7.** An existing buffer for non-heap area and our object buffer for heap area with very high object locality.

In our changed Java Card, the existing buffer is similar to that of a traditional Java Card in terms of a size and a function. The existing buffer can write between 1 byte and up to 256 consecutive bytes to non-heap area at a time. However, our object buffer is for only heap area in flash. This object buffer of 256 bytes can be programmed to one page of flash memory simultaneously.

Figure 8 below shows the main algorithm using a existing buffer and our object buffer. The writing of non-heap area is performed with the existing page buffer. The writing of heap area is executed with our object buffer. When the Java Card writes data into flash memory, the first operation is to check if the target address is in heap area or not. If the target address is in non-heap area, this data is written into the existing buffer, and then the buffer is flushed. Otherwise, the target address is checked if it is within our object buffer or not. If the target address is not in our object buffer, a current object buffer will be first flushed into flash memory, and then, one page that the target address belongs is copied to the object buffer.

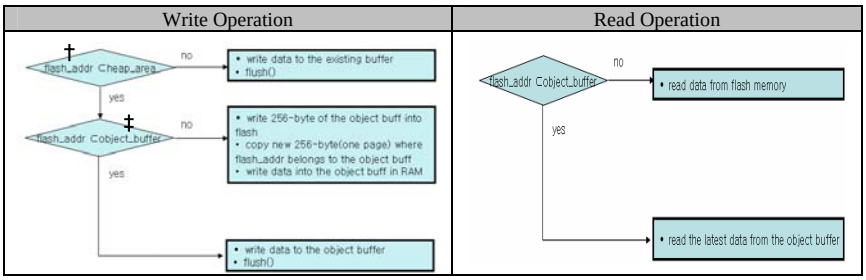


**Fig. 8.**writing mechanism of our approach to reduce the number of flash writing and reading mechanism of our approach to get up-to-data values from the object buffer(†flash_addr : the flash address that data will be written, ‡ object-buffer : our new object buffer with caching and buffering function for just heap area in flash)

In our algorithm, the up-to-data values during the execution might be in the object buffer. It means that a reading mechanism in order to read the up-to-data values is needed. Even though it is an additional operation to JCVM, the situation can be better

in case of the much more expensive write operation. As mentioned earlier, writing costs are more expensive than reading costs and flash writing costs also are much more expensive than RAM writing costs [10].

## 5 Evaluation of Our approach

The key of our approach is improve an execution speed of the Java Card by reducing the number of flash writing. The main idea is also that flash writes are typically more than 2,000 times slower than writes to RAM. One of the analyzed results of a traditional Java Card is that the existing buffer algorithm is to write data to flash regardless of the high locality of Java objects stored in heap area. For this reason, we developed new object buffer algorithm.

In the GSM and Ubiquitous environment, one of the most important point is the downloading time of applications [10, 11]. We examined our approach with many applets. We tested our algorithms in terms of speed and the number of flash writing. To get more precise figure regarding our approach to the real Java Card, we made an experiment with S3FS9SK [16] with ARM 32-bit Microcontroller for contact smart cards.

Below Table 4 and Figure 9 show the comparison between a traditional Java Card and our changed Java Card in regard to the number of EEPROM writing and the execution speed. During the dynamic downloading of applets called a post-issuance, the speed of downloading, installation and execution is also reduced by 44%. Consequentially, the reduced EEPROM writing caused Java Card to improve an execution speed.

**Table 4.** The comparison between a traditional Java Card and our changed Java Card with regard to an execution speed. (Experiment is made with ARM 32-bit Microcontroller for contact smart cards)

| Applets | Original | Our approach | Reduced Rate |
|---|---|---|---|
| Channel Demo | 5323 | 2928 | 45% |
| JavaLoyalty | 5654 | 3505 | 38% |
| JavaPurse | 16458 | 9381 | 43% |
| ObjDelDemo | 12656 | 8226 | 35% |
| PackageA | 6578 | 3618 | 45% |
| PackageB | 5686 | 3525 | 38% |
| PackageC | 2348 | 1479 | 37% |
| Photocard | 3946 | 2525 | 36% |
| RMIDemo | 3956 | 2374 | 40% |
| Wallet | 3544 | 2410 | 32% |
| EMV small Applet | 4387 | 2895 | 34% |
| EMV Large Applet | 8542 | 5467 | 36% |
| **Average** | | | **38%** |

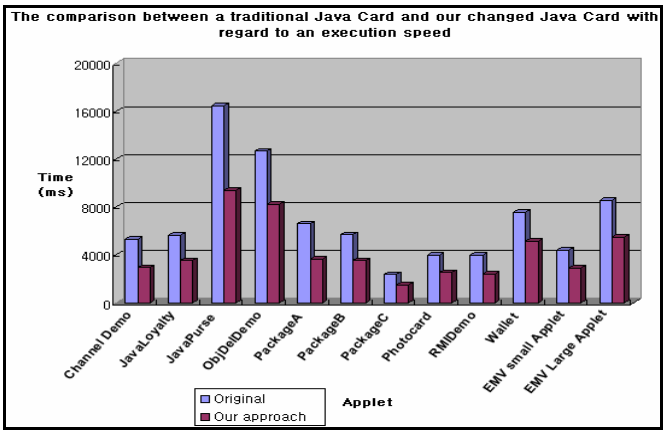


**Fig. 9** The comparison between a traditional Java Card and our changed Java Card with regard to an execution speed.

One applet consists of over 11 components that include all information of one applet package. We also produced the downloading result about all component of Wallet applet.

| Component | Tradi-tional | Our Approach | Reduction |
|---|---|---|---|
| Initialize | 280 | 160 | 120 |
| Select Install | 268 | 143 | 125 |
| CAP Begin | 226 | 124 | 102 |
| Header | 374 | 241 | 133 |
| Directory | 257 | 155 | 102 |
| Import | 240 | 134 | 106 |
| Applet | 310 | 204 | 106 |
| Class | 286 | 190 | 96 |
| Method | 568 | 446 | 122 |
| StaticField | 406 | 398 | 8 |
| ConstantPool | 740 | 690 | 50 |
| RreferenceLocation | 2819 | 1531 | 1288 |
| CAP End | 218 | 201 | 17 |
| Create Applet | 548 | 513 | 35 |
| Total | 7544 | 5130 | 2414 |

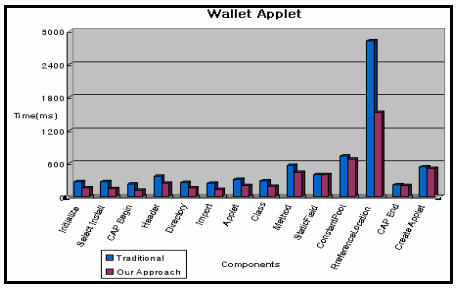


**Fig. 10.** The comparison between a traditional Java Card and our changed Java Card in regard to Wallet applet's downloading.

## 6 Conclusion and Future Work

Java Card technology provides a secure, vendor-independent, ubiquitous Java platform for smart cards and other memory constrained devices such as SIM technology. It is also an open standard in SIM and UIM technology for 3G environment [11]. However, a Java language is basically slower than other languages, the reasons why Java Card technology is selected as a standard are a post-issuance and a platform independence. When Java Card downloads new application, a post-issuance generally spends a lot of time [10, 11].

In this paper, we have proposed the method to reduce the number of flash writing with our new object buffer based on the high locality of Java Card objects. It also

makes the execution speed of Java Card faster. Even though our approach is very simple, with our approach, the number of flash writing and the downloading speed reduced by about 50% and 38% separately. It also enables an application to be downloaded more quickly in the case of an application sent to a mobile phone via the GSM network (SIM).

## References


1. Sun Microsystems, Inc. JavaCard 2.2.1 Virtual Machine Specification. Sun Microsystems, Inc. URL: http://java.sun.com/products/javacard (2003).
2. Sun Microsystems, Inc. JavaCard 2.2.1 Runtime Environment Specification. Sun Microsystems, Inc. URL: http://java.sun.com/products/javacard (2003).
3. Chen, Z. Java Card Technology for Smart Cards: Architecture and programmer's guide. Addison Wesley, Reading, Massachusetts (2001).
4. W.Rankl,. W.Effing,. : Smart Card Handbook Third Edition, John Wiley & Sons (2001).
5. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha.: The Java Language Specification, Second Edition. Addison-Wesley, http://java.sun.com/docs/books/jls/index.html (2001).
6. Marcus Oestreicher, Ksheerabdhi Krishna. : USENIX Workshop on Smartcard Technology, Chicago, Illinois, USA, May 10–11, (1999).
7. M. Oestreicher and K. Ksheeradbhi, "Object Lifetimes in JavaCard," Proc. Usenix Workshop Smart Card Technology, Usenix Assoc., Berkeley, Calif., (1999) pp 129–137.
8. Michael Baentsch, Peter Buhler, Thomas Eirich, Frank Höring, and Marcus Oestreicher, IBM Zurich Research Laboratory, Java Card From Hype to Reality (1999).
9. Pieter H. Hartel , Luc Moreau. : Formalizing the safety of Java, the Java virtual machine, and Java card, ACM Computing Surveys (CSUR), Vol..33 No.4, (2001) pp 517-558.
10. M.Oestreicher, "Transactions in JavaCard,", Proc. Annual Computer Security Applications Conf., IEEE Computer Society Press, Los Alamitos, Calif., to appear, Dec. (1999).
11. Kim, J. S., and Hsu, Y.2000. Memory system behavior of Java programs: methodlogy and analysis. In Proceedings of the ACM Java Grande 2000 Conference, June.
10. http://www.gemplus.com. : OTA White Paper. Gemplus (2002).
11. the 3rd Generation Partnership Project. : Technical Specification Group Terminals Security Mechanisms for the (U)SIM application toolkit. 3GPP (2002).
12. MCULAND, http://mculand.com/e/sub1/s1main.htm.
13. X. Leroy. Bytecode verification for Java smart card. Software Practice & Experience, (2002 ) pp 319-340
14. SAMSUNG, http://www.samsung.com/Products/Semiconductor
15. SIMAlliance, http://www.simalliance.org
16. http://www.samsung.com/Products/Semiconductor/Support/ebrochure/systemlsi/smartcard_controller_200511.pdf
17. Min-Sik Jin, Won-ho Choi, Yoon-Sim Yang, Min-Soo Jung. "The research on How to Reduce the Number of EEPROM Writing to Improve Speed of Java Card", ICESS 2005, pp 71-84.
18. Min-Sik Jin, Min-Soo Jung. "A Study on How to Reduce Time and Space by Redefining New Bytecode for Java Card, RTCSA 2005, pp 551-554.