A Study on Fast JCVM with New Transaction Mechanism and Caching-Buffer Based on Java Card Objects with a High Locality*

Min-Sik Jin[1], Won-Ho Choi, Yoon-Sim Yang, and Min-Soo Jung[2]

Dept of Computer Engineering, Kyungnam University, Masan, Korea
{comsta6, hoya9499, ysyang}@kyungnam.ac.kr
msjung@kyungnam.ac.kr

Abstract. Java Card is now a mature and accepted standard for smart card and SIM technology. Java Card is distinguished primarily by its independence from hardware platforms and portability and is now the most important open standard. However, the main concern of Java Card is now its low execution speed caused by the hardware limitation. In this paper, we propose how to improve a execution speed of Java Card by reducing the number of EEPROM writing. Our approaches are an object-buffer based on a high locality of Java Card objects, the use of RAM, has a speed more faster 1000 times than EEPROM, as much as possible and new transaction mechanism using RAM.

1 Introduction

Java Card technology [1, 2, 3] enables smart cards and other devices with very limited memory to run small applications, called applets, that employ Java technology such as a platform independence and a dynamic downloading(post-issuance). For these reasons, Java Card technology is an accepted standard for smart card and SIM technology [15]. SIM cards are basically used to authenticate the user and to provide encryption keys for digital voice transmission. However, when fitted with Java Card technology, SIM cards can provide transactional services such as remote banking and ticketing, and also service a post-issuance function to manage and install applications in cards after the cards issued [1, 3, 15].

Java Card uses generally RAM and EEPROM. The difference of both memory is that writing operations to EEPROM are typically more than 1,000 times slower than to RAM. In a traditional Java Card, the specific area, transactionbuffer(T_Buffer), in EEPROM is used to support an atomicity and transaction [1, 3]. It makes the speed of the Java Card more slowly. In addition to the transaction mechanism, a traditional Java Card has a low-level EEPROM writing with a page-buffer. The size of a page-buffer depends on platforms such as ARM, Philips and SAMSUNG [15]. This page-

* This work is supported by Kyungnam University Research Fund, 2005.
[1] Ph.D Student of Kyungnam University.
[2] Professor of Kyungnam University.

T. Enokido et al. (Eds.): EUC Workshops 2005, LNCS 3823, pp. 91 – 100, 2005.

buffer is just to write one byte or consecutive bytes less than the size of the page-buffer at a time into EEPROM. However, this page-buffer of Java Card generally is made regardless of a high locality of Java Card Objects [5, 7].

In this paper, we suggest two ideas to improve the speed of Java Card. One is new transaction mechanism in RAM, not EEPROM. Another is new object-buffer based on a high locality of Java Card objects to support a caching and buffering of heap area.

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 EEPROM. Section 3 explains about a transaction and object writing of a traditional Java Card using a lot of EEPROM writing. Section 4 outlines our approach about new transaction mechanism using RAM and 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 The Java Card Environment

2.1 Different Types of Memory in Java Card

A typical Java Card system places the JCRE code(virtual machine, API classes) and COS in ROM. 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. persistent data such as post-issuance applet classes, applet instances and longer-lived data are stored in EEPROM [3,5].

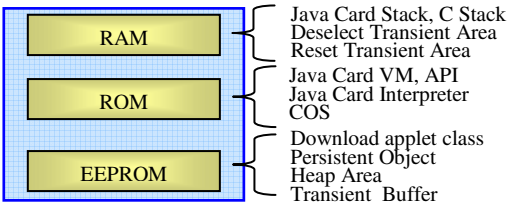


Fig. 1. The general memory model of Java Card that is consisted of three areas and its contents

EEPROM provides similar read and write access as RAM does. However, The difference of both memory is that writing operations to EEPROM are typically more than 1,000 times slower than to RAM and the possible number of EEPROM writing over the lifetime of a card is physically limited [4].

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

Type of Memory	Number of write/erase cycles	Writing time per memory access	Typical cell size with 0.8-μm technology
RAM	**unlimited**	**70 ns**	**1700 μm^2**
EEPROM	**10,000 – 1,000,000**	**3-10 ms**	**400 μm^2**

2.2 How to Write Objects in EEPROM in Java Card

In the latest release, Java Card 2.2.1, one EEPROM mainly consists of 3 areas; ***static field area***, ***heap area*** to save many Java Card objects including transient object table(TOT) and persistent object table(POT) and ***transactionbuffer(T_buffer area)***[7].

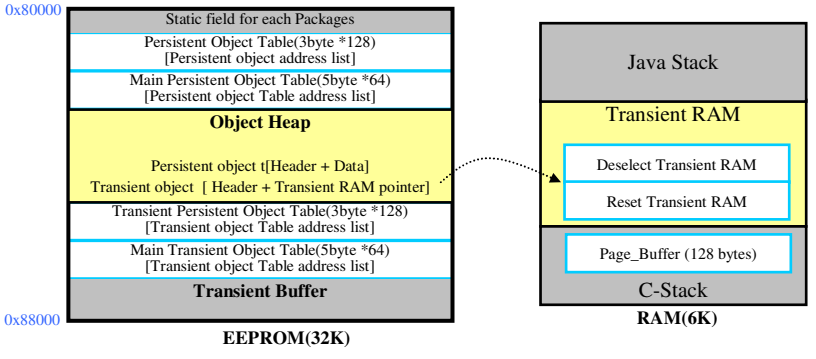


Fig. 2. The inner structure of RAM and EEPROM consisting of several areas. Especially, all objects that are made by Java Card is saved in Heap area with a high locality.

A transaction mechanism [10] using the T_Buffer area in EEPROM is used to support an atomicity [3]. In a traditional Java Card, to support this transaction, the Java Card temporarily saves old_data in T_Buffer in EEPROM until the transaction is complete.

In a point of COS's view lower level than Java Card, smart cards such as Java Card use only one page-buffer in RAM to write data in EEPROM,. The size of the page-buffer depends on platforms such as ARM, Philips and SAMSUNG. In fact, the data is first written into the page-buffer in RAM, when the Java Card writes one byte or consecutive bytes less than the size of the page-buffer into EEPROM. However, the most important point about writing operation using the page-buffer is that the writing time of both 1 byte and 128 consecutive bytes is almost the same.

3 A Transaction and Object Writing of a Traditional Java Card

3.1 Atomic Operations and Transaction in a Traditional Java Card

A transaction is a set of modifications performed atomically, which means that either all modifications are performed or none are performed. This is particularly for smart cards, because the card reader powers them: when you unexpectedly remove the card from the reader (this is called "tearing"), it's possible that you're interrupting a critical operation that needed to run to completion. This could put the card in an irrecoverable state and make it unusable.

To prevent this, the Java Card platform offers a transaction mechanism. As soon as a transaction is started, the system must keep track of the changes to the persistent environment(EEPROM). The Java Card must save old_value of EEPROM address

that will be written into a particular area(T_Buffer) in EEPROM. In other words, If a transactional computation aborts, the Java Card must be able to restore old_value from the T_Buffer in EEPROM to its pervious position.

In case of commit, the check_flag byte of the T_Buffer must just be marked invalid and the transaction is completed. In case of abort, the saved values in the buffer are written back to their former locations when the Java Card is re-inserted to CAD.

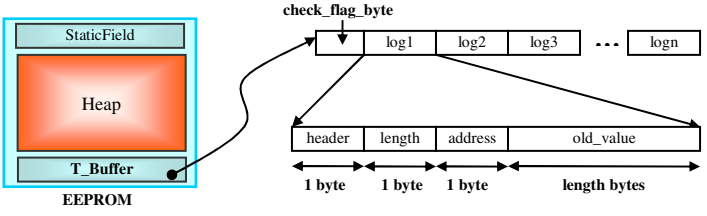


Fig. 3. The inner structure of T_buffer has a lot of logs and each log consists of 4 parts; header, length, address and old_value

Table 2 below shows the number of EEPROM writing per each area of whole EEPROM. T_buffer area writing is about 75 to 80 percent of total number. The reason why the writing number of this area is higher than other areas is a transaction mechanism of a traditional Java Card to guarantee an atomicity. In other words, The transaction mechanism protects data corruption against such events as power loss in the middle of a transaction. In a traditional Java Card, this transaction mechanism makes the Java Card more slow and inefficient. In this paper, we suggested new transaction mechanism using RAM, not EEPROM.

Table 2. The number of EEPROM writing per each area of whole EEPROM during the downloading and executing of each applet

EMV Applet		**Wallet Applet**	
EEPROM area	the number of writing	EEPROM area	the number of writing
StaticField	1,681	staticfield	752
Heap	1,659	Heap	1,121
T_buffer	**10,121**	**T_buffer**	**8,478**
Total	**13,461**	**Total**	**10,351**

3.2 A Traditional Java Card with One Page Buffer

In a general Java Card environment, one page-buffer in RAM is used to write data into EEPROM. the size of a page-buffer depends mainly on platforms. It is between 128 and 256 bytes. our chip with CalmCore16 MPU from SAMSUNG has 128 bytes page buffer that a Java Card can write up to 128 consecutive bytes to EEPROM at a time. Namely, a Java Card can write between 1 byte and 128 consecutive bytes with this page buffer into EEPROM. For example, If EEPROM addresses of objects that will be written by a Java Card are sequentially 0x86005 and 0x86000, although both addresses are within 128 bytes, Java Card will first writes one object data in 0x86005 through the page-buffer, and then, after the page-buffer is clear, another object data will be written in 0x86000.

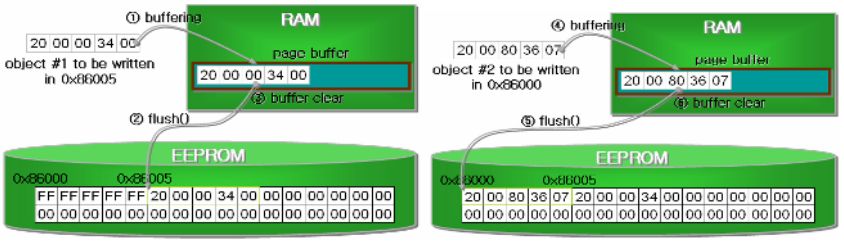


Fig. 4. how to write objects to EEPROM of the traditional Java Card using an inefficient page-buffer algorithm

Above figure 4 shows the page-buffer algorithm of a traditional Java Card. this page-buffer is just to write consecutive data to EEPROM. It dose not have the function for caching. When an applet is executed on Java Card, if the information such as objects and class data that the applet writes are close to each other, the total number of EEPROM writing would be reduced by adding a caching function to the page-buffer. first of all, to do this, the writing address of objects and data created by Java Card must have a high locality. It causes the number of EEPROM writing to reduce and also makes a hitting rate of caching function more high.

We investigated a general tendency of writing operation in accordance with the EEPROM address. we discovered the Java Card has internally a rule about the locality of EEPROM writing address. Consequently, a locality of Java Card objects and data is considerably high.

3.3 A High Locality of Heap Area in EEPROM

As mentioned earlier, a traditional Java Card System has only one page-buffer in RAM to write data into EEPROM. The page-buffer has a function for the buffering of just consecutive bytes. In this paper, we suggest the 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.

When the wallet class is created by install() method, the wallet object (2011C3A600000000) that have 3 fields is first written in EEPROM, and then, Own-

```
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

erPIN 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.

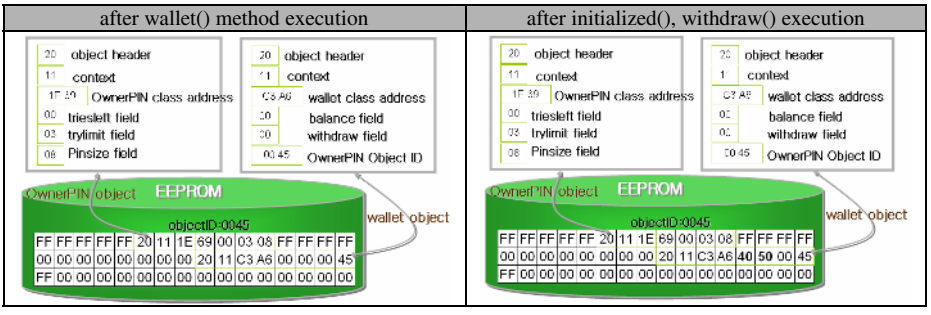


Fig. 6. The creation process of the wallet applet and the OwnerPIN object in EEPROM and the process of the changing localized-fields and rewriting them

Figure 5 and 6 showed several EEPROM 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 page-buffer above-mentioned, it might spends much time in writing and changing localized-data like above example.

4 Our Changed Java Card with a Fast Execution Speed

4.1 New Transaction Algorithm with T_Buffer in RAM

As mentioned in the related works, smart cards including a Java Card supports a transaction mechanism by saving old_values in EEPROM. the number of EEPROM writing in order to support the transaction is about 75 to 80 percent of the total number of EEPROM writing. EEPROM writing is typically more than 1,000 times slower than writing to RAM. It makes also Java Card much more slow and inefficient.

We suggested new transaction mechanism using RAM, not EEPROM in this paper. If such tearing such as power loss happens in the middle of a transaction, all data after transaction began should be ignored. If T_Buffer area to save old_values places in RAM, in case of power loss, RAM is automatically reset. It means the preservation of old_values.

Figure 7 shows the transaction mechanism of a traditional Java Card. After a transaction begin, if tearing such as power loss occurs, Java Card restore data involved in the transaction to their pretransaction(original) values the next time the card is powered on. To do this, Java Card must store all old_values in T_Buffer in EEPROM whenever Java Card writes some data in EEPROM.

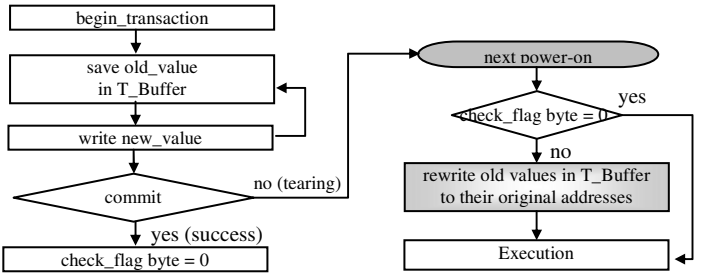


Fig. 7. The transaction mechanism with T_buffer in EEPROM of a traditional Java Card

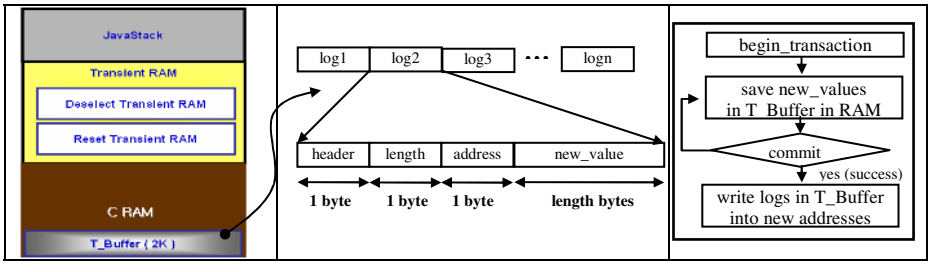


Fig. 8. RAM structure to support our changed transaction mechanism, the structure of our T_buffer and our transaction mechanism with T_buffer in RAM of a traditional Java Card

In this paper, we suggest that T_Buffer to support a transaction is in RAM in order to reduce EEPROM writings. Our T_buffer in RAM saves all new_values that will be written in EEPROM after a transaction began. Our T_Buffer also could have many logs until a transaction commit. Figure 8 below shows the structure of T_Buffer. Each log entry consists of four fields. The length field is the number of bytes of old data. The address field is original data in EEPROM. The last old_data field is old data bytes.

4.2 Our Object-Buffer Based on Java Card Objects with a High Locality

In chapter 3, we explained how to write data in EEPROM by using one page buffer in a traditional Java Card. It is the one of causes of a Java Card's performance drop in

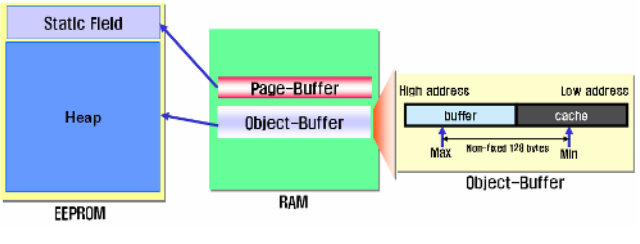


Fig. 9. The heap-buffer that is consisted in 2 part; the buffer and cache. The data between Min and Max can be written to EEPROM at a time.

company with the transaction mechanism of a traditional Java Card. We discovered that all objects and data that the Java Card creates during the execution has a high locality. It means that an additional caching function makes the number of EEPROM writing go down. For these reasons, we developed new Java Card with two page buffer in RAM; one is the existing page buffer for non-heap area, another (object-buffer) is for heap area in EEPROM. The heap area is where objects created by Java Card are allocated.

Figure 10 below shows the main algorithm using the object-buffer and page-buffer The writing of non heap-area is performed with the existing page buffer. The writing of heap-area is executed with the object-buffer. When the Java Card writes data related to Java Card objects into heap area of EEPROM, the first operation is to get 128 bytes lower than the address that will be written and to copy them to the cache area of the object-buffer. Next, the buffer area(128-byte) of the object-buffer is cleared. Two points, Max and Min have the highest and lowest points that are written after Java Card get new 256 bytes to the object-buffer. the gap between them continually is checked in order to write the heap buffer to EEPROM. Max and Min are non-fixed points to raise the efficiency of the heap buffer. The reason why the gap between Max and Min is 128 bytes is that our target chip, CalmCore16, supports the EEPROM writing of 128 bytes at a once.

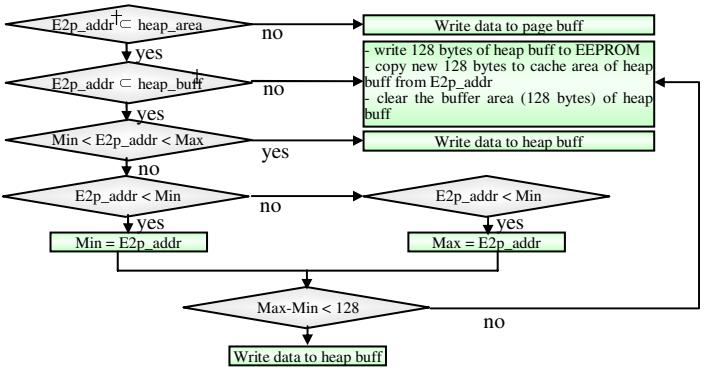


Fig. 10. The object-buffer algorithm that checks continually the Min and Max points to write the object-buffer to EEPROM when Java Card writes data to heap area. (†E2p_addr : the EEPROM address that data will be written, ‡ heap_buff(object-buff) : our new heap buffer with caching and buffering function for just heap area in EEPROM).

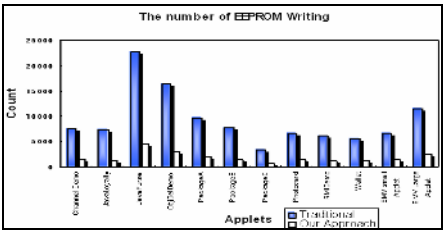
5 Evaluation of Our Approach

The key of our approach is improve an execution speed of the Java Card by reducing the number of EEPROM writing. The main idea is also that EEPROM writes are typically more than 1,000 times slower than writes to RAM. One of the analyzed results of a traditional Java Card is that Java Card has a inefficient transaction mechanism to guarantee an atomicity and page-buffer algorithm to write data to EEPROM regardless of the high locality of Java objects. For this reason, we developed new transaction mechanism and new page buffer algorithm.

In our approach, to get more precise figure in the real Java Card, we made an experiment with CalmCore16 MCU [14], SAMSUNG MicroController for smart card.

Figure 11 below shows the comparison between a traditional Java Card and our changed Java Card in regard to the number of EEPROM writing and the execution speed. First of all, the number of EEPROM writing is reduced by about 80% by using the T_Buffer and the object buffer in RAM.

Applets	Traditional	Our Approach	Reduced
ChannelDemo	7552	1586	79%
JavaLoyalty	7291	1322	82%
JavaPulse	22712	4537	80%
ObjDelDemo	16416	3025	82%
PackageA	9685	2000	79%
PackageB	7698	1406	82%
PackageC	3439	745	79%
PhotoCard	6737	1400	79%
RMIDemo	6119	1261	79%
Wallet	5641	1190	79%
EMV small Applet	6721	1419	79%
EMV Large Applet	11461	2433	79%
Aerage	**80%**		



Applets	Traditional	Our Approach	Reduced
ChannelDemo	76140	49438	35%
JavaLoyalty	72703	46187	36%
JavaPulse	232100	150359	35%
ObjDelDemo	159420	99157	38%
PackageA	90530	56375	38%
PackageB	74859	49937	33%
PackageC	32743	20907	36%
PhotoCard	64608	41407	37%
RMIDemo	57328	36235	34%
Wallet	57140	37438	37%
EMV small Applet	61766	38859	37%
EMV Large Applet	119812	79422	34%
Aerage	**36%**		

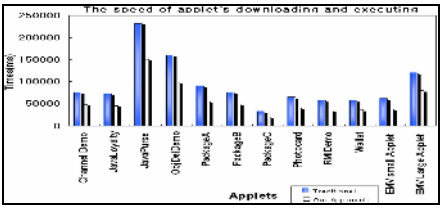


Fig. 11. The comparison between a traditional Java Card and our changed Java Card with regard to the number of EEPROM writing and the execution speed

Components	Traditional	Our Approach	Differnce
Initialize	1485	1688	-203
Select Install	6281	3812	2469
CAP Begin	1234	485	749
Header	3562	2156	1406
Directory	3969	2344	1625
Import	2875	1640	1235
Applet	3250	1922	1328
Class	2203	1484	719
Method	11266	8641	2625
StaticField	2297	1469	828
ConstantPool	6781	4984	1797
ReferenceLocation	9141	4719	4422
CAP End	625	422	203
Create Applet	2171	1672	499
Total	57140	37438	19702

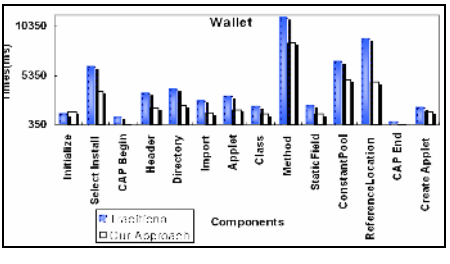


Fig. 12. The comparison between a traditional Java Card and our changed Java Card in regard to Wallet applet's downloading and execution speed per each component

One applet consists of over 11 components that include all information of one applet package. We also produced downloading results about each component. Basically, when Java Card installer downloads one applet, the component that takes a long time is the referencelocation component. The reason is that both are related to the resolution of indirect references during the downloading. our approach almost reduced the downloading time of the referencelocation by 50%.

6 Conclusion and Future Work

Java Card technology is already a standard for smart cards and SIM cards [11, 15]. A Java language is basically slower than other languages. The card platforms also have a heavy hardware limitation. In spite of a Java's slow speed, 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 EEPROM writing with new robust transaction mechanism and new object-buffer based on the high locality of Java Card objects. It also makes Java Card more fast. With our approach, the number of EEPROM writing and the downloading speed reduced by 80% and 35% 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). This technology will be applied to embedded systems such as KVM, PJAVA, CLDC with a Java Technology.

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) 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) 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.
12. 10. http://www.gemplus.com. : OTA White Paper. Gemplus (2002).
13. the 3rd Generation Partnership Project. : Technical Specification Group Terminals Security Mechanisms for the (U)SIM application toolkit. 3GPP (2002).
14. MCULAND, http://mculand.com/e/sub1/s1main.htm.
15. X. Leroy. Bytecode verification for Java smart card. Software Practice & Experience, 2002 319-340
16. SAMSUNG, http://www.samsung.com/Products/Semiconductor
17. SIMAlliance, http://www.simalliance.org.