스마트 카드기반의 자바카드 가상기계 최적화연구

황욱철*°, 양윤심*, 권오형*, 최원호*, 김도우*, 정민수*
*경남대학교 컴퓨터공학과
**한국전자통신연구원

A Study On the Optimization of the Java Card Virtual Machine Based SmartCard

Wook-Chul Hwang °, Yoon-Sin Yang, Oh-Hyung Kweon, Won-Ho Choi , Do-Woo Kim, Min-Soo Jung
*Dept. of Computer Engineering, Kyungnam University.
**Electronics and Telecommunications Research Institute

요 약

Java Card 플랫폼을 내장한 스마트 카드는 현재의 스마트 카드에 적용되는 모든 표준을 따르는 전형적인 스마트 카드인데, 자바카드 플랫폼의 장점을 최대한 이용하기 위해 적용할 기술은 메모리 측면에서 보면 사용할 수 있는 자원이 적어 다양한 용도로 사용에 있어 제한적이다. 따라서 본 논문에서는 적은 메모리 자원을 가지고 보다 효율적이고 최적화된 성능을 가지는 자바 플랫폼을 지원하기 위해 자바 플랫폼의 성능에 핵심이 되는 자바 카드 가상기계(JCVM)의 성능 최적화 방안에 대한 방법을 제시하고자 한다.

1. 서론

현재 스마트 카드는 통신, 금융, 교통, 신분 확인, 전자 화폐 등의 여러 응용 서비스에서 널리 사용되고 있으며 점차로 그 용도가 확대되어 금융면에서는 기본의 신용 카드를 대체하고 다양한 형태의 장비를 통하여 네트워크와 연결되어 사용되는 추세로 바뀌고있다. 또한, 이와 더불어 빈번한 사용에 따라 개인 정보의 안전한 저장 및 사용, 사용자 인증을 포함한 정보 보안의 문제도 크게 부각되고 있다. 이러한 조류에 발맞추어 스마트 카드의 하드웨어도 발전하고 있다. CPU는 기존의 8bit 위주에서 보다 처리 성능이 뛰어난 32bit로 전환 되고, 데이터의 암.복호화를 빠르게 하기 위해 특정 암호 기법에 대한 전용 보조프로세서의 사용이 늘고 있다. 또, 다양한 응용 서비스용 프로그램들을 수용하기 위한 저장 공간과 수행 시 필요한 임시 사용 공간으로 ROM, EEPROM 및 RAM의 메모리 크기를 늘이고 있는 실정이다.

Java Card 플랫폼을 내장한 스마트 카드는 현재의 스마트 카드는 현재의 스마트 카드에 적용되는 모든 표준을 따르는 전형적인 스마트 카드인데 하위의 운영체제 위에 존재하는 자바카드 가상기계(JCVM,

Java Card Virtual Machine)가 자바카드 애플릿의 바이트 코드(bytecode)를 수행하고 메모리, I/O 같은 스마트 카드 내의 모든 자원에 대한 접근을 제어한다는 점에서 차이가 난다. 자바카드 플랫폼의 이러한 장점을 최대한 이용하기 위해 적용할 기술은 메모리 측면에서 보면 사용할 수 있는 자원이 적어 다양한 용도로의 사용에 있어 제한적이다. 따라서 본 논문에서는 적은 메모리 자원을 가지고 보다 효율적이고 최적화된 성능을 가지는 자바 플랫폼을 지원하기 위해 자바 플랫폼의 성능에 핵심이 되는 자바 카드 가상기계(JCVM)의 성능 최적화 방안에 대한 방법을 제시하고자 한다.

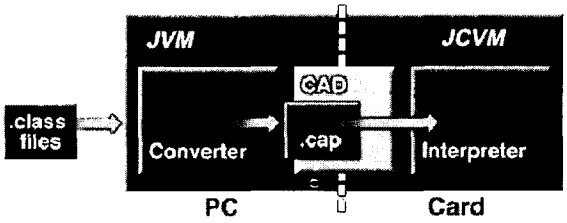
본 논문의 이후 구성은 다음과 같다. 2장에서는 자바 카드와 관련한 연구로서 기본적인 자바 카드 기술과 카드상에서 실제 수행되는 바이트 코드인 CAP(Converted Applet Program)의 구조에 대하여 살펴보겠다. 그리고 3장에서는 자바카드 가상기계의 최적화 방법에 대하여 기술하고, 마지막으로 4장에서는 실험의 결과, 5장에서 결론과 향후 연구방향에 대한 설명으로 마무리 하도록 하겠다.

2. 관련연구

2.1 자바카드 가상기계 (Java Card Virtual Machine)

Java Card 바이트코드를 수행하는 수행 엔진인 JCVM은 JVM에 비해 스마트 카드에 적합하도록 최적화되어 applet간의 방화벽, 최적화된 명령어 등 subset의 개념이라기보다는 지원 내용은 작지만 별도의 새로운 수행환경을 구성한다. JCVM과 기존의 JVM(Java Virtual Machine)과의 차이점은 Off-Card VM과 On-Card VM으로 나뉘는 분할 가상기계로 이루어지며, [2,3] On- Card VM은 좁은 의미로는 수행 엔진인 인터프리터를 지칭하며 넓은 의미로는 프래임워크(framework)가 중심이 되는 system class API와 인터프리터, 메모리 관리 루틴, 예외 처리 루틴 및 운영체제와의 인터페이스(interface) 등을 포함하는 자바 카드 수행 환경(Java Card Runtime Environment, JCRE)을 의미 한다. 분리된 JCVM의 구성은 아래 그림 1에서 보이며 각 기능을 간략히 기술하면 다음과 같다.

- Off-Card VM
  - Java class loading, linking, name resolution 및 package 변환
  - Verification(JCVM 사양에 대한 체크 및 resolution 결과에 대한 체크 포함)
  - 바이트코드의 최적화
  - CAP File과 Export File 생성
- On-Card VM
  - 바이트코드 수행 및 수행시 메모리 관리(stack-machine을 에뮬레이트함)
  - 방화벽 루틴을 수행



[그림 1] Java Card Virtual Machine의 구성

2.2 CAP 파일

CAP 파일은 자바 컴파일러에 의해 생성된 클래스 파일을 컨버터(conveter)에 의해 자바카드에서 사용할 수 있는 형태로 변형한 파일로서, 스마트 카드에서 메모리 제한 및 로드(load)시의 안정성을 고려하여 자바 플랫폼에서 post-issuance를 위해 사용하는 파일의 형식으로 패키지 단위의 실행 가능한 바이너리 표현을 가지고 있으며 Jar container 형식내의 컴포넌트별로 분할되고, 최적화된 자바 바이트 코드를 소유할 뿐만 아니라 내부 constant pool을 통한 resolution 경로를 소유한다.[2,3,5]CAP 파일은 클래스, 메소드, 정적변수 등의 분리된 11개의 컴포넌트들로 구성되며 기존의 클래스 파일의 스트링 기반의 링킹(linking)이 아닌 컨버터에 의해 지정되는 토큰(token)과 오프셋(offset)에 의해 CAP 파일 내 바이트 코드의 내외부의 참조에 대한 링킹(linking)을 수행한다. CAP 파일의 컴포넌트들을 기능 별로 분류해

보면 아래와 같이 분류 할 수 있다.

□Information Components : Header, Diretory

□Core Components : Class, Method, StaticField

□Link Components : Import, Constant Pool, ReferencedLocation

□Other Components : Applet, Export, Descriptor

2.3 최적화 기술

Loop 최적화

루프 최적화는 불변 루프의 확인과 자바 프로그램의 성능개선을 위하여 그러한 것들을 루프의 밖으로 옮기는 것에 초점을 맞춘다. 아래의 예에서, 루프 조건 $k < A[0] + 3.2 - (2 * A[n-1])$ 와 배정문 $z = x - 4 * n$ 불변이고, for 루프의 밖으로 옮겨 질 것이다.[9].

```
void foo(float[] A) {
    float x = 3.5f;
    int n = A.length;
    float z;
    for( int k = 0; k < A[0] + 3.2 - (2 * A[n-1]);
                    k++) {
        float y = k * 3;
        z = x - 4 * n;
        System.out.println(k + " " + y);
    }
}
```

자바 바이트 코드의 압축

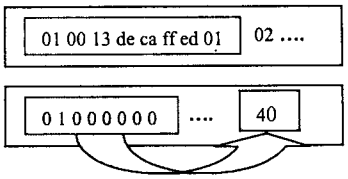
3. 자바 바이트 코드 최적화 알고리즘

최적화 알고리즘의 접근은 CAP 파일 내에서 00 의 위치 정보를 다시 표현하는 맵(Map) 파일을 만드는 것 이다. 왜냐하면 CAP 파일 내에서 00 은 패딩(padding) 요소이기 때문이다.

3.1 맵(Map) 파일 생성

00 의 위치 정보를 저장하기위한 맵 파일을 생성하기위해서 맵 파일 생성기는 입력으로 CAP 파일을 읽는다. 맵 파일 생성기의 기본적은 처리는 아래와 같다 :

```
Loop ( End of CAP file )
        read 8byte in CAP file
        8 byte is assigned to Temp[ ] variable
        for each byte in Temp[ ]
            if ( each byte == 0 )
                write location information in map file
```



이러한 처리 이후에 두개의 파일이 생성되는데, 하나는 CAP 파일에서 00 스트림을 제거한 파일이고, 또 다른 하나는 00 의 위치정보를 포함하고있는 맵 파일이다.

```
01 00 13 de ca ff ed 01 02 02 00 01 09 a0 00 00
00 62 03 01 0c 01
02 00 1f 00 13 00 1f 00 00 00 0b 00 36 00 0c 00
67 00 0a 00 13 00 09 00 6c 00 00 00 00 00 00 01
00 00
04 00 0b 01 00 01 07 a0 00 00 00 62 01 01
05 00 36 00 0d 02 00 00 00 06 80 03 00 03 80 03
01 01 00 00 00 06 00 00 01 03 80 0a 01 03 80 0a
                        :
0b 00 6c 01 00 01 00 00 00 00 01 00 03 00 02 00
00 00 00 1c 00 84 00 01 00 1e 00 11 00 00 00 00
01 09 00 14 00 30 00 09 00 00 00 00 07 01 00 1f
00 33 00 46 00 00 00 00 00 0d 00 1c 00 1e 00 1e
ff ff 00 1e 00 1c 00 20 00 20 00 22 00 24 00 27
```

[그림 3.1] HellowWorld.CAP

```
01 13 de ca ff ed 01 02 02 01
09 a0 62 03 01 0c 01 02 1f 00
13 1f 0b 36 0c 67 0a 13 09 6c
01 04 0b 01 01 07 a0 62 01 01
05 36 0d 02 06 80 03 03 80 03
01 01 06 01 03 80 0a 01 03 80
0a06 03 80 0a 07 03 80 0a 09
03 80 0a 04 03 80 0a 05 06 80
10 02 03 80 0a 03
          :
          :
          :
0b 6c 01 01 01 03 02 1c 84
01 1e 11 01 09 14 30 09
07 01 1f 33 46 0d 1c 1e 1e ff ff
1e 1c 20 20 22 24 27 2a 2e
01 b0 01 10 01 40 02 41 03 44
10 04b4 41 06 b4 b4 44 02 44
04 b4 31 06 68 a1
```

```
40 23 81 55 55
5d f9 43 81 24
70 a7 10 76 00
00 00 00 51 1c
a1 08 90 44 04
20 04 20 02 02
01 04 42 2f fd
42 02 5e af 55
79 57 95 7d 51
55 55 00 00 00
30
```

[그림3.2] HelloWorld.out and HellowWorld.map

4. 수행결과

본 논문의 최적화 알고리즘의 효과를 테스트하기 위해서 아래의 표 1에서 보여지는 것처럼 많은 CAP 파일에 우리의 알고리즘을 적용해 보았고 그 결과로

각 CAP 파일이 원래 크기가 평균 85%로 감소됨을 알 수 있었다. 즉 실제 약 10% 정도의 크기 감소 효과를 볼 수 있었다.

No.	File Name	File Size			
	(Cap files)	Original	Delete Zero Byte	Zero Map	Difference Size
		*.cap	*.out	*.map	*.cap-(*.out+*.map)
1	HelloWorld.cap	2337	1779	293	265
2	JavaLoyalty.cap	2575	1948	322	305
3	JavaPurse.cap	6431	5244	804	383
4	NullApp.cap	2155	1616	270	269
5	SampleLibrary.cap	2138	1629	268	241
6	wallet.cap	2763	2098	346	319
	Average Byte	3066.5	2385.67	383.83	297
	Size Effective Ratio				10%↓

[표 1] CAP 파일

이러한 이유로 이 알고리즘이 메모리 제한적인 장치를 위한 자바 바이트 코드 최적화에 적용할 수 있음을 증명할 수 있다. 그러나 클래스 로더(loader)는 생성된 output 파일과 맵 파일을 다시 합성하기위한 오버헤드(overhead)를 가진다.

5. 결론 및 향후 연구 방향

위에서 서술한 바와 같이 메모리 제한적인 장치에서 바이트 코드 최적화 알고리즘은 원래 바이트 코드의 사이즈를 10~15%정도 까지 줄일 수 있었다. 그러나 실행시간 오버헤드 문제와, 생성된 파일을 실행하기 위해서는 현재의 자바카드 가상기계를 수정해야 된다는 문제점을 가지고 있다.

본 논문에서 소개한 최적화의 방향은 카드의 실제 바이트 코드가 되는 CAP 파일에 대한 최적화이다. 즉 Off-Card VM(Virtual Machine)에서의 최적화를 의미 하는 것이다. 실제 스마트 카드에 적용을 위해서는 실제 CAP 파일을 카드로 로드 시키는 역할을 하는 Off-Card Installer와 On-Card Installer의 수정이 선행 되어야 할 것이다.

[참고 문헌]

[1] T. Lindholm and F. Yellin, *The Java™ Virtual Machine Specification* ADDISON-WESLEY, 1997.

[2] Sun Microsystems, Inc., The *Java Card™ 2.1.1Virtual Machine Specification,* SUN, 2000

[3] Sun Microsystems, Inc., The *Java Card™ 2.1.1Runtimel Environmen(JCRE)t Specification,* SUN, 2000

[4] A. Taivalsaari, *Implementation a Java Virtual Machine in the java programming Language*, SUN Lab, 1997.

[5] B. Venners, *Inside the Java Virtual Machine*, McGraw-Hill, 1997.

[6] Min-Soo Jung, Jong-Dong Lee, "Design and Implementation of Call Graph Viewer for Java", The 25th KISS Spring Conference, pp74~76, 1998.

Dong-Hang Ryu, Min-Soo Jung, "Design