# 자바카드 플랫폼상에서 자바 클래스 파일의 최적화 연구

김도우[†] · 정민수[††]

**요 약**

자바카드 기술은 스마트카드나 메모리 제한적인 장치에서 자바 프로그래밍 언어로 작성된 응용프로그램을 동작 가능하게 한다. 자바카드 기술은 높은 안전성, 이식성, 다중의 응용프로그램을 관리하고 저장하는 기능을 제공한다. 그러나 자바카드 플랫폼의 제한적인 메모리 자원은 다양한 용도로 자바카드가 보급되는데 저해 요인으로 작용하고 있다. 따라서 본 논문에서는 자바카드의 효율적인 메모리 사용을 위해서 바이트코드 최적화 알고리즘을 제안한다. 이 알고리즘은 예외처리 구문 try-catch-finally에서 catch절의 매개변수에 대한 기억장소를 공유하게 함으로써 생성되는 바이트코드의 크기를 줄일 수 있다.

# A Study On The Optimization of Java Class File under Java Card Platform

Do-Woo Kim[†] and Min-Soo Jung[††]

**ABSTRACT**

Java Card technology allows us to run Java applications on smart cards and other memory-constrained devices. Java Card technology supports high security, portability and ability of storing and managing multiple applications. However, constrained memory resources of the Java Card Platform hinder wide deployment of the Java Card applications. Therefore, in this paper we propose a bytecode optimization algorithm to use the memory of a Java Card efficiently. Our algorithm can reduce the size of the bytecode by sharing the memory of the parameters of the catch clause in the try-catch-finally sentence.

**Key words:** 자바카드, 바이트코드, 최적화

## 1. 서 론

자바카드 플랫폼을 내장한 스마트카드는 현재의 스마트카드에 적용되는 모든 표준을 따르는 전형적인 카드이다. 이 카드는 하위의 운영체제 위에 존재하는 자바카드 가상기계(Java Card Virtual Machine : JCVM)가 자바카드 애플릿의 바이트코드(byte-code)를 수행하고, 메모리, I/O 같은 스마트카드 내의 모든 자원에 대한 접근을 제어한다는 점에서 차이가 난다. 자바카드 기술은 플랫폼간에 이진 코드의 이식성(portability) 즉, 상호운용성(inter-operability)이 뛰어나고 타입검사 등에 의해 악의적 코드에 대한 보안성을 지닌 자바언어를 스마트카드의 실시간 환경에 대해서 최적화하고 있다. 스마트카드에서 가상기계 이용으로 인한 장점은 응용프로그램과 운영체제를 분리하는 개방형(open-platform) 운영체제를 갖는 것이다. 이것은 하드웨어 의존적인 어셈블리 코드가 아닌 상위언어인 자바언어로 쉽게 응용프로그램을 작성 및 수행할 수 있게 하고, 카드가 최종 사용자에게 발급된 이후에도 필요한 응용서비스에 따른 응용프로그램을 스마트카드에 적재(post-issuance)

본 연구는 경남대학교 학술 논문게재 연구비 지원으로 이루어졌음.
접수일 : 2003년 3월 25일, 완료일 : 2003년 5월 13일
[†] 경남대학교 컴퓨터공학과 박사과정
[††] 종신회원, 경남대학교 정보통신학부 교수

도 가능하게 한다. 이는 다수의 다양한 응용 프로그램을 수용할 수 있는 유연성(flexibility)을 가지게 한다. 또한 자바카드에서는 자바언어 자체의 보안 특성 이외에 응용프로그램간의 방화벽을 제공함으로써 엄격한 보안성을 보장한다[1,2].

자바카드 플랫폼의 메모리 측면을 살펴보면 사용할 수 있는 자원이 적어 다양한 용도로 사용하는 것은 제한적일 수 밖에 없다. 따라서, 본 논문에서는 이러한 문제점을 해결하기 위해서 적은 메모리 자원을 가지고 보다 효율적이고 최적화된 성능을 가지는 자바카드 플랫폼을 지원하기 위해서 자바카드 플랫폼의 성능에 핵심이 되는 효율적인 바이트코드 생성을 통해서 자바카드 가상기계의 성능 개선에 대한 방안을 제시하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 자바카드 및 가상기계 내에서 메소드 수행과 관련한 연구를 기술하고, 3장에서는 자바카드 플랫폼에서 효율적인 바이트코드 생성을 위한 최적화 알고리즘의 설계에 관해 기술한다. 4장에서는 최적화 알고리즘의 구현에 관하여 기술하고, 마지막으로 5장에서는 결론과 연구 활용 방안에 대해 기술한다.

## 2. 관련 연구

### 2.1 자바카드

자바카드 기술은 자바언어로 쓰여진 프로그램이 스마트카드나 혹은 그 밖에 제한적인 자원을 가진 장치에서 동작을 가능하게 한다. 자바카드 기술의 설계는 스마트카드에 자바 시스템 소프트웨어를 구축하는 것이다. 그리고 해법은 자바언어의 특징을 부분적으로 지원하고, 분할모델을 적용할 수 있는 자바가상기계(Java Virtual Machine)를 구현하는 것이다.

자바카드 바이트코드를 수행하는 자바카드 가상기계는 자바가상기계에 비해서 스마트카드에 적합하게 최적화되어 있다. 그림 1과 같은 자바카드 가상기계가 기존의 가상기계와 가장 큰 차이점은 자바카드 가상기계가 오프-카드(Off-Card) 가상기계와 온-카드(On-Card) 가상기계로 나뉘는 분할 가상기계로 이루어진다는 것이다. 기존의 가상기계에서 실행 시점에 처리되는 부분 중에서 클래스 적재(class loading), 바이트코드 검증(bytecode verification), 클래스 링킹(linking)과 해석(resolution) 부분은 자원에 제약을 받지 않는 오프-카드에서 처리된다[3-6].

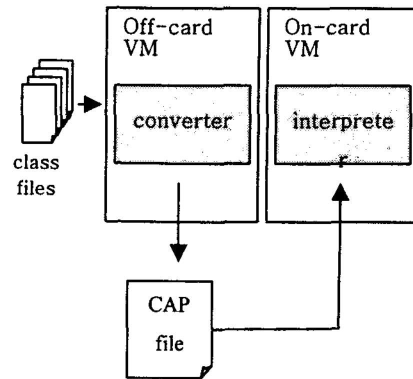



그림 1. 자바카드 가상기계의 구성

자바카드 가상기계의 좁은 의미는 수행 엔진인 인터프리터를 지칭하며 넓은 의미로는 시스템 클래스 API와 인터프리터, 메모리 관리, 예외처리 및 운영체제와 인터페이스 등을 포함하는 자바카드 수행환경(Java Card Runtime Environment : JCRE)을 의미한다.

변환기(Converter)는 번역을 통해 생성된 클래스 파일을 온-카드 가상기계 상에서 수행 가능한 이진 파일 형식인 CAP 파일과 링크 및 검증을 위한 인터페이스 이진파일 형식인 EXP 파일을 생성하는 오프-카드 가상기계의 한 부분이다. 자바가상기계는 한 번에 하나의 클래스를 처리하지만 변환기의 변환 단위는 패키지이다. 변환기는 자바가상기계가 수행하는 작업 중에서 클래스를 적재하는 작업을 담당한다[5,6].

자바카드 인터프리터는 바이트코드 명령을 수행함으로써 궁극적으로 자바카드 애플릿을 실행하고, 메모리 할당을 관리하며 객체를 생성하는 역할을 한다.

### 2.2 가상기계의 메소드 수행

자바가상기계가 자바 메소드를 호출할 때 메소드 실행에 필요한 지역변수 개수와 오퍼랜드 스택의 크기를 결정하기 위하여 클래스 데이터를 검사한다. 클래스 데이터 크기는 번역 시에 결정되고 각 메소드에 대한 자료가 클래스 파일에 포함된다. 자바가상기계는 메소드에 대한 적당한 크기의 스택 프레임을 생성하고 자바스택에 스택 프레임을 넣는다.

자바스택 프레임의 지역 변수부는 워드들의 배열

로 구성되는데 메소드의 매개변수와 지역변수들을 포함한다. 컴파일러는 선언된 순서에 따라 지역변수 배열에 매개변수와 지역변수들을 차례로 배치한다. 단, 그림 2에 나타난 바와 같이 정적 메소드와 인스턴스 메소드의 선언 순서가 다르다[7,8].

인스턴스 메소드의 경우는 지역 변수부의 첫번째 요소로 this 객체에 대한 참조를 포함한다. 그 다음은 선언된 순서에 따라 매개변수와 지역변수를 차례로 배치한다. 정적 메소드의 경우는 선언된 순서에 따라 매개변수와 지역변수만 차례로 배치한다.

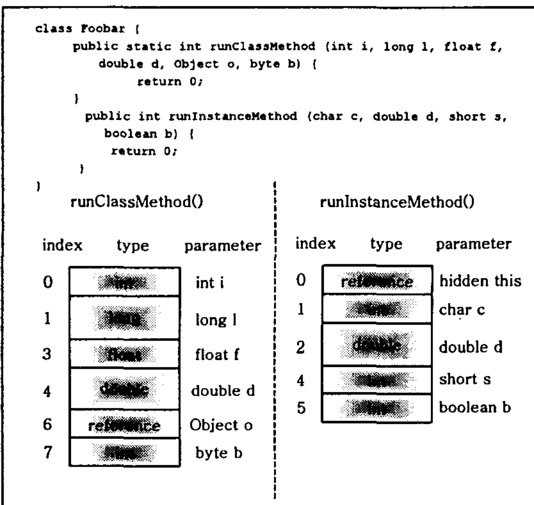



그림 2. 메소드 호출 시 자바 스택 내의 지역 변수부

## 3. 바이트코드 최적화 설계

### 3.1 바이트코드 최적화

#### 3.1.1 예외처리

예외는 메소드의 호출과 실행, 부정확한 데이터, 그리고 시스템 오류 등 다양한 상황에서 야기된다. 이러한 예외는 검사되고 적절히 처리되어야 한다. 그렇지 않으면 프로그램의 비정상적인 종료 뿐만 아니라 잘못된 실행 결과를 초래할 수도 있다.

자바는 예외를 체계적으로 검사하고 처리할 수 있는 방법을 제공한다. 예외처리 방법은 기대되지 않은 상황에 대해서 해당하는 예외를 발생시키고, 야기된 예외를 적절히 처리할 수 있는 예외 처리기를 프로그래머가 직접 작성하는 것이다. 이와 같이 예외처리를 위한 방법을 언어 시스템에서 제공함으로써 얻을 수 있는 장점은 응용프로그램의 신뢰성을 높일 수 있다는 것과 가능한 한 기계 의존성을 벗어나 예외적인 경우도 프로그래머가 제어를 계속 유지할 수 있다는 것이다.

자바에서 예외를 검사하고 처리하는데 사용하는 구문은 try-catch-finally이며 그 형태는 그림 3과 같다.

그림 3의 try블록 안에서 예외가 검사된다. 만약 예외가 발생하면 해당하는 catch블록에서 예외가 처리된다. 따라서 catch블록을 예외 처리기라고 부른다.

자바의 클래스를 정의하는데 23.3%~24.5%가 try-catch절을 포함하고 있고, 메소드 정의도 16%정도가 예외처리기를 가지고 있다[9].

예외 처리기는 단순히 오류에 대한 메시지를 출력하고 프로그램을 정상 종료하는 기능을 한다. 그 이유는 첫째로 발생한 예외를 처리한 후에 그 시점까지의 실행 결과를 신뢰할 수 없고, 둘째로 예외 발생 이전의 상태로 되돌려서 실행을 재개하는 것은 많은 오버헤드를 요구하기 때문이다. 따라서 대부분의 자바 예외처리기도 예외객체 형에 대한 정보출력과 정상 종료하는 수준이다.

자바카드 플랫폼의 경우도 자바의 경우와 크게 다르지 않다. 카드 상의 호출된 일반적인 메소드 및 원격 메소드는 예기치 않은 조건들로 인해서 예외를 발생시킬 수 있다. 카드 상의 RMIService 클래스는 발생된 예외를 포착(catch)하고, 클라이언트로 이 정보를 반환한다. 원격 메소드 호출 동안 발생하는 예외와는 별도로 통신상의 오류, 마샬링(marshalling), 프로토콜, 언마샬링(unmarshalling) 등 RMI 메소드 호출과 관련한 예외들은 RemoteException 예외의 결과로 클라이언트로 전파된다. 카드 상에서 발생한

```
try {
      //... ... ...
  } catch ( ExceptionType identifier )
  {   //... ... ...
  } catch ( ExceptionType identifier )
  {   //... ... ...
  } finally {
      //... ... ...
  }
```

그림 3. 자바언어의 예외처리 구문

예외가 자바카드 API에 정의된 예외이면, 예외는 스터브 객체를 통해서 클라이언트로 전파된다[6].

일반적으로 자바 컴파일러는 catch절의 매개변수에 대해서 각각 독립된 변수로 취급하여 메소드 호출 시 기억장소를 할당한다. 하지만 이는 생성된 바이트코드의 크기를 증가시키고, 실행하는데 더 많은 기억장소를 요구한다. 따라서 예외처리 구문 try-catch-finally의 catch절의 매개변수를 공유함으로써 생성되는 바이트코드의 크기와 실행 시 필요한 메모리의 크기를 줄일 수 있다.

예를 들어, 그림 4의 원시 프로그램은 자바카드를 사용하여 예금업무를 수행하는 클래스를 정의하고 있다. 그림 5는 PurseImpl 클래스의 credit() 메소드

```
import javacard.framework.UserException;
import javacard.framework.CardRemoteObject;
import java.rmi.RemoteException;
public class PurseImpl extends
      CardRemoteObject implements Purse {
  private short balance = 0;
  public void credit(short m) throws
          RemoteException, UserException {
    try
    {
      if(m<=0)
        UserException.throwIt(BAD_ARGUMENT);
      if((short)(balance+m) > MAX_AMOUNT)
        UserException.throwIt(OVERFLOW);
      balance +=m;
    }
    catch(RuntimeException runtimeexception)
    {
        throw runtimeexception;
    }
    catch(UserException userexception)
    {
        throw userexception;
    }
    catch(Exception exception)
    {
        throw new UnexpectedException();
    }
  }
}
```

그림 4. PurseImpl 클래스에 대한 자바카드 원시 프로그램

```
Method void credit(short)
0 iload_1
1 ifgt 10
4 sipush 24578
7 invokestatic #4
            <Method void throwIt(short)>
10 aload_0
11 getfield #2 <Field short balance>
14 iload_1
15 iadd
16 i2s
17 sipush 400
20 if_icmple 29
23 sipush 24577
26 invokestatic #4
            <Method void throwIt(short)>
29 aload_0
30 dup
31 getfield #2 <Field short balance>
34 iload_1
35 iadd
36 i2s
37 putfield #2 <Field short balance>
40 goto 54
43 astore_2
44 aload_2
45 athrow
46 astore_3
47 aload_3
48 athrow
49 astore 4
51 aload 4
53 athrow
54 return
Exception table:
 from to target type
  0 40 43 <java.lang.RuntimeException>
  0 40 46 <javacard.framework.UserException>
  0 40 49 <java.lang.Exception>
```

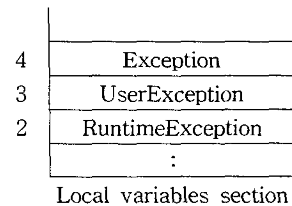



그림 5. PurseImpl 클래스의 credit() 메소드에 대한 자바 바이트코드

```
Method void credit(short)
0 iload_1
1 ifgt 10
4 sipush 24578
7 invokestatic #4
            <Method void throwIt(short)>
10 aload_0
11 getfield #2 <Field short balance>
14 iload_1
15 iadd
16 i2s
17 sipush 400
20 if_icmple 29
23 sipush 24577
26 invokestatic #4
            <Method void throwIt(short)>
29 aload_0
30 dup
31 getfield #2 <Field short balance>
34 iload_1
35 iadd
36 i2s
37 putfield #2 <Field short balance>
40 goto 54
43 astore_2
44 aload_2
45 athrow
46 astore_2
47 aload_2
48 athrow
49 astore_2
50 aload_2
51 athrow
52 return
Exception table:
 from to target type
  0 40 43 <java.lang.RuntimeException>
  0 40 46 <javacard.framework.UserException>
  0 40 49 <java.lang.Exception>
```

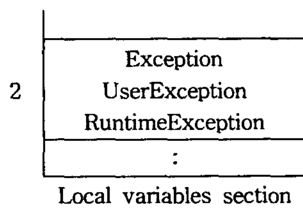



그림 6. 효율적으로 생성된 credit() 메소드의 자바 바이트 코드

에 해당하는 바이트코드다.

credit() 메소드의 바이트코드에서 pc 옵셋(offset) 0에서 37사이의 명령 수행 중에 예외가 발생하면, 자바가상기계는 발생한 예외를 스택에 넣고, catch절을 포함한 메소드와 연관된 예외 테이블을 검색한다. 가상기계는 테이블을 구성하는 예외의 순서에 따라서 차례로 검색하여 일치하는 예외를 찾으면, 프로그램 카운터를 새 pc 옵셋으로 설정하고, 예외처리를 계속한다. 일치하는 예외를 찾지 못하면, 자바가상기계는 자바 스택으로부터 현재의 스택 프레임을 꺼내고, 이 메소드를 호출한 메소드의 스택 프레임에서 이 예외를 재발생시킨다.

credit() 메소드의 경우 세 가지의 예외를 가지는데, 이들 모두 매개변수를 가진다. 실제, 예외 발생 시 catch절에 나열된 예외를 하나씩 검사하여 일치된 예외를 찾으면, 그에 해당되는 예외 처리기로 이동한다. 따라서 예외에 대한 매개변수를 각각 할당할 필요 없이 매개변수를 공유하도록 함으로써 그림 6과 같이 좀더 효율적인 바이트코드의 생성이 가능하다.

### 3.1.2 클래스 설계

최적화된 바이트코드 생성을 위해서는 번역 과정 중에 catch절의 매개변수에 대한 부분을 제어함으로써 가능하다. 인스턴스 메소드의 경우, 메소드의 지역변수의 크기를 결정하는 요소는 this객체, 매개변수와 지역변수다. 이들의 합이 지역 변수부의 크기가 된다. 메소드의 지역 변수부를 구성하는 LocalMember 클래스를 정의하여 메소드의 구문분석 단계에서 인스턴스 메소드의 경우, this 객체와 매개변수만을 지역 변수부의 구성요소로 먼저 추가한다. 정적 메소드의 경우는 매개변수만 지역 변수부의 구성요소로 먼저 추가한다. 구문분석 단계를 거치면서 그림 7과 같

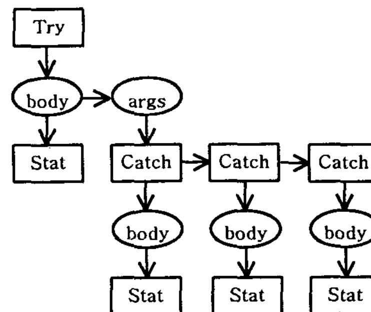



그림 7. try 문의 구문 트리

은 try문의 구문트리가 생성된다. 이 구문트리를 구성하는 TryStatement 클래스와 CatchStatement 클래스를 사용하여 메소드의 지역 변수부의 요소들을 검색한다.

## 4. 구 현

최적화된 바이트코드를 생성하기 위해서 자바 컴파일러의 클래스들에 추가된 필드와 메소드에 대해서 기술한다.

### 4.1 최적화된 바이트코드 생성

최적화된 바이트코드의 생성을 위해서 메소드 지역 변수부의 구성요소를 나타내는 LocalMember 클래스를 정의한다. 메소드의 구문분석 단계에서 인스턴스 메소드의 경우, this 객체와 매개변수만을 지역변수부의 구성요소 즉, LocalMember 객체로 먼저 등록한다.

그림 8의 LocalMember 클래스 정의에서 number 필드는 지역변수부의 색인(index), prev 필드는 이전의 지역 변수부의 구성요소를 의미한다. 그리고 modifiers 필드는 지역 변수의 접근 수정자, type은 지역변수의 형을 의미한다. 한편 name 필드는 지역변수 이름을 의미한다. 실제 number 필드의 값을 지역 변수부의 인덱스와 관련된 바이트코드 명령을 생성할 때 사용한다.

예외처리 구문 try-catch-finally에 대한 구문트리를 구성하는 TryStatement 클래스와 CatchStatement 클래스에 대한 정의는 다음과 같다.

그림 9의 TryStatement 클래스 정의에서 body 필드는 try 블록, args[] 필드는 catch 블록을 의미한다.

```
class LocalMember {
        int number;
        LocalMember prev;
        int modifiers;
        Type type;
        Identifier name;
                ...
        LocalMember()
}
```

그림 8. LocalMember 클래스 정의

```
class TryStatement {
        Statement body;
        Statement args[];
                ...
        public TryStatement()
        check()
        code()
}
```

그림 9. TryStatement 클래스 정의

그리고 check() 메소드는 body 필드 즉, 선언문, 대입문, 조건문, 반복문과 같은 try 블록을 구성하는 여러 문장들의 check() 메소드를 호출하여 각 문장들의 구문 트리를 따라서 검색하면서 지역 변수부의 구성요소를 찾아 LocalMember 객체로 등록한다.

그림 10의 CatchStatement 클래스 정의에서 mod 필드는 id 필드의 접근 수정자, texpr 필드는 예외형을 의미한다. 그리고 id 필드는 매개변수의 이름, body 필드는 catch절 내의 문장을 의미한다.

```
class CatchStatement {
        int mod;
        Expression texpr;
        Identifier id;
        Statement body;
                ...
        public CatchStatement()
        check()
        code()
}
```

그림 10. CatchStatement 클래스 정의

### 4.2 바이트코드 최적화 성능 평가

본 논문에서는 효율적인 바이트코드 생성을 위한 알고리즘을 평가하기 위해서 기존의 자바 컴파일러를 수정・추가하였다. 논문에서 제시한 최적화 알고리즘을 적용하여 구현된 컴파일러와 선(SUN)사에서 제공하는 기존의 자바 컴파일러에 대해 자바 응용프로그램들이 번역된 바이트코드의 크기를 비교하였다.

표 1은 기존의 자바 컴파일러로 응용프로그램을 번역한 바이트코드의 크기와 본 논문에서 제시한 알

표 1. 바이트코드 생성

단위 : 바이트(byte)

| 파일명 | 본논문 | javac | difference |
|---|---|---|---|
| SayName.java | 1,794 | 1,935 | 141 |
| CalcuObject.java | 1,759 | 1,904 | 145 |
| CustomerService.java | 2,741 | 3,015 | 274 |
| Calculator.java | 2,476 | 2,714 | 238 |
| HelloRMI.java | 1,634 | 1,780 | 146 |
| BankImpl.java | 2,162 | 2,366 | 204 |
| RMIDayTime.java | 1,669 | 1,811 | 142 |
| Bank.java | 2,162 | 2,366 | 204 |
| PoolingChatServer.java | 2,156 | 2,343 | 187 |
| ChatClient.java | 1,755 | 1,899 | 144 |
| ChatServerImp.java | 2,346 | 2,562 | 216 |
| EchoServer.java | 1,764 | 1,904 | 140 |
| ComputeEngine.java | 1,749 | 1,892 | 143 |
| MyRmiImpl.java | 1,692 | 1,829 | 137 |
| EchoImpl.java | 1,768 | 1,906 | 138 |
| Calendar.java | 1,659 | 1,799 | 140 |
| Pserver.java | 5,731 | 6,316 | 585 |
| UserImpl.java | 4,369 | 4,775 | 406 |
| ChatServer.java | 3,392 | 3,742 | 350 |
| ClientHandler.java | 2,535 | 2,778 | 243 |
| HelloImpl.java | 1,631 | 1,768 | 137 |
| RMIDayTimeImpl.java | 1,671 | 1,813 | 142 |
| RMIChatServer.java | 2,413 | 2,628 | 215 |
| GernericServer.java | 1,784 | 1,927 | 143 |
| Average Bytes | 2,284 | 2,491 | 207 |

고리즘을 적용하여 번역한 바이트코드의 크기를 보여주고 있다.

선사의 자바 컴파일러는 catch절의 매개변수에 대해 각각 독립된 변수로 취급하여 메소드 호출 시 기억장소를 달리하여 할당한다. 이는 생성된 바이트코드의 크기를 증가시키고, 실행에 있어서도 더 많은 기억장소를 요구한다. 따라서, 예외처리 구문 try-catch-finally에서 catch 절의 매개변수에 대한 기억장소를 공유하게 함으로써 생성되는 바이트코드의 크기를 줄일 수 있었다.

본 논문에서 제시한 알고리즘을 적용하여 번역해 생성한 바이트코드와 자바 컴파일러를 사용하여 번역해 생성한 바이트코드의 크기를 비교하면 평균적으로 207바이트의 차이가 발생한다. 선사에서 제공하는 기존의 자바 컴파일러를 통해서 생성된 바이트코드의 크기를 100으로 볼 때 본 논문에서 제안한 방법을 적용해 생성된 바이트코드의 크기는 91.6으로 감소함을 알 수 있다.

예외가 발생하면 예외의 형에 해당되는 예외처리기를 찾기 위해서 catch구문에 나열된 순서대로 검사한다. 따라서 발생한 예외는 가장 먼저 나타난 catch 블록에 대응되는 예외 처리기에 의해서 처리된다. 따라서 예외처리기를 기술하는 순서는 매우 중요하다. 특히, 서로 상속관계에 있는 예외의 형에 대한 예외처리기인 경우에는 반드시 서브클래스에 대한 예외처리기를 먼저 기술해 주어야 한다. 그렇지 않을 경우, 컴파일러는 오류를 발생한다.

번역 시에 모든 예외처리기의 각 매개변수에 대해 지역 변수부 내의 인덱스를 설정하고, 실행 시에 이 인덱스에 해당하는 지역 변수부의 위치를 사용하여 예외를 처리하게 된다. 따라서 번역 시에 메소드 수행에 필요한 지역 변수부의 크기와 인덱스가 결정된다. 그런데 실제 실행 시에는 메소드 호출과 동시에 모든 지역변수들을 지역 변수부의 인덱스에 위치시키는 것이 아니고 호출이 이루어지면, 명령 수행과정 중에 필요한 지역변수의 인덱스에 해당하는 기억장소를 사용하여 수행한다. 그런데 여러 예외가 동시에 발생할 수 없기 때문에, 한꺼번에 여러 예외처리기의 매개변수에 대한 기억공간을 할당할 필요가 없다. 따라서 다른 예외처리기의 매개변수들 사이에는 기억장소의 공유가 가능하게 된다.

## 5. 결론 및 향후 활용방안

본 논문에서는 자바카드 플랫폼의 성능 개선을 위해서 최적화된 바이트코드를 생성하기 위한 방안을 제시하였다.

예외처리 구문 try-catch-finally에서 catch절의 매개변수에 대한 기억장소를 공유함으로써 생성되는 바이트코드의 크기를 줄일 수 있고, 실행 시 필요한 메모리도 줄일 수 있다.실제로 본 논문에서 제시한 바이트코드 최적화 알고리즘은 기존의 자바 컴파일러를 통해서 생성된 바이트코드의 크기를 9% 정도 줄일 수 있었다. 이러한 바이트코드 크기의 감소는 저장공간의 감소를 의미하므로 일반 컴퓨터 시스템 환경보다 내장 시스템이나 스마트 카드와 같은

자원 제약적인 시스템 환경에서 더욱 효과가 크다.

본 논문에서 제시한 방안들을 다양한 응용서비스를 개발에 적용하여 활용할 수 있다. 현재 수요가 일어나고 있는 정보통신 · 금융 · 의료 · 교통 · 가전 · 보안 등 각종 응용 분야의 서비스 개발에 접목을 시도함으로써 사용자들에게 더욱 향상된 자바카드 서비스를 제공할 수 있다.

## 참 고 문 헌


[1] Zhiqun Chen, *Java Card Technology for Smart Cards*, Addison-Wesley, 2000.

[2] Sun Microsystems, Inc., *The Java Card$^{TM}$ 2.2 Virtual Machine Specification*, SUN, 2002.

[3] Uwe Hansmann, Martin S. Nicklous, Thomas Schack, Frank Seliger, *Smart Card Application Development Using Java*, Springer, 2002.

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

[5] Sun Microsystems, Inc., *The Java Card$^{TM}$ 2.2 Runtime Environment(JCRE) Specification*, SUN, 2002.

[6] Sun Microsystems, Inc., *cJDK_Users_Guides*, SUN, 2002.

[7] T. Lindholm and F. Yellin, *The Java$^{TM}$ Virtual Machine Specification* ADDISON- WESLEY, 1997.

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

[9] Gupta M, Choi J-D, Hind M., "Optimizing Java programs in the presence of exceptions", *In Proceedings of the European Conference on Object-Oriented Programming, Cannes, France, 2000.(Lecture Notes in Computer Science,vol.1850) Bertino E(ed). Springer-Verlag*, pp. 422~446, 2000.

[10] Adl-Tabatabai and M. Cierniak and Huei-Yuan Lueh and V. M. Parikh and J. M. Stichnoth, "Fast, Effective Code Generation in a Just-In-Time Java Compiler", *In Proceedings of the ACM SIGPLAN*, pp. 280-290, 1998.

[11] Aho, A. V., Sethi, R., Ullman, J. D., *Compilers : Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[12] Cierniak. M. and Li, W., "Optimizing Java bytecodes", Concurrency: Practice and Experience 9, pp. 427-444, 1997.

[13] Clausen. L. R, "A Java bytecode optimizer using side-effect analysis", Concurrency: Practice and Experience 9, pp. 1031-1045, 1997.

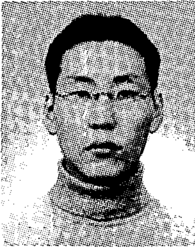
[14] Lars R. Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller, "Java Bytecode Compression for Low-End Embedded Systems", ACM Transactions on Programming Languages and Systems, Vol. 22, No. 3, 2000.

[15] Nystrom, N., *Bytecode level analysis and optimization of java classes, M.S thesis*, Purdue University, 1998.

[16] Saurabh Sinha and Mary Jean Harrold, "Analysis and Twsting of Programs with Exception Handling Constructs", IEEE Computer Society, Vol. 26, No. 9, 2000.

[17] http://www.artima.com/, ARTIMA SOFTWARE COMPANY.

[18] http://java.sun.com/, Sun Microsystem, Java Home Page.



김 도 우

1997년 2월 경남대학교 컴퓨터공학과 학사
1999년 2월 경남대학교 컴퓨터공학과 공학석사
1999년 3월～현재 경남대학교 컴퓨터공학과 박사과정
관심분야 : Java Technology, HomeNetworking, JavaCard

정 민 수

1986년 서울대학교 컴퓨터공학과 학사
1988년 한국과학기술원 전산학과 공학석사
1994년 한국과학기술원 전산학과 공학박사
1990년～현재 경남대학교 정보통신공학부 교수
관심분야 : Embedded System, JavaMachine, Compiler

교 신 저 자

김 도 우 631-701 경남 마산시 월영동 449 경남대학교 컴퓨터공학과