2001년 한국정보처리학회 춘계 학술발표논문집 제8권 제1호

# 자바 Bytecode에서 MSIL로의 변환을 위한 번역기의 설계 및 구현

민 정 현[0] 오 세 만
동국대학교 컴퓨터공학과
salvatore@orgio.net smoh@dgu.ac.kr

# Design and Implementation of a Translator for Translating Java Bytecode into MSIL

Junghyun Min[0] Seman Oh
Dept. of Computer Engineering, Dongguk University

요 약

자바는 객체지향 언어이고, 한번 작성된 프로그램은 자바 가상 기계가 있는 모든 곳에서 수정없이 실행될 수 있기 때문에 소프트웨어의 개발과 유지 보수에 많은 장점을 가진 언어이다. 이러한 특징으로 인하여 개발되는 제품들이 자바로 구현되는 경우가 많다. 그러나 아직 대다수 소프트웨어 개발자들은 주로 C 언어나 C++ 언어를 사용하고 있으며, 최근에는 C#이라는 언어를 사용하고 있다. 자바가 플랫폼에 독립적인 장점은 가지고 있지만, 다수의 개발자 및 사용자가 마이크로소프트 윈도우 운영체제를 사용하고 있다는 것을 감안한다면 그리 탁월한 장점만은 될 수 없다. 또한, 최근의 개발동향이 COM(Component Object Model)을 지향하고 있고, 이는 더 이상 개발자들에게 프로그래밍 언어에 구애를 받지 않고 오직 개발 제품에 대한 집중력을 가질 수 있는 환경을 제공할 수 있다면, 그 개발 효율에 있어서 상당한 이점을 가질 수 있다는 의미이다. 따라서, COL(Component Object Language)을 기반으로 하고 있는 C#(C sharp)언어를 사용하여 개발을 함에 있어서 자바의 언어를C# 언어로 변환할 수 있다면, 신생 언어인 C#에 있어서 기존 자바로 되어 있는 유용한 개발 제품들을 보다 효율적으로 이용할 수 있을 것이다.

본 논문에서는 두 언어(자바, C#)를 하나로 잇는 교량(bridge)역할을 할 수 있도록 자바의 중간 언어인 Bytecode를 C#의 중간 언어인 MSIL(MicroSoft Intermediate Language)로 바꿀 수 있는 중간 언어 번역기를 설계하고 구현하였다. 이를 위한 방법으로는 먼저, 자바 Bytecode와 MSIL의 어셈블리 형태에서의 명령어 매칭과정을 매핑 테이블을 이용하여 처리하였고, MSIL에서 자바 Bytecode의 함수와 같은 기능을 하는 메소드의 변환을 위하여 마크로 변환기법을 이용하여 해결하였다.

## 1. 서론

자바로 작성된 프로그램은 JVM이 있는 곳에서는 프로그램의 수정과 재컴파일 과정 없이 실행가능하고, 객체 지향을 기반으로 하는 언어이기 때문에 최신 객체지향 기술을 쉽게 적용할 수 있다는 장점을 가진다. 그러나, 최근의 기술 동향을 살펴보면, 네트웍 환경이 점점 빠르게 발전함에 따라, COM을 기반으로 하는 컴포넌트 기반 기술들이 중점을 이루고 있는 것을 볼 수 있다. 또한, 실제로 자바를 사용하는 개발자들과 자바로 구현되는 라이브러리가 시간이 갈수록 늘어남에도 불구하고, C/C++를 사용하는 개발자가 훨씬 많은 것이 현실이다. 또한, 최근 마이크로소프트 사의 Visual studio.NET 플랫폼에서 제공하고 있는 C# 언어가 상당한 주목을 받고 있는 가운데, 이를 이용한 컴포넌트의 개발이 폭 넓게 적용되어 지고 있다. C# 언어는 마이크로소프트 사의 Visual Basic 과 같은 코드 작성 편리함과 유지보수의 용이성, 그리고 C++의 유연함과 강력함을 바탕으로 COM+와 웹 서비스를 더욱 쉽고 빠르게 개발할 수 있는 언어이다. 이는 기존의 마이크로소프트 사에서 제공하는 컴포넌트를 모두 재사용할 수 있고, 윈도우즈 API 를 80% 이상 수용하고 있다. 따라서, C# 이 실행되는 플랫폼에 기존에 개발되어진 자바 라이브러리들을 이용할 수 있다면, 개발에

본 연구는 한국과학재단의 특정기초연구(과제번호: 1999-1-30300-3) 지원에 의한 것임.

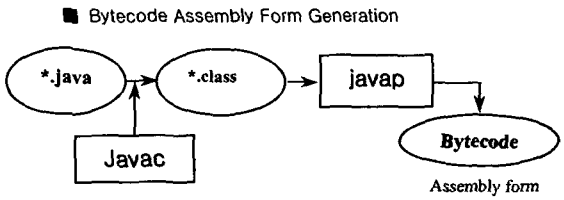
있어서 가장 최적화된 환경에서 최신의 개발 기술들을 이용할 수 있을 것이다. 이를 위하여 본 논문에서는 자바의 중간 언어인 Bytecode 를 C#의 중간 언어인 MSIL 로 바꿀 수 있는 번역기를 언어적인 부분에서부터 설계 및 구현하여, 궁극적으로 C#에서 자바언어로 짜여진 프로그램을 사용할 수 있게 하는 것을 목적으로 한다.

## 2. 관련 연구

### 2.1 Bytecode 개요

Bytecode 는 JVM(Java Virtual Machine)의 기계언어로 간주할 수 있고, JVM 이 클래스 파일을 로드할 때, 클래스 내에서 각각의 메소드에 대하여 스트림 형태로 얻어지며, 이 때의 스트림은 8-bit의 바이트로 구성된 이진스트림 형태이다. 또한 Bytecode 는 기본적으로 스택 지향 구조를 가지고 있고, 처음부터 인터프리터를 목적으로 설계 되었으며( JVM 에 의해 인터프리트 되든지, 클래스 로딩시 컴파일된다 ), 한편으로 Bytecode 는 프로그램의 기능추가 및 성능향상을 위하여 Bytecode 의 변경을 해야 하고, 클래스 및 객체는 반드시 자바언어로 구성하여야 한다는 제약성도 가지고 있다.

다음은 자바 원시코드에서 Bytecode 어셈블리 파일이 얻어지는 과정을 도식화한 것이다. Javac 에 의해서 얻어진 클래스 파일을 IL Translator 의 입력인 어셈블리 형태의 파일로 얻어내기 위하여 JDK1.3 의 javap -c Option 을 사용한다. javap 는 클래스 파일안에 있는 여러 정보들 중에서 원시코드에 대한 내용들만을 추출하여 주는 기능을 수행한다.

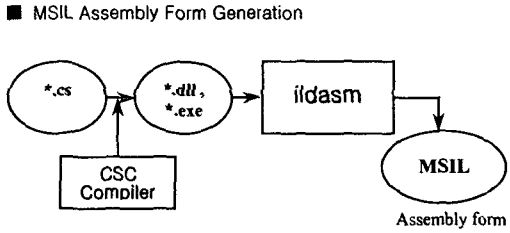



[그림 1] Bytecode 시스템 구성도

### 2.2 MSIL 개요

MSIL 은 Microsoft Intermediate Language 의 약자로 C#의 중간 언어이며, 컴파일러나 다른 도구에 의해 원시코드로부터 쉽게 생성될 수 있도록 설계된 스택 기반의 명령어 집합이다. 이 명령어의 종류에는 산술/논리 연산, Control flow, DMA, 예외처리, Method Invocation 등이 있고, 객체지향 프로그래밍 구조에 영향을 주는 Virtual Method Call, Field Access, Array Access, 객체 할당과 초기화 등도 MSIL 에서 직접 지원하는 명령어 종류이다. MSIL 명령어 집합은 스택상에서 데이터 타입들을 탐색하고, 직접 인터프리트된다. 또한, MSIL 은 하드웨어 및 플랫폼에 독립적이고, 처음부터 JIT 를 목적으로 설계 되었으며, 자바언어 와는 다르게 처음부터 언어독립적으로 설계되어, 포괄적인 프로그래밍(Generic Programming)을 목적으로 하기 때문에 프로그램의 기능 및 구조의 변화에 잘 적응하는 언어이다.

다음은 IL Translator 의 결과 형태를 알아보기 위하여 역으로 MSIL 의 코드를 다음의 방법으로 추출하여 변환될 부분의 형태를 예측하도록 한다. *.cs 파일은 C#의 소스코드이며, CSC(C# Compiler)에 의해 dll 이나 exe 와 같은 실행 파일로 변환되고, 이 실행 파일을 어셈블리 코드 형태인 MSIL 로 보기 위해서 마이크로소프트 사의 Visual Studio .NET 에서 제공하는 디어셈블리 유틸리티인 ildasm.exe 를 이용하여 MSIL 의 형태를 추출해낸다. 이 부분에서 생성되는 코드는 자바언어의 클래스 파일에서와 같이 다양한 정보들을 포함하고 있으며 본 논문에서는 프로그램의 코드 부분에 대한 정보들만을 기초로 정보를 추출하기로 한다.




[그림 2] MSIL 시스템 구성도

## 3. 번역기의 설계

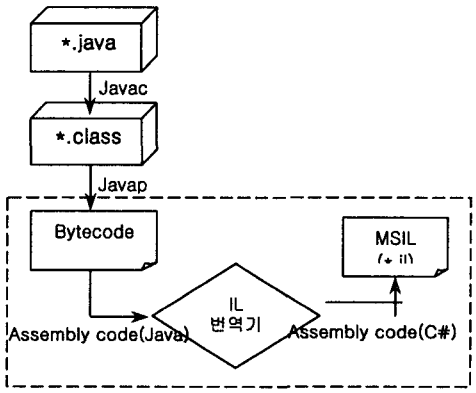
### 3.1 IL Translator 기본 개념

이 절은 IL Translator 의 설계에 대한 부분으로 매크로 변환 기법을 이용하여 Bytecode 를 MSIL 로 변환하는 코드 번역기를 설계하고 구현 하도록 한다. 이는 아래 [그림 4]에서와 같이 IL Translator 를 이루는 주요 부분인 명령어 매핑 부분과 함수형태 구조변환 부분이 Bytecode 와 MSIL 간에 기능적으로 동일한 부분이 될 수 있도록 코드간의 매칭을 매핑테이블을 이용하여 변환할 수 있다는 가정 하에서 이루어 진 것이다. 또한, 이를 위하여 실제 구현에서는 매크로 변환 기법을 이용하여 코드간에 각각에 해당하는 테이블을 참조하기로 한다. 따라서, 그 결과 코드와 입력 코드의 기능을 같게 하고, 궁극적으로 전체 프로그램의 기능을 같게 하여, 자바언어로 짜여진 프로그램을 C# 언어에서 사용 가능하도록 구현하는 것을 목적으로 한다.

### 3.2 IL Translator 구조 및 동작원리

다음의 [그림 3]은 IL Translator 의 시스템 구성도로 전체적으로 자바에서 C#의 중간 언어인 MSIL 까지 변환하는 과정을 도식화한 것이다. 먼저, 변환 하고자 하는 프로그램의 소스파일인 *.java 를 Javac 에 의해서 *.class 로 바꾸고 이를 어셈블리 형태로 얻어내기 위

해서 Javap -c 옵션을 이용하여 코드 부분의 어셈블리 형태를 얻어 낸다. 본 논문에서 해야 할 것은 앞서 만들어 낸 코드부분의 어셈블리 형태를 입력으로 받아서 구현하는 IL Translator 에 의해 MSIL 코드 부분의 어셈블리 형태로 바꾸는 것을 구체적으로 설계 및 구현하는 것이다.




[그림 3] IL Translator 시스템 구성도

위의 [그림 3]에서 점선으로 표시된 부분이 본 논문에서 구현해야 할 부분이다

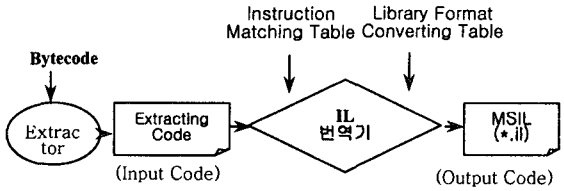
3.3 Translator 제공 기능

C#에서 자바 Bytecode 를 사용하기 위해서 IL Translator 는 다음과 같은 기능을 제공한다.

- Instruction Code Translation
- Library Format Converting
- Code Pattern Customizing

## 4. 번역기의 구현

자바 Bytecode 를 C#의 중간 언어인 MSIL 로 바꾸는 번역기 IL Translator 를 개발하기 위해서는 구체적으로 아래 [그림 4]의 단계로 세부 구현 사항을 진행해 나가기로 한다. 이를 위해서 먼저, 생성된 클래스 파일을 어셈블리 형태인 Bytecode 로 얻어야 하고, 이 부분 중에서 소스 코드 부분만의 정보들을 추출해야 하는데, 그러기 위해서 추출기 역할을 하는 javap -c 를 이용한다. 다음 단계로는 분리된 코드를 IL Translator 의 입력으로 하여 명령어 매핑 테이블과 함수형태 구조변환 테이블을 이용하여 IL Translator 의 결과 코드인 MSIL 형태로 구성하도록 한다. 아래의 그림에서 보면, Bytecode 에서 MSIL 로의 결과를 얻어내기 위해, Extracting 과정을 통해서 코드를 추출하고, 두 가지의 매핑 및 변환과정을 통해서 결과가 나오는 형태를 취한다.




[그림 4] IL Translator 내부 시스템 구성도

4.1 개발환경

이 번역기를 개발하는데 요구되는 기본 환경은 다음과 같다.

- JDK 1.3
- Java Disassemble Utility
  - D-java
  - Decafe pro 3.6
- Visual studio .NET (Version 7.0)
  - FrontPage 2000 Server Extensions
  - Windows 2000 Service pack 1
  - IE 5.5
  - Windows Component Update
    - bootstrap.msi
    - mso9.msi
    - msxml3.msi
  - Microsoft Data Access Components
  - Microsoft .NET SDK(W2K)
- O/S □ Windows 2000 Server 이상
- Database □ MS-SQL 2000

4.2 상세 구현

IL Translator 의 입력 부분을 생성하기 위해서 클래스 파일의 코드부분에 해당하는 것을 Bytecode 형태로 추출한다. 다음은 이를 위한 것으로 아래의 [그림 5]는 자바를 이용하여 정수 i1, i2 에 1, 2 를 각각 입력하여 결과 값으로 i1+i2 값인 3 을 출력하는 프로그램에 해당하는 Bytecode 이다. 4 번 라인에서 먼저 시스템 콜을 하고나서, 10 번 라인에서 해당 메소드의 호출이 이루어 지는 것을 볼 수 있다.

```
0 iconst_1
1 istore_1
2 iconst_2
3 istore_2
4 getstatic #2 <Field java.to.PrintStream.out>
7 iload_1
8 iload_2
9 iadd
10 invokevirtual #3 <Method void print(int)>
13 return
```

[그림 5]. (자바) 코드 부분의 중간 언어

위의 [그림 5]에서 0 에서 13 까지의 명령어를 MSIL 의 출력형태로 얻어 내기 위해서 본 논문에서는 Instruction Matching Table 을 작성하여 입력 코드인 Bytecode 의 해당 라인에 해당하는 명령어를 테이블 참조를 통하여 MSIL 명령어 형태로 만들어 주는 기능을 구현한다. 다음의 [표 1-1]은 위의 [그림 5]에 대한 명령어 매핑 테이블을 도시한 것이다.

| *Ref_num* | *Bytecode Instruction* | *MSIL Instruction* |
|---|---|---|
| 000 | nop | nop |
| : | : | : |
| 004 | iconst_1 | ldc.i4.1 |
| 005 | iconst_2 | ldc.i4.2 |
| : | : | : |
| 027 | iload_1 | ldloc.0 |
| 028 | iload_2 | ldloc.1 |
| : | : | : |
| 070 | istore_1 | stloc.0 |
| 071 | istore_2 | stloc.1 |
| : | : | : |
| 094 | iadd | add |
| : | : | : |
| 098 | isub | sub |
| : | : | : |
| 102 | imul | mul |
| : | : | : |
| 108 | idiv | div |
| : | : | : |
| 177 | return | ret |
| : | : | : |

[표 1-1]. Instruction Mapping Table

위의 테이블에서 Ref_num은 IL_Tranlator에서 참조하는 참조번호이며, Sun 사에서 제공하는 Bytecode 참조 문서의 순서를 따른 것이다.

다음으로 코드에서 호출되는 함수에 대한 변환문제인데, 이는 함수가 호출되는 형태가 메소드 이름만 바뀌고, 전체 형태는 거의 유사하므로 MSIL 로의 형태 변환 유도하기 위해서 마크로 변환기법을 이용하여 해결했다. 즉, 위의 [그림 5]에서의 시스템 호출 부분에서의 코드인 getstatic #2 <Field java.to.PrintStream.out> 에서 getstatic 은 MSIL 에서 call 로, <Field java.to.PrintStream.out>은 instance void [mscorlib]SystemObject::.ctor() 로 변환 되므로 이 형태를 맞추어 주고, 메소드 호출 부분인 invokevirtual #3 <Method void print(int)>에서 invokevirtual 도 역시 call 로 호출하고, <Method void print(int)> 부분은 void [mscorlib]System.Console::Write(int32)로 변환한다.

다음의 [표 1-2]는 위의 테이블 매핑 및 변환 과정을 거쳐서 변환된 코드를 보여 준다. 이는 javap 에 의해 보이는 Bytecode 의 테이블 참조 및 메소드 형태 변환에 의한 MSIL 의 결과를 나타낸 것으로 코드 부분만의 변환을 나타낸 것이다.

| Bytecode | MSIL |
|---|---|
| iconst_1 | ldc.i4.1 |
| istore_1 | stloc.0 |
| iconst_2 | ldc.i4.2 |
| istore_2 | stloc.1 |
| getstatic #2 <Field jav | instance void [mscorlib |
| a.lang.Object()> | ]System.Onject::.ctor() |
| iload_1 | ldloc.0 |
| iload_2 | ldloc.1 |
| iadd | add |
| invokevirtual #3 <Metho | call void [mscorlib]Sys |
| d void print(int)> | tem.Console::Write(int32) |
| return | ret |

[표 1-2]. Bytecode to MSIL 변환 결과

## 5. 결론 및 향후 연구방향

본 논문에서는 C# 언어의 환경하에서 자바 코드를 사용할 수 있게 하기 위해서 C#의 중간 언어인 MSIL 과 자바의 중간언어인 Bytecode를 비교 분석하여, Bytecode 를 MSIL 로 변환할 수 있는 번역기를 설계 및 구현하였다.

현재 연구의 진행정도는 Bytecode 를 MSIL 로 변환하는 과정에서 어셈블리 명령어 매핑 테이블과 함수형태 구조 변환 테이블을 구성하였고, 전체적인 코드 형태를 대형 프로그램에 대해서도 변환할 수 있도록 최적화 연구를 수행하고 있다.

향후 연구방향은 자바 언어에서 지원하는 기능 및 모듈들을 C#에서도 동일하게 사용할 수 있는 개발 환경을 구축할 뿐만 아니라, C#에서 사용할 수 있는 언어적인 장점을 자바에서도 사용 가능하게 만드는 것을 목표로 하고 있다. 또한, 이것이 단순히 언어의 일대일 변환에서 그치는 것이 아닌, 코드 형태의 분석에 의한 통계적인 코드 최적화로 이어지기 위한 연구를 지속적으로 해 나갈 것이다.

## 참고 문헌

[1]. Bill Venners, Bytecode basics, A first look at the bytecodes of the Java virtual machine
[2]. Eric Gunnerson, A Programmer's Introduction to C#, Apress™ , 2000
[3]. Java Bytecode Assembler, http://www.eg.bucknell.edu/~cs360/java-assembler/
[4]. Ken Arnold & James Gosling, The Java™ Programming Language, 1997
[5]. Microsoft.NET , C# & ASP.NET Programming, .net press
[6]. Microsoft Corporation, C# Language Specification, Nov. 20. 2000
[7]. Microsoft Corporation, MSIL Instruction Set Specification, Nov. 20. 2000
[8]. Microsoft Corporation, The IL Assembly Language Programmers'Reference, Oct. 10. 2000
[9]. O'REILLY, Jon Meyer & Troy Downing, Java Virtual Machine, Mar. 1997