바이트코드로부터 네이티브 코드 생성을 위한 중간 코드 변환기의 설계 및 구현

고 광 만†

요. 약

자바 프로그래밍 언어는 웹 브라우저에서 실행되는 작은 크기의 응용 프로그램 수행에서는 실행 속도 문제가 중요한 요소가 아니지만 대형 프로그램의 수행에서는 실행 속도가 현저히 저하되는 단점을 지니고 있다. 이러한 문제점을 해결하기 위해 전통적인 컴파일 방법을 사용하여 바이트코드를 특정 프로세서에서 수행될 수 있는 목적기계 코드로 변환하는 다양한 연구가 진행중이다. 본 연구에서도 자바 응용 프로그램의 실행 속도의 개선을 위해 바이트코드로부터 직접 i386 코드를 생성하는 네이티브 코드 생성 시스템을 위한 중간 코드 변환기를 설계하고 구현한다. 중간 코드 변환기는 자바 언어의 중간 코드인 *.class 파일을 입력으로 받아 레지스터 기반의 중간 코드로 변환한다.

Design and Implementation of Intermediate Code Translator for Native Code Generation from Bytecode

Ko Kwang-Man†

ABSTRACT

The execution speed is not an important factor for Java programming language when implementing small size application program which is executed on the web browser, but it becomes a serious limitation when the huge-size programs are implemented. To overcome this problem, the various research is conducted for translating the Bytecode into the target code which can be implemented in the specific processor by using classical compiling methods.

In this research, we have designed and realized an intermediate code translator for the native code generation system with which we can directly generate i386 code from Bytecode to improve the execution speed of Java application programs. The intermediate code translator generates the register-based intermediate code from *.class files which are the intermediate code of Java.

Key words: 자바 프로그래밍언어, 바이트코드, 코드 확장(Macro Expansion)

1. 서 론

자바 프로그래밍 언어는 인터넷 및 분산 환경 시스템에서 효과적으로 응용 프로그램을 작성할 수 있도록 설계된 언어로서 객체지향 패러다임 및 다양한 개발 환경을 지원하고 있다[5,7]. 자바 언어 시스템에서는 자바 언어를 플랫폼에 독립적으로 실행시키기 위해 가상 기계 코드인 바이트코드를 사용하며 자바 기상 기계의 인터프리터를 이용하여 실행하고 있다. 이로 인해 웹 브라우저에서 실행되는 작은 크기의 자바 응용 프로그램 수행에는 실행 속도 문제가 중요한 요소가 아니지만 대형 프로그램의 수행에는 실행 속도가 현저히 저하되며 또한 C/C++와 같은 기존 프로그램의 실행 속도에 비해 매우 느린 단점을 지니고 있다. 따라서 실행 속도의 문제점을 해결하기 위

본 연구는 정보통신부에서 지원하는 대학기초연구지원사업(과제번호: 2001-021-3)으로 수행

† 상지대학교 컴퓨터정보공학부 교수

해 현재 다양한 방법이 연구되고 있다. 첫 번째는 바이트코드를 위한 프로세서를 제작하여 직접 하드웨어로 실행시키고자 하는 의도로서 자바 칩을 이용하는 방법이다. 두 번째는 JIT 방식으로 실행 시간에 필요에 따라 메소드 단위로 컴파일하여 처리하는 방법이다. 세 번째는 전통적인 컴파일 방법을 사용하여 컴파일 과정에서 중간 코드인 바이트코드를 특정 프로세서에 수행될 수 있는 목적 코드로 바꾸는 후단부를 사용하는 방법이다. 따라서 자바 응용 프로그램의 실행 속도 개선을 위해 SUN에서도 기존의 JVM에서 인터프리터를 이용하는 방식과 병행하여 바이트코드를 특정 프로세서의 네이티브 코드로 컴파일하여 수행하는 방법을 사용하고 있다[7].

자바 언어의 중간 코드인 바이트코드는 플랫폼 독립적인 특성으로서 1960년대에 제안된 UNCOL과 유사한 개념이며 바이트코드를 실행하기 위한 시스템인 JVM은 P-코드의 영향을 받았다. 또한 바이트코드는 ACK의 중간 코드인 EM 코드와 유사하게 스택 기반 중간 코드 방식이며 스택 기반의 중간 코드를 특정 기계의 레지스터 기반 목적 코드로 변환하는데 많은 연구가 진행되어 왔다[3,12]. 자바 응용 프로그램의 실행 속도를 개선하기 위한 다양한 시도로서 JVM의 인터프리터 방식과 별도로 JIT 컴파일러의 구현이 연구소 및 산업계에서 진행되어 왔다. CACAO[1]는 Alpha 프로세서를 위한 64비트 JIT 컴파일러로서 바이트코드를 입력으로 받아 Alpha 프로세서를 위한 네이티브 코드를 생성한다. 네이티브 코드 생성 과정에서 스택 기반의 바이트코드는 CACAO에서 설계한 레지스터 기반 중간 코드로 변환된 후 다시 레지스터 기반 중간 코드로부터 Alpha 프로세서를 위한 네이티브 코드로 변환된다. NET 컴파일러[12]는 바이트코드로부터 네이티브 코드를 생성하는 최적화 컴파일러로서 IMPACT에서 제안된 시스템이다. NET 컴파일러에서도 스택 기반의 바이트코드를 특정 프로세서에 적합한 네이티브 코드를 생성하기 위해 고안된 레지스터 기반의 중간 코드를 사용하고 있다. Jcc[10]는 자바 언어를 위한 오프라인 컴파일러로서 바이트코드로부터 직접 x86 및 SPARC 프로세서를 위한 네이티브 코드를 생성하는 시스템이다. Jcc의 전단부에서는 자바 언어를 입력으로 받아 레지스터 기반 중간 코드인 *.gasm 파일을 생성하며 후단부에서는 *.gasm 파일을 입력으로 받아 특정 기계에 대한 어셈블리 코드를 생성한다.

본 연구는 자바 언어의 실행 방법인 (1) 인터프리터 이용 방법, (2) JIT 컴파일러 이용 방법, (3) Bytecode에 대한 네이티브 코드 생성 방법 중에서 (3)번째 방법에 대한 연구이다. 세 가지 방법에 대한 장/단점 등은 이미 다수의 논문[1,12] 등에서 비교되고 있으며 응용 프로그램의 종류에 따라 각각의 장점을 가지고 있다. (3)번째 방법에서, 자바 언어에 대한 오프라인 컴파일러인 Jcc의 후단부(*.gasm -> *.s)가 개발되어 기존의 방식과 비교하여 실행 속도에서 성능(특히, 실행 속도)의 우수성을 보이고 있지만 Jcc는 반드시 입력으로 *.java를 입력으로 사용하므로 자바 소스 언어에 대한 실행만을 허용하고 있다.

따라서 본 연구에서는 자바 프로그램의 실행 속도 개선을 위해 바이트코드로부터 직접 i386 코드를 생성하는 네이티브 코드 생성 시스템을 위한 중간 코드 변환기를 설계하고 구현한다. 중간 코드 변환기는 자바 언어의 중간 코드인 *.class 파일을 입력으로 받아 레지스터 기반의 중간 코드로 변환한다. 이를 위해 Jcc에서 제안된 레지스터 기반의 중간 코드를 사용한다. 따라서 본 시스템에서는 바이트코드의 집합인 *.class 파일을 입력으로 받아 레지스터 기반의 중간 코드인 *.gasm으로 변환하는 중간 코드 변환기(Intermediate Code Translator)를 구현한다. 변환된 *.gasm을 입력으로 받아 i386 코드를 생성하는 네이티브 코드 생성 시스템은 Jcc의 후단부를 이용하여 변환된 코드를 검증한다. 중간 코드 변환기를 사용하면 *.class 파일을 쉽게 레지스터 기반 목적기계에서 실행할 수 있으며 코드를 생성할 수 있는 장점을 갖는다. 다만, 중간 코드 변환에 소비되는 번역시간 부담이 (1), (2)번 방법과 비교하여 발생되지만 응용 프로그램의 종류(반복 동작을 많이 수행하는 프로그램) 등에 따라 큰 문제가 되지 않는다.

본 논문의 구성은 제 2장에서 본 연구의 기반이 되는 CACAO, IMPACT의 NET, Jcc와 같은 네이티브 코드 생성 시스템에 대한 고찰 및 본 연구의 기반이 되는 클래스 파일과 레지스터 기반 중간 코드에 대해 소개한다. 제 3장에서는 본 연구에서 구현하고자 하는 전체 시스템 모델 및 중간 코드 번역기 모델과 각 요소의 특성에 대해 기술한다. 구체적으로 3.2절에서 실제적으로 바이트코드를 입력으로 받아 코드 확장 방식으로 레지스터 기반 중간 코드를 생성하는

과정과 생성된 결과를 기술한다. 3.3절에서는 중간 코드의 변환 결과 및 생성된 네이티브 코드에 대한 정확성을 증명하며 실험 결과를 제시한다. 4장에서는 본 연구의 결과와 향후 연구 방향에 대해 기술한다.

2. 기반 연구

2.1 네이티브 코드 생성 시스템

CACAO[8]는 Alpha 프로세서를 위한 64비트 JIT 컴파일러로서 바이트코드를 입력으로 받아 Alpha 프로세서를 위한 네이티브 코드를 생성한다. 네이티브 코드 생성 과정에서 스택 기반의 바이트코드는 CACAO에서 설계한 레지스터 기반 중간 코드로 변환된 후 다시 레지스터 기반 중간 코드로부터 Alpha 프로세서를 위한 네이티브 코드로 변환된다. 네이티브 코드 생성 시에 지역 변수와 스택의 위치는 32 비트 크기를 갖는 pseudo 레지스터로 대치되며 pseudo 레지스터로부터 실제 Alpha 프로세서를 위한 레지스터 할당을 위해 빠른 레지스터 할당 알고리즘이 적용되며 실제적으로 목적 코드 생성 시에는 필드 오프셋이 64 비트 크기로 조정된다. CACAO 시스템은 컴파일 시간과 로딩 시간 부담이 발생되지만 JDK 인터프리터에 비해 85배정도 빠르게 자바 프로그램을 실행하며 Kaffe JIT에 비해 약 8배 가량의 속도 향상을 얻을 수 있다. CACAO 시스템은 바이트코드로부터 네이티브 코드를 생성하기 위해 4 단계를 거쳐 진행된다. 첫 번째 단계에서는 기본 블록이 결정되며 두 번째 단계에서 각각의 바이트코드가 레지스터 기반 중간 코드로 변환된다 세 번째 단계에서는 실질적인 레지스터 할당이 진행되며 네 번째 단계에서 네이티브 코드 생성이 이루어진다. 실질적으로 "a=b*c+d"에 대한 바이트코드로부터 레지스터 기반 중간 코드로 변환된 결과는 표 1과 같다.

NET 컴파일러[12]는 바이트코드로부터 네이티브 코드를 생성하는 최적화 컴파일러로서 IMPACT에서 제안된 네이티브 코드 생성 시스템이다. NET 컴파일러에서도 스택 기반의 바이트코드로부터 특정 프로세서에 적합한 네이티브 코드를 생성하기 위해 고안된 레지스터 기반의 중간 코드를 사용하고 있다. NET 컴파일러에서 바이트코드로부터 네이티브 코드를 생성하는 과정은 그림 1과 같다.

NET 컴파일러에서는 자바 언어의 중간 코드인 클래스 파일로부터 네이티브 코드 생성을 위한 내부 코드 형태인 Java IR로 변환된 후에 효율적인 코드 생성을 위한 최적화 동작이 수행되며 최적화된 내부 코드로부터 네이티브 코드가 생성된다. NET 컴파일러의 목적은 자바 응용 프로그램의 수행 속도를 개선하기 위해 전통적인 컴파일 방식을 적용하여 기존의 C/C++ 언어와 같이 네이티브 코드를 생성한 후 실행 속도 등에서 성능을 높였다. 또한 스택 기반 중간 코드로부터 레지스터 기반 중간 코드로 변환하는 기법과 다양한 최적화 기법을 제시하였다.

Jcc[9]는 오프라인 상태에서 자바 언어를 입력으로 받아 네이티브 코드를 생성하는 컴파일러이다. 이식이 용이한 컴파일러 제작을 위해 RTL에 기반을

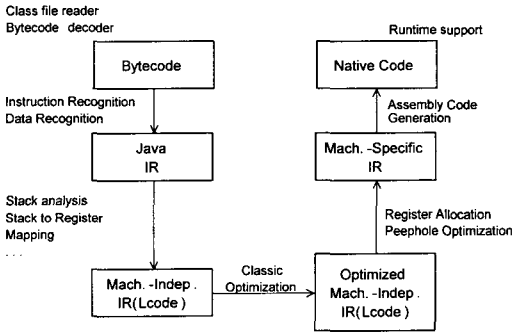

그림 1. NET 컴파일러 네이티브 코드 생성 모델

표 1. CACAO 중간 코드 변환

바이트코드	CACAO 중간 코드
iload b ; 변수 b의 값을 로드	
iload c ; 변수 c의 값을 로드	OP2(IMUL) b, c, t2
imul ; b*c를 계산	;t0 == b, t1 == c
iload d ; 변수 d의 값을 로드	OP2(IADD) t2, d, a
iadd ; (b*c)+d를 계산	;t3 == d, t4 == a
istore a ; 변수 a의 값을 스택의 탑에 저장	

둔 레지스터 기반 중간 코드(*.gasm)를 사용하고 있으며 중간 코드에 대한 다양한 최적화 동작을 수행한다. Jcc에서 자바 언어를 입력으로 받아 네이티브 코드를 생성하는 과정은 그림 2와 같다.

Jcc의 전단부에서는 Jade라는 파서 생성기를 구현하여 파서를 생성하며 생성된 어휘 분석기와 파서를 이용하여 자바 언어의 입력에 대해 Jcc의 중간 코드인 *.gasm 코드를 생성한다. gasm 중간 코드는 네이티브 코드 생성을 용이하게 하고 효율적인 코드를 생성할 수 있도록 고안된 중간 언어로서 RTL 중간 코드와 유사한 특성을 가지고 있으며 스택 기반과 레지스터 기반 특성을 동시에 갖고 있다. gasm의 개략적인 특성 및 명세는 2.3절에서 기술한다.

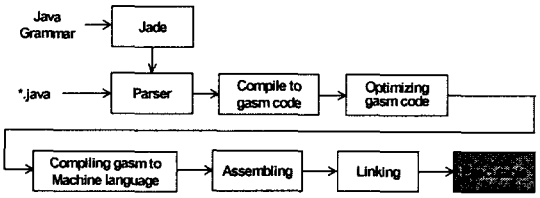

그림 2. Jcc, 자바 오프라인 컴파일러

2.2 클래스 파일 및 바이트코드

클래스 파일은 플랫폼 독립성을 위해 설계된 8비트 크기의 바이트 스트림으로 구성되어 있다. 따라서 16비트, 32비트, 64비트 크기는 2개, 4개, 8개의 연속적인 바이트 스트림으로 구성되며 데이터 아이템은 상위 바이트가 먼저 오는 순서로 저장된다. 또한 클래스 파일은 ClassFile 구조체를 이용하여 그림 3과 같은 형식을 갖는다.

클래스 파일에서 상수 기억 장소인 constant_pool은 Class, Fieldref, Methodref와 같은 타입을 가지고 있다. 속성 테이블인 attributes 필드에는 클래스 파일에서 미리 정의된 Sourcefile, Constantvalue, Code, Exceptions, Linenumbertable, Localvariabletable에 대한 구조체로 구성되어 있다. 클래스나 인터페이스의 메소드에 대해 기술되어 있는 method_info 타입의 구조체에는 메소드에 대한 접근 권한을 갖는 access_flags와 메소드의 이름을 나타내기 위해 상수 기억 장소에 대한 인덱스를 갖는 name_index, 자바 메소드 기술자를 나타내기 위해 상수 기억 장소에 대한 인덱스를 갖는 descriptor_index가 있다. 또한, 메소드가 갖는 attribute 테이블의 수를 나타내는 attributes_count, 속성 테이블이 기술되어 있는 attributes로 구성되어 있으며, 이 attributes에 있는 Code 구조체에 바이트코드가 저장되어 있다. Code 속성을 갖는 구조체는 그림 4와 같다.

```
ClassFile {
    u4 magic;
    // . . .
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    // . . .
    attribute_info attributes[attributes_count];
}
```

그림 3. 클래스 파일 구조체

```
Code_attribute{
    u2 attribute_name_index;
    // . . .
    u2 exception_table_length;
    {
        // . . .
    }
    // . . .
    attribute_info attributes[attributes_count];
}
```

그림 4. 코드 속성 구조체

자바 컴파일러는 초기화 블록과 각종 메소드에 포함된 프로그램 코드를 자바 바이트코드로 변환시킨다. 이러한 바이트코드는 자바 가상 기계의 어셈블리 코드라고 할 수 있으며 모든 명령어가 한 바이트 크기로 구성되어 있다. 또한 바이트코드는 기본적으로 스택 기반 구조를 가지고 있으며 인터프리터 방식으로 처리된다. 즉, 바이트코드의 피연산자는 컴파일 시간에 결정되며 실행 시간에 실질적으로 계산되어 결과 값이 스택에 저장되는 형태를 가진다. 바이트코드에서 지원하는 자료형은 크게 정수형에 속하는 byte, short, int, long, char 형과 실수형에 속하는 float, double 형으로 구분된다. 바이트코드는 크게 니모닉(operation code)과 피연산자로 이루어진다. 실제 동작 명령에 해당되는 것은 니모닉이며 피연산자는 니모닉이 사용할 부가 정보를 제공한다. 니모닉은 종류에 따라 필요로 하는 피연산자 개수가 달라지

며 때로는 피연산자 스택에서 피연산자를 가져와 사용하기도 한다. 또한 대부분의 경우 수행된 결과 값은 다시 피연산자 스택에 저장된다.

2.3 gasm 중간 코드

본 연구에서 스택 기반 바이트코드들 입력으로 받아 중간 코드 변환기에 의해 생성되는 gasm 코드는 자바 언어를 입력으로 받아 네이티브 코드 생성을 오프라인 자바 컴파일러인 Jcc에서 사용하는 레지스터 기반 중간 코드이다. 이러한 gasm은 move, add, compare 등과 같은 명령어를 가지고 있으며 개략적인 형식은 그림 5와 같다.

```
Load {R0-R3} (constant)
Push {R0-R3}
Pop  {R0-R3}
Move Offset {R0-R3}
Move {R0-R3} Offset
{Add, Sub, Or, And, ShiftLeft, ...} {R0-R3} (constant)
{Add, Sub, Or, And, ShiftLeft, ...} {R0-R3} {R0-R3}
Jump {Bigger, Smaller, BiggerEqual, . . ., NotEqual}
Jump {R0-R3}
. . .
```

그림 5. gasm의 중간 코드 형식

일반적인 산술 명령, 이동 명령 등은 스택 대신에 레지스터에서 동작되며 조건문의 실행과 반복문 등의 실행을 위해 플래그-레지스터를 사용한다. gasm에서 레지스터는 네 개의 범용 레지스터와 한 개의 기본 주소 레지스터를 명시적으로 지원하고 있으며 실질적으로는 네이티브 코드들 생성하고자 하는 목적기계에 적합하도록 다수의 레지스터를 지원하고 있다. 실질적으로 자바 프로그램을 입력으로 받아 생성된 gasm 코드의 형태는 그림 6과 같다.

3. 중간 코드 변환기의 설계 및 구현

3.1 시스템 모델 및 개요

본 논문에서는 자바 바이트코드를 입력으로 받아 i386 어셈블리 코드를 생성하기 위해 오프라인 자바 컴파일러인 Jcc의 후단부 입력에 적합한 gasm 코드를 생성하는 중간 코드 변환기를 설계하고 구현한다.

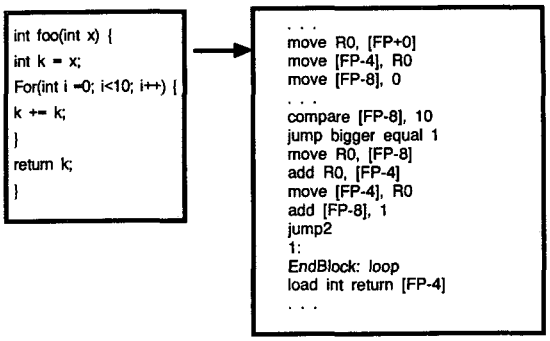

그림 6. 자바 프로그램에 대한 gasm 코드 예

바이트코드로부터 생성된 gasm 코드는 Jcc 후단부에서 제공하는 목적기계 정보(Machine Description; MD)와 라이브러리를 참조하여 i386 어셈블리 코드를 생성한다. 본 논문에서 최종적으로 구현하고자 하는 전체 시스템 모델은 그림 7과 같다.

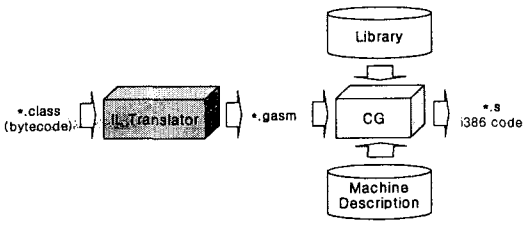

그림 7. 전체 시스템 구성도

본 논문에서 실질적으로 구현하고자 하는 중간 코드 변환기는 그림 8과 같은 구조를 가지고 있다.

바이트코드 추출기는 클래스 파일을 입력으로 받아 클래스 파일의 상수 풀 정보를 분석하여 바이트코

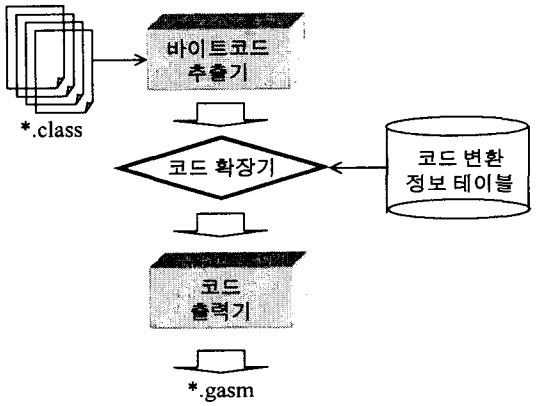

그림 8. 중간 코드 변환기 구조

드를 추출한다. 코드 변환기는 바이트코드에 대한 gasm 코드를 생성하는 핵심 부분으로서 코드 변환 테이블 정보를 참조하여 각 바이트코드에 대한 코드 확장(macro expansion) 기법을 이용한다. 따라서 하나의 바이트코드에 대해 동일한 기능을 갖는 gasm 코드로 변환된다. 코드 변환 테이블에는 중간 코드 변환 정보에 대한 실질적인 정보를 저장하는 부분으로서 바이트코드 특성을 고려하여 유사한 기능을 갖는 명령어 그룹으로 구성되어 있다.

본 논문에서 중간 코드 변환기를 사용하는 근본적인 이유는 *.class 파일의 Bytecode를 직접 i386, SPARC과 같은 목적기계 코드로 변환할 경우 기본적으로 스택 기반 중간 코드를 레지스터 기반 중간 코드로 변환시에 요구되는 다양한 고려 사항이 선행되어야 한다. 하지만 본 논문에서는 사용하고 있는 *.gasm 코드는 스택 기반 특성과 레지스터 기반 특성을 혼용하여 가지고 있으며 실질적인 레지스터 기반 코드로의 변환은 Jcc에서 이루어지고 있습니다. 따라서 중간 코드 변환기를 사용하여 *.class 파일을 쉽게 레지스터 기반 목적기계에서 실행할 수 있는 코드를 생성할 수 있는 장점을 갖는다. 다만, 중간 코드 변환에 소비되는 번역시간 부담이 발생되지만 응용 프로그램의 종류(반복 동작을 많이 수행하는 프로그램) 등에 따라 큰 문제가 되지 않는다.

3.2 중간 코드 변환기

중간 코드 변환기는 크게 바이트코드 추출기, 코드 확장기, 코드 출력기로 구성되어 있다. 바이트코드 추출기는 클래스 파일을 입력으로 받아 클래스 파일의 상수 풀 정보를 분석하여 바이트코드를 추출한다. 이러한 바이트코드 추출기는 Jasmin[12] 문법과 유사하게 바이트코드 명령어를 추출한다. 실질적인 바이트코드 추출기의 구현은 기존에 사용된 도구를 이용하였다. 이를 위해 javap 명령어 "-c" 옵션을 이용하였다. 클래스 파일로부터 바이트코드 명령어 집합을 출력하는 과정은 그림 9와 같다.

본 연구에서 적용하는 코드 확장 기법은 바이트코드에 대한 gasm 코드로 변환하는 루틴을 적용하는 방식이다. 따라서 하나의 바이트코드에 대해 해당되는 각각의 루틴을 호출하여 코드를 생성하므로 빠른 시간 내에 코드를 생성할 수 있는 장점을 갖는다. 하지만 생성된 코드의 질이 효율성이 떨어질 수 있으므로 최적화 동작이 추가적으로 수행되어야하는 단점을 갖는다. 바이트코드 추출기로부터 생성된 각각의 바이트코드에 대해 해당되는 루틴을 호출하는 형식은 그림 10과 같이 switch-case 문장을 이용하였다.

그림 9. 바이트코드 추출기

```
Bytecode_Opcode opcode = (Bytecode_Opcode) *code {
switch(opcode) {
        case NOP :
        . . .
        case ICONST_M1:  Push_Const(General,  32,
             integer, -1);
        . . .
        case LCONST_0:   Push_Const(General,  64,
             Long, 0);
}
```

그림 10. 바이트코드에 대한 코드 변환 루틴 호출

그림 10에서는 상수 값을 실질적으로 레지스터를 할당받아 저장하는 루틴을 호출하는 형식을 보여주고 있다. "ICONST_M1 : Push_Const(AllocReg_Type, Reg_Size, Type, Value);"에 대한 호출 형식에서 AllocReg 매개 변수는 저장되는 값을 위한 레지스터 종류를 결정하며 Reg_Size 매개 변수는 할당되는 레지스터 크기를 결정한다. 자료형에 따라서 할당되는 레지스터 크기가 32 비트 크기에 다양하게 결정될 수 있다. Type 매개 변수는 레지스터 저장되는 값의 자료형을 지정하며 실질적인 값은 Value 매개 변수에 저장된다. 바이트코드 각각에 대한 코드 변환 루틴은 바이트코드가 적재/저장 등과 같이 유사한 특성을 갖는 명령어 별로 그룹화 할 수 있다. 바이트코드에 대한 코드 변환 시에 코드 확장 기법의 적용은 전체 바이트코드에 대한 분석 없이 각각에 대해 매핑하므로 코드 질을 개선할 수 있는 다양한 최적화 동작이 요구된다. 코드 변환 정보 테이블의 구조는 각각 바이트코드에 대한 명령어 호출 루틴을 유사한 특성을 갖는 명령어 그룹을 관리하여 보다 효율적으로 코드 확장 루틴을 정보를 호출할 수 있도록 설계되어 있다.

3.3 바이트코드에 대한 gasm 코드 변환

바이트코드를 입력으로 받아 실질적으로 gasm 코드로 변환하는 과정은 바이트코드의 특성에 따라 Load/Store, 산술 연산, 배열 연산, 스택 관리, 자료형 변환 명령어 등으로 구분하여 단계적으로 변환하였다. 그림 11과 같은 바이트코드의 Load/Store 바이트코드를 중간 코드 변환기의 입력으로 사용하였다.

Java program	class file(bytecode)

public void op_LoadStore(){
int a, b, c, d, e;
a= -1;
b= 2;
c= 3;
d= 0;
e= 5;
b= c;
e= c;
}
	public void op_LoadStore() {
	//0 0:iconst_m1
	//1 1:istore_1
	//2 2:iconst_2
	//3 3:istore_2
	//4 4:iconst_3
	//5 5:istore_3
	//6 6:iconst_0
	//7 7:istore 4
	//8 9:iconst_5
	//9 10:istore 5
	//10 12:iload_3
	//11 13:istore_2
	//12 14:iload_3
	//13 15:istore 5
	//14 17:return
	}

그림 11. 중간 코드 변환기에 대한 Load/Store 바이트코드 입력

Load/Store 바이트코드에 대해 생성되는 gasm의 생성 결과는 그림 12와 같다.

Load/Store 바이트코드 입력에 대해 생성되는 gasm 중간 코드는 크게 3부분으로 구성되어 있다.

```
// 0:iconst_ml  //1  1:istore_1
(%g21800 = local int & _36, %i21801 = -1, *(0+%g21800, int) = %i21801);
//2  2:iconst_2  //3  3:istore_2
(%g21802 = local int & _37, %i21803 = 2, *(0 + %g21802, int) = %i21803);
//4  4:iconst_3  //5  5:istore_3
(%g21804 = local int & _38, %i21805 = 3, *(0 + %g21804, int) = %i21805);
//6  6:iconst_0  //7  7:istore  4
(%g21806 = local int & _39, %i21807 = 0, *(0 + %g21806, int) = %i21807);
//8  9:iconst_5  //9  10:istore  5
(%g21808 = local int & _40, %i21809 = 5, *(0 + %g21808, int) = %i21809);
//10  12:iload_3  //11  13:istore_2
(%g21810 = local int & _37, %i21811 = local int _38, *(0 + %g21810, int)
= %i21811) ;
//12  14:iload_3  //13  15:istore 5
(%g21812 = local int & _40, %i21813 = local int _38, *(0 + %g21812, int)
= %i21813 ) ;
```

그림 12. Load/Store 바이트코드 입력에 대한 gasm 출력

첫째, 변수 선언에 대해서 l-value에 해당되는 부분으로서 저장 공간을 확보하는 부분이다. 둘째, r-value에 해당되는 부분으로서 실제로 l-value에 저장되는 값을 가지고 있다. 마지막으로 r-value에 저장되어 있는 실질적인 값을 l-value에 할당하는 부분으로 구성되어 있다. 실질적으로 그림 6의 "e = 5"에 대한 "//8 9:iconst_5 //9 10:istore 5"와 같은 바이트코드에 대해 생성되는 gasm은 그림 13과 같이 동작된다.

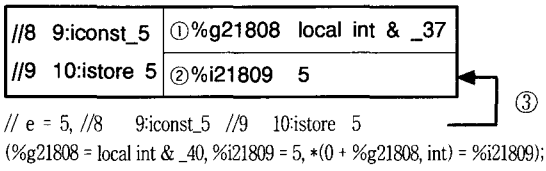

그림 13. gasm 코드의 수행 과정

본 연구의 최종적인 목적은 바이트코드로부터 i386 네이티브 코드 생성하기 위해 Jcc 후단부 입력에 적합한 gasm 코드생성이다. 따라서 논 논문에서 구현된 중간 코드 변환기를 이용하여 생성된 gasm 코드는 Jcc 후단부에 입력되어 변환된 코드의 정확성을 증명하였다. 본 논문에서 생성된 결과를 검증하기 위해 그림 11의 자바 프로그램을 입력으로 받아 먼저 JDK의 자바 컴파일러(javac)를 이용하여 클래스 파일을 생성한 후 다시 JDK의 자바 인터프리터(java)를 이용하여 실행하여 각 변수의 값을 출력하였다. 다음으로 JDK에서 생성된 클래스 파일을 다시 본 연구에서 구현한 중간 코드 변환기의의 입력으로 받아 gasm 파일을 생성하였다. 생성된 gasm 파일이 올바르며 JDK를 이용하여 생성된 결과와 일치도를 확인하기 위해 gasm 파일을 Jcc의 입력으로 사용하여 생성된 결과를 JDK 수행 결과와 비교하여 동일함을 확인하였다. 본 연구에서 구현된 중간 코드 변환기를 통해서 생성된 실행 결과는 아직 모든 바이트코드에 대한 변환이 이루어지지 않으므로 gasm으로부터 생성된 코드가 기존의 인터프리터 방식에 비해 실행 속도면에서 우수함을 비교하기 힘들다. 향후 보완 연구를 통해 모든 바이트코드에 대한 변환이 이루어지도록 하며 대형 프로그램에 대한 변환을 통해 다양한 성능 검사를 수행할 예정이다.

4. 결론 및 향후 연구

자바 프로그래밍 언어는 인터넷 및 분산 환경 시

스템에서 효과적으로 응용 프로그램을 작성할 수 있도록 설계된 언어로서 객체지향 페러다임 특성 및 다양한 개발 환경을 지원하고 있다. 자바 언어 시스템에서는 자바 언어를 플랫폼에 독립적으로 실행시키기 위해 가상 기계 코드인 바이트코드를 사용하며 자바 기상 기계의 인터프리터를 이용하여 실행하고 있다. 이로 인해 웹 브라우저에서 실행되는 작은 크기의 자바 응용 프로그램 수행에는 실행 속도 문제가 중요한 요소가 아니지만 대형 프로그램의 수행에는 실행 속도가 현저히 저하되며 또한 C/C++와 같은 기존 프로그램의 실행 속도에 비해 매우 느린 단점을 지니고 있다. 특히, 전통적인 컴파일 방법을 사용하여 컴파일 과정에서 중간 코드인 바이트코드를 특정 프로세서에 수행될 수 있는 목적 코드로 바꾸는 후단부를 사용하는 방법에 대한 다양한 연구가 진행되고 있다.

본 연구에서도 자바 응용 프로그램의 실행 속도의 개선을 위해 바이트코드로부터 직접 i386 코드를 생성하는 네이티브 코드 생성 시스템을 위한 중간 코드 변환기를 설계하고 구현한다. 중간 코드 변환기는 자바 언어의 중간 코드인 *.class 파일을 입력으로 받아 레지스터 기반의 중간 코드로 변환한다. 이를 위해 Jcc에서 제안된 레지스터 기반의 중간 코드를 사용한다. 따라서 본 시스템에서는 바이트코드의 집합인 *.class 파일을 입력으로 받아 레지스터 기반의 중간 코드인 *.gasm으로 변환하는 중간 코드 변환기를 구현하였다.

중간 코드 변환기는 크게 바이트코드 추출기, 중간 코드 변환기, 코드 변환 테이블로 구성되어 있다. 바이트코드 추출기는 클래스 파일을 입력으로 받아 클래스 파일의 상수 풀 정보를 분석하여 바이트코드를 추출한다. 코드 변환기는 바이트코드에 대한 gasm 코드를 생성하는 핵심 부분으로서 코드 변환 테이블 정보를 참조하여 각 바이트코드에 대한 1:1 코드 확장(macro expansion) 기법을 이용한다. 따라서 하나의 바이트코드에 대해 동일한 기능을 갖는 gasm 코드로 변환된다. 코드 변환 테이블에는 중간 코드 변환에 대한 실질적인 정보를 저장하는 부분으로서 바이트코드 특성을 고려하여 유사한 기능을 갖는 명령어 그룹으로 구성되어 있다. 마지막으로 구현된 중간 코드 변환기를 이용하여 생성된 결과를 검증하기 위해 *.class 파일을 JDK의 인터프리터를 수행한 결과와 *.class로부터 변환된 *.gasm 코드를 Jcc 후단부 입력으로 사용하여 최종적인 결과 값의 일치도를 검증하였다.

현재 본 연구에서는 생성된 gasm 코드에 대한 검증 및 실행 결과를 확인하기 위해 검증된 Jcc의 후단부를 이용하고 있는 단점과 생성된 gasm 코드에 대한 효율성에 대한 문제점을 가지고 있다. 따라서 앞으로 Jcc 후단부에 상응하는 네이티브 코드 생성 시스템의 개발과 양질의 gasm 코드 생성을 위해 패턴 기술을 통한 코드 질 향상을 위해 연구를 진행할 예정이다. 본 연구의 가치는 스택 기반 바이트코드에 대해 RTL과 유사한 레지스터 기반 중간 코드인 gasm으로 변환을 통해 네이티브 코드 생성 시스템 구축을 위한 기반 연구로 활용될 수 있다.

참 고 문 헌

[1] A. krall and R. Grafl, CACAO : A 64 bit Java VM Just-in-Time Compiler, Concurrency: practice and experience, 1997. http://www.complang.tuwien.ac.at/~andi.

[2] Alfred V. Aho, Mahadevan Ganapathi, Steven W. K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," ACM TOPLAS, Vol. 11, No. 4., pp.491-516, Oct., 1989.

[3] Hans van Staveren, "The table driven code generator from ACK 2nd. Revision," report-81, Netherlands Vrije Universiteit, 1989.

[4] Jonathan Meyer, Jasmin Assembler, http://www.cat.nyu.edu/meyer/jasmin.

[5] Jon Meyer and Troy Downing, JAVA Virtual Machine, O'REYLLY, 1997.

[6] Karen A. Lemone, Design of Compilers : Techniques of Programming Language Translation, CRC Press, 1992.

[7] Ken Arnold and James Gosling, The Java Programming Language, Sun Microsystems, 1996.

[8] Mahadevan Ganapathi, Charles N. Fischer, John L. Hennessy, "Retargetable Compiler Code Generation", ACM Computing Surveys, Vol. 14, No. 4, pp.573-592, Dec., 1982.

[9] R. G. G. Cattell, "Automatic Derivation of Code Generators from Machine Descriptions," ACM TOPLAS, Vol. 2, No. 2, pp.173-190. Apr., 1980.

[10] Ronald Veldema, Jcc, A Native Java compiler, Vrije Universiteit Amsterdam, July, 1998.

[11] Susan L. Graham, "Table-Driven Code Generation", IEEE Computer, Vol.13, No.8, pp.25-34, Aug., 1980.

[12] Wen-mei W. Hwu, "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results", The proceeding of the 29th Annual International Symposium on Microarchitecture. Dec., 1996.

고 광 만

1991년 원광대학교 컴퓨터공학과 졸업(학사)

1993년 동국대학교 대학원 컴퓨터공학과 졸업(석사)

1998년 동국대학교 대학원 컴퓨터공학과 졸업(박사)

1998년 3월~2001년 08월 광주여자대학교 정보통신학부 교수

2001년 9월~현재 상지대학교 컴퓨터정보공학부 교수

관심분야 : 프로그래밍언어론, 컴파일러구성론, 모바일 컴퓨팅 etc.

E-mail : kkman@mail.sangji.ac.kr