바이트코드로부터 코드 확장 기법을 이용한 중간 언어 변환기의 설계 및 구현

고 광 만
상지대학교 컴퓨터정보공학부
kkman@mail.sangji.ac.kr

Design and Implementation of Intermediate Language Translator using Code Expansion Technique from Bytecode

Kwang-Man Ko
School of Computer, Information and Communication, SangJi University

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 개선하기 위해 전통적인 컴파일 방법을 사용하여 바이트코드를 특정 프로세서에 적합한 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 언어 변환기를 설계하고 구현한다. 중간 언어 변환기는 자바 언어의 중간 언어인 *.class 파일을 입력으로 받아 레지스터 기반의 중간 언어로 변환한다.

1. 서 론

자바 프로그래밍 언어는 인터넷 및 분산 환경 시스템에서 효과적으로 응용 프로그램을 작성할 수 있도록 설계된 언어로서 객체지향 패러다임 특성 및 다양한 개발 환경을 지원하고 있다[10][11]. 자바 언어 시스템에서는 자바 언어를 플랫폼에 독립적으로 실행시키기 위해 가상 기계 코드인 바이트코드를 사용하며 자바 가상 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 위해 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. 또한 자바 응용 프로그램의 실행 속도 개선을 위해 SUN에서도 기존의 JVM에서 인터프리터를 이용하는 방식과 병행하여 바이트코드를 특정 프로세서의 네이티브 코드로 컴파일하여 수행하는 방법을 사용하고 있다[11].

자바 언어의 중간 언어인 바이트코드는 플랫폼 독립적인 특성으로서 1960년대에 제안된 UNCOL과 유사한 개념이며 바이트코드를 실행하기 위한 시스템인 JVM은 P-코드의 영향을 받았다. 또한 바이트코드는 ACK의 중간 언어인 EM 코드와 유사하게 스택 기반 중간 언어 방식이며 스택 기반의 중간 언어를 실질적인 특정 기계의 레지스터 기반 목적 코드로 변환하는데 많은 연구가 진행되어 왔다[6][7]. 자바 언어를 실행하기 위한 다양한 시도로서 JVM의 인터프리터 방식과 별도로 JIT 컴파일러의 구현이 연구소 및 산업계에서 진행되어 왔다. 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 언어로 변환된 후 다시 레지스터 기반 중간 언어로부터 Alpha 프로세서를 위한 네이티브 코드로 변환된다. .NET 컴파일러[7]은 바이트코드로부터 네

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

이티브 코드를 생성하는 최적화 컴파일러로서 IMPACT에서 제안된 시스템이다. NET 컴파일러에서도 스택 기반의 바이트코드를 특정 프로세서에 적합한 네이티브 코드를 생성하기 위해 고안된 레지스터 기반의 중간 언어를 사용하고 있다. Jcc[9]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 언어인 *.gasm 파일을 생성하며 후단부에서는 *.gasm 파일을 입력으로 받아 특정 기계에 대한 어셈블리 코드를 생성한다.

본 연구에서도 자바 응용 프로그램의 실행 속도 개선을 위해 바이트코드로부터 직접 i386 코드를 생성하는 네이티브 코드 생성 시스템을 위한 중간 언어 변환기를 설계하고 구현한다. 중간 언어 변환기는 자바 언어의 중간 언어인 *.class 파일을 입력으로 받아 레지스터 기반의 중간 언어로 변환한다. 이를 위해 Jcc에서 제안된 레지스터 기반의 중간 언어를 사용한다. 변환된 *.gasm를 입력으로 받아 i386 코드를 생성하는 네이티브 코드 생성 시스템은 Jcc의 후단부를 이용하여 변환된 코드를 검증한다.

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 언어로 변환된 후 다시 레지스터 기반 중간 언어으로부터 Alpha 프로세서를 위한 네이티브 코드로 변환된다. 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.

NET 컴파일러[7]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 언어를 사용하고 있다. 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.

Jcc[9]는 오프라인 상태에서 자바 언어를 입력으로 받아 네이티브 코드를 생성하는 컴파일러이다. 이식이 용이한 컴파일러 제작을 위해 RTL에 기반을 둔 레지스터 기반 중간 언어(*.gasm)를 사용하고 있으며 중간 언어에 대한 다양한 최적화 동작을 수행한다. Jcc의 전단부에서는 Jade라는 파서 생성기를 구현하여 파서를 생성하며 생성된 어휘 분석기와 파서를 참조하여 자바 언어의 입력에 대해 *.gasm 코드를 생성한다.

2.2 gasm 중간 언어

본 연구에서 스택 기반 바이트코드를 입력으로 받아 중간 언어 변환기에 의해 생성되는 gasm 코드는 자바 언어를 입력으로 받아 네이티브 코드 생성을 오프라인 자바 컴파일러인 Jcc에서 사용하는 레지스터 기반 중간 언어이다. 이러한 gasm은 move, add, compare 등과 같은 명령어를 가지고 있으며 개략적인 형식은 그림1과 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 1) gasm의 중간 언어

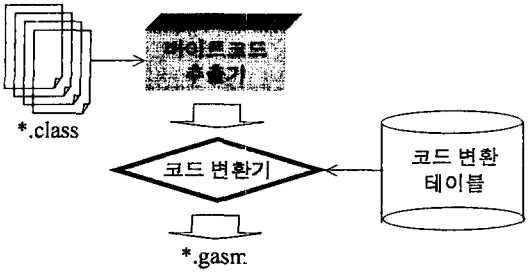
일반적인 산술 명령, 이동 명령 등은 스택 대신에 레지스터에서 동작되며 조건문의 실행과 반복문 등의 실행을 위해 플래그 레지스터를 사용한다. gasm에서 레지스터는 네 개의 범용 레지스터와 한 개의 기본 주소 레지스터를 명시적으로 지원하고 있으며 실질적으로는 네이티브 코드를 생성하고자 하는 목적기계에 적합하도록 다수의 레지스터를 지원하고 있다.

3. 중간 언어 변환기의 설계 및 구현

3.1 중간 언어 변환기 구조

본 논문에서는 자바 바이트코드를 입력으로 받아 i386 코드를 생성하기 위해 오프라인 자바 컴파일러인 Jcc의 후단부 입력에 적합한 gasm 코드를 생성하는 중간 코드 변환기를 설계하고 구현하며 그림 2와 같은 구조를 가지고있다. 바이트코드로부터 생성된 gasm 코드는 Jcc 후단부에서 제공하는 기계 언어(Machine Description; MD) 정보와 라이브러리를 참조하여 i386 코드를 생성한다.

바이트코드 추출기는 클래스 파일을 입력으로 받아 클래스 파일의 상수 풀 정보를 분석하여 바이트코드를 추출한다. 코드 변환기는 바이트코드에 대한 gasm 코드를



(그림 2) 중간 언어 변환기 구조

생성하는 핵심 부분으로서 코드 변환 테이블 정보를 참조하여 각 바이트코드에 대해 대응하는 코드를 생성한다. 따라서 하나의 바이트코드에 대해 동일한 기능을 갖는 gasm 코드로 변환된다. 코드 변환 테이블에는 중간 언어 변환 정보에 대한 실질적인 정보를 저장하는 부분으로서 바이트코드 특성을 고려하여 유사한 기능을 갖는 명령어 그룹으로 구성되어 있다.

3.2 바이트코드에 대한 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 9와 같은 바이트코드의 Load/Store 바이트코드를 중간 언어 변환기의 입력으로 사용하였다. Load/Store 바이트코드에 대해 생성되는 gasm의 생성 결과는 그림 3과 같다.

```
// b = 2, //2    2:iconst_2  //3    3:istore_2
(%g21802 = local int & _37, %i21803 = 2, *(0 + %g21802, int) =
%i21803);
// c = 3, //4    4:iconst_3  //5    5:istore_3
(%g21804 = local int & _38, %i21805 = 3, *(0 + %g21804, int) =
%i21805);
// b = c, //10    12:iload_3  //11    13:istore_2
(%g21810 = local int & _37, %i21811 = local int _38, *(0 + %g21810,
int) = %i21811) ;
```

(그림 3) Load/Store 입력에 대한 gasm 출력

Load/Store 바이트코드 입력에 대해 생성되는 gasm 중간 언어는 크게 세부분으로 구성되어 있다. 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.

3.3 실험 결과 및 분석

본 연구에서 생성된 결과를 검증하기 위해 *.class 파일을 JDK의 인터프리터를 수행한 결과와 *.class로부터 변환된 *.gasm 코드를 Jcc 후단부 입력으로 사용하여 최종적인 결과 값의 일치도를 검증하였다. 230여 개의 바이트코드에 대한 중간 언어 변환을 위해 가장 기본적인 Load/Store 명령어, 산술 및 논리 연산 명령어, 스택 관리 명령어, 자료형 변환 등에 관련된 명령어에 대한 코드 변환 테이블을 현재까지 완성하여 실행 결과를 확인할 수 있다.

4. 결론 및 향후 연구

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 언어 변환기를 설계하고 구현하였다. 중간 언어 변환기는 *.class 파일을 입력으로 받아 레지스터 기반의 중간 언어로 변환한다. 이를 위해 Jcc에서 제안된 레지스터 기반의 중간 언어를 사용하였다. 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.

5. 참고 문헌

[1] 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.

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

[3] 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.

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

[5] Ronald Veldema, Jcc, a native Java compiler, Vrije Universiteit Amsterdam, July, 1998.

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

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