2001년 한국정보처리학회 춘계 학술발표논문집 제8권 제1호

Bytecode로부터 목적 기계 코드 생성 규칙 기술에 관한 연구

고 광 만
광주여자대학교 정보통신학부
e-mail:kkman@www.kwu.ac.kr

A Study on the Target Code Generation Rule Description from Bytecode

Ko Kwang-Man
Division of Information Communication,
Kwang-Ju Women's University

요 약

컴파일러 후단부 개발시 중간 코드로부터 목적기계 코드를 생성하기 위해서는 각각의 중간 코드 명령어를 목적기계 코드로 치환하는 방법과 다양한 중간 코드 패턴에 대한 목적기계 코드 생성 규칙을 기술하는 방법으로 구분된다. 특히, 컴파일러 후단부 전체를 재구성하지 않고 중간 코드로부터 목적기계 코드를 생성하는 정형화된 규칙을 이용하면 다양한 목적기계 코드를 효율적으로 생성할 수 있다.

본 논문은 Bytecode로부터 정형화된 코드 생성 규칙을 이용하여 Pentium기계에 대한 코드 생성이 가능하도록 코드 생성 규칙 기술 모델을 제시하며 실질적으로 목적기계 코드 생성시에 참조 가능한 정보를 생성하는 코드-생성기 생성기를 연구한다. 본 연구를 통해서 Bytecode로부터 다양한 목적기계 코드를 생성할 수 있는 정형화된 코드 생성 규칙 기술 방법을 얻을 수 있다.

1. 서 론

재목적 코드 생성 시스템은 컴파일러 후단부에서 새로운 목적기계에 대해 원시 프로그램의 중간 코드를 목적기계 코드로 바꾸는 과정을 정형화된 방법을 통하여 자동적으로 구성하는 것이다. 특히, 목적기계 코드 생성 단계에서는 중간 코드로부터 최적의 목적기계 코드를 생성하기 위해 중간 코드의 각각의 명령어를 목적기계 코드로 확장하는 기법과 중간 코드에 존재하는 다양한 패턴에 대해 패턴 매칭 알고리즘을 적용하여 목적기계 코드를 생성하는 기법이 제시되고 있다[1].

컴파일러 자동화 도구인 ACK에서는 EM 중간 코드에 대해 다양한 목적기계 코드를 생성하기 위해 목적기계 표현 테이블에 목적기계 정보 및 코드 생성 규칙을 기술하고 있다. 코드-생성기 생성기는 이러한 목적기계 표현 테이블을 입력으로 받아 실질적으로 목적코드 생성시에 필요한 정보를 생성한다. 하지만 ACK 목적기계 표현 테이블은 양질의 목적기계 코드 생성을 위해 보다 정련된 방법으로 코드 생성 규칙을 기술할 수 있도록 보완되어야 하며 다양한 목적기계 특성을 기술할 수 있도록 재구성되어야 한다. 이를 위해 본 논문에서는 다양한 목적기계에 대한 양질의 목적기계 코드 생성이 가능하도록 Java 언어의 중간 코드인 Bytecode에 대한 목적 코드 생성 규칙을 정형화된 방법으로 표현하는 기법에 관한 연구이다. 또한 정형화된 기법으로 기술된 코드 생성 규칙을 입력으로 받아 실질적으로 코드 생성시에 필요한 정보를 생성하는 코드-생성기 생성기를 개발한다[1][2].

본 논문의 구성은 2장에서 ACK를 기반으로 하여 코드 생성 규칙 기술 방법에 대해 고찰하며 3장에서는 본 연구에서 제안하는 Bytecode로부터 목적기계 코드 생성 규칙 기술 기법 및 코드-생성기 생성기에 대해 설명한다. 마지막으로 4장에서는 본 연구에 대한 결론 및 향후 연구 방향에 대해 기술한다.

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

2. 코드 생성 규칙 기술 기법

2.1 ACK 코드 확장기

코드 확장기를 이용한 목적코드 생성 방법에서는 중간 코드 각 명령어에 대해 특정 기계의 어셈블리 코드를 생성할 수 있는 루틴을 이용하여 어셈블리 코드를 생성한 후에 어셈블리 코드를 다시 실행 형태로 변환해주는 라이브러리 함수로 구성되어 있다. 이러한 코드 확장기를 이용한 방식에서는 ACK의 중간 코드인 EM 코드 각각에 대해 직접 목적기계 코드로 변환하므로 목적 코드를 생성하는데 필요한 시간을 단축할 수 있다. 하지만 생성된 목적기계의 코드 질을 개선하기 위한 부가적인 동작이 수행되어야 하는 문제점을 가지고 있다[1].

2.2 ACK 코드 생성기-생성기

ACK에서는 정형화된 방법으로 중간 코드에 대한 목적기계 코드 생성하기 위해 목적기계 표현 테이블에 코드 생성 규칙을 기술한다. 즉, 목적기계 코드를 생성하는데 핵심적인 부분인 중간 코드를 특정 목적기계의 어셈블리 코드로 변환시키는데 필요한 모든 정보를 기술하게 된다. 따라서 양질의 목적기계 코드 생성을 위한 정보는 목적기계 표현 테이블에 모두 기술되어야 한다. 목적기계 표현 테이블은 크게 11개의 절(section)로 구성되어 있으며 각 절의 정보를 이용하여 마지막에 실질적인 중간 코드 패턴에 대한 목적기계 코드 생성 규칙을 기술한다.

3. Bytecode 및 Pentium 코드

3.1 Bytecode 및 클래스 파일

Bytecode는 자바 프로그래밍 환경에서 지원하는 중간 언어로서 이 기종간의 실행 환경에 적합하도록 설계된 스택 기반의 가상 기계 코드이다. 자바 컴파일러는 자바 프로그램을 입력으로 받아 자바 가상 기계의 명령어 집합인 Bytecode를 생성한다. 중간 코드 형태인 Bytecode는 이 기종간의 실행환경에 적합하도록 설계되어 이식성과 유연성이 높으며 인터프리터 방식과 JIT(Just-In-Time) 컴파일러 방식에 의해 실행되어 진다. Bytecode의 형태는 연산 코드 부분과 연산 코드에 따라 여러 개의 피연산자 부분으로 구성되어 있다. Bytecode는 어떠한 기계에서도 해석되기 쉽고, 성능 향상 요구가 있을 때 동적으로 기계 명령어로 번역되도록 설계되어 있다. 현재 Bytecode 명령어의 개수는 230개이며 이는 202개의 일반 명령어 집합과 25개의 _quick 명령어, 3개의 예약 명령어로 구성된다[3].

자바 컴파일러에 의해 생성되는 클래스 파일은 8비트 크기를 갖는 스트림으로 구성되며 하나의 ClassFile 구조체로 표현된다.

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

[그림 1] 클래스 파일 구조체

클래스 파일에서 상수 기억 장소인 constant_pool은 Class, Fieldref, Methodref와 같은 타입을 가지고 있다. 속성 테이블인 attributes 필드에는 클래스 파일에서 미리 정의된 Sourcefile, Costantvalue, Code, Exceptions, Linenumbertable, Localvariabletable에 대한 구조체로 구성되어 있다. method_info 타입의 구조체에는 클래스나 인터페이스의 메소드에 대해 기술하고 있으며 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 Bytecode가 저장되어 있다. Code 속성을 갖는 구조체는 그림 2와 같다.

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

[그림 2] 코드 속성 구조체

Bytecode의 피연산자는 컴파일 시간에 결정되며 실행 시간에 실질적으로 계산되어 결과값이 스택에 저장되는 스택 지향적 구조를 가진다. Bytecode의 데이터 형에는 정수형에 속하는 byte, short, int, long, char 형이 있고, 실수형에 속하는 float, double 형이 있다.

3.2 Pentium 코드

펜티엄 프로세서 칩은 앞에서도 설명한 바와 같이 32비트 프로세서 구조에 64비트의 데이터 버스 구조로 구성되어 있으며, 이와 같은 구조는 기억 장치로부터 판독한 64비트의 자료가 펜티엄 프로세서에 의해 직접 사용되는 것이 아니라 내부에 집적되어 있는 캐시와의 데이터 전송에 사용하기 위함이다. 따라서 펜티엄 칩의 내부 버스와 외부 버스는 내부 캐시를 통해 통신을 한다는 것을 기억하기 바라며, 이와 같이 프로세서 내부 구조는 32비트로, 그리고 외부 데이터 버스 구조는 64비트로 이루어진 대표적인 마이크로프로세서 칩으로는 인텔 i860 RISC 프로세서와 모토롤러 88110 RISC 프로세서들이 있다. 펜티엄 프로세서의 내부 데이터 버스는 128비트와 256비트로 구성되어 있으며, 이는 데이터와 명령어 코드를 보다 고속으로 전송시키기 위함이다.

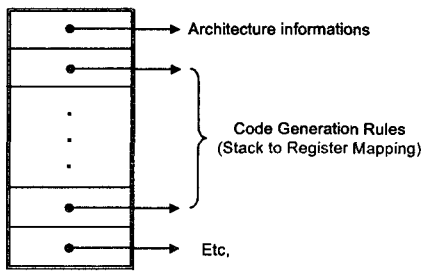
펜티엄 프로세서는 파이프라인 구조의 컴퓨터에서 처리 속도를 떨어뜨리게 되는 분기 문제를 해결하는 분기 예측이 가능하며, 파이프라인 구조의 컴퓨터에서 분기 예측을 수행하지 않을 경우 기억 장치로부터 미리 인출된 다수의 명령어들은 명령어 큐에 미리 저장되며, 현재 해독된 명령어가 분기 명령어일 경우 명령어 큐는 리셋되어 명령어 큐에 저장된, 즉 분기 명령어를 해독하기 전에 인출된 명령어들은 모두 명령어 큐로부터 제거되고 분기된 기억 장치 주소로부터 다시 명령어들을 미리 인출하여 이들 명령어들을 명령어 큐에 저장하여야 한다. 따라서 분기 예측을 수행할 수 있는 파이프라인 구조의 컴퓨터는 명령어 큐를 리셋시키지 않고 프로그램의 환경이 다른 위치로 용이하게 분기할 수 있다. 따라서 분기 예측이란 실행되어야 하는 명령어 세트 가운데 가장 있음직한 명령어 세트를 미리 결정함으로써 해결된다.

펜티엄 프로세서에 존재하는 레지스터는 EAX, EBX, ECX, EDX와 ESI, EDI 및 EBP 레지스터들이다. 범용 레지스터 가운데 데이터 레지스터인 EAX, EBX, ECX와 EDX 레지스터들은 모두 32비트 데이터를 저장할 수 있을 뿐만 아니라 바이트 또는 워드 단위의 데이터들도 저장할 수 있지만, ESI, EDI와 EBP 레지스터들은 16비트 데이터와 32비트 더블워드 데이터만을 저장할 수 있다. 또한 기억 장치의 세그먼트 영역의 시작 주소를 가리키는 6개의 16비트 세그먼트 레지스터가 있으며, 이는 CS, DS, SS, ES와 GS 레지스터이고 명령어 포인터와 스택 포인터인 32비트의 EIP와 ESP, 그리고 32비트 플래그 레지스터 EFLAG 레지스터가 있다.

4. 코드 생성 규칙 기술

4.1 코드 생성 규칙 모델

본 연구에서 제시하는 정형화된 코드 생성 규칙 기술 방법은 컴파일러 자동화 도구인 ACK의 목적 기계 표현 테이블 표현, IMPACT 및 Zephyr 프로젝트를 기반으로 하여 그림 3과 같이 구성되어 있다 [5].



[그림 3] 코드 생성 규칙 모델

중간 코드에 대한 목적기계 코드 생성 규칙을 기술하기 위해서는 먼저 특정 목적기계의 정보를 정형화된 방법으로 기술해야 한다. 목적기계 정보에는 레지스터의 종류 및 특성 등이 기술된다. 실질적인 코드 생성 규칙 부분에서는 목적기계 정보를 참조하여 목적기계 명령어의 특성 및 명령어가 갖는 피연산자 등을 기술한다. 또한 중간 코드 패턴에 대한 목적 코드로의 변환 규칙을 기술하며 올바른 변환을 위해 다양한 변환 규칙을 기술한다. 본 연구에서 설계한 Bytecode에 대한 목적기계 코드 생성 규칙 문법은 그림 4와 같다.

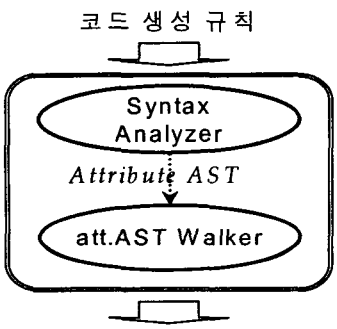
```
Table  ::= (RULE)*
RULE  ::= C_instr(COND_SEQUENCE|SIMPLE)
COND_SEQUENCE ::= (condition SIMPLE)*
"default" SIMPLE
```

```
SIMPLE    ::= "==>" ACTION_LIST
ACTION_LIST ::= [ACTION( ; ACTION)*] .
ACTION    ::= AS_INSTR | function_call
AS_INSTR  ::= """[label":"] [INSTR]"""
INSTR     ::= mnemonic[operand( , operand)*]
. . .
```

[그림 4] 목적기계 코드 생성 규칙 문법

4.2 코드-생성기 생성기

Byetcode로부터 목적기계 코드 생성 규칙을 정형화하게 기술할 수 있도록 작성된 문법 규칙은 그림 5와 같은 과정을 거쳐 실질적으로 Bytecode에 대한 Pentium 목적기계 코드 생성시에 참조될 수 있도록 정보를 생성한다.



[그림 5] 코드-생성기 생성기

본 연구에서 기술한 코드 생성 규칙을 이용하여 Bytecode로부터 Pentium 코드 생성을 위한 코드 생성 규칙은 flex와 bison을 이용하여 문법 검사를 수행하게 되며 올바른 코드 생성 규칙에 대해서는 문법-지시적 변환 방법을 이용하여 속성을 가진 구문 트리(attributed AST)를 생성하게 된다. 구문 트리 순회기(AST walker)는 작성된 속성을 가진 구문 트리를 순회하면서 코드 생성시에 필요한 정보를 생성한다.

코드-생성기 생성기는 본 연구에서 제시한 모델중에서 구문 분석 후 구문 트리를 순회하면서 코드 생성 정보를 생성하는 부분이다. 코드-생성기 생성기에 의해 출력되는 구조는 다음 단계인 코드 생성시에 참조되는 정보를 제공하는 부분이므로 코드 생성 기법과 밀접한 구조를 가지고 있다.

5. 결론 및 향후 연구 방향

컴파일러 후단부 개발시 중간 코드로부터 목적기계 코드를 생성하기 위해서는 각각의 중간 코드 명령어를 목적기계 코드로 치환하는 방법과 다양한 중간 코드 패턴에 대한 목적기계 코드 생성 규칙을 기술하는 방법으로 구분된다. 특히, 컴파일러 후단부 전체를 재구성하지 않고 중간 코드로부터 목적기계 코드를 생성하는 정형화된 규칙을 이용하면 다양한 목적기계 코드를 효율적으로 생성할 수 있다.

본 논문은 Bytecode로부터 정형화된 코드 생성 규칙을 이용하여 Pentium기계에 대한 코드 생성이 가능하도록 코드 생성 규칙 기술 모델을 제시하고 구현하였다. 또한 실질적으로 목적기계 코드 생성시에 참조 가능한 정보를 생성하는 코드-생성기 생성기 모델을 제시하고 구현하였다. 본 연구를 통해서 Bytecode로부터 다양한 목적기계 코드를 생성할 수 있는 정형화된 코드 생성 규칙 기술 방법을 얻을 수 있으며 앞으로 많은 보완 연구를 통해 양질의 목적기계 코드 생성을 위한 코드 생성 규칙을 추가할 예정이다.

참고문헌

[1] Frans Kaasheok, Koen Langendeon, "The Code Expander Generator", Dept. of Math. and Computer Science, Vrije Universiteit, 1990.

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

[3] Christoph M. Hoffmann and Michael J. O'Donnell, "Pattern Matching in Trees," Journal of the Association for Computing Machinery, Vol. 29, No.1, pp. 68-95, Jan., 1982.

[4] Jon Meyer, Troy Downing, "Java Virtual Machine", O'REILLY, 1997.

[5] Wen-Mei W. Hwu, "IMPACT: An Architecture Framework for Multiple-Instruction Issue Processors", Proceeding of the 18th International Symposium on Computer Architecture, pp. 266-275, 1991.

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