재목적 코드 생성 기법을 이용한 자바 Bytecode에서 SPARC 코드로의 번역

(Translating Java Bytecode to SPARC Code using Retargetable Code Generating Techniques)

오 세 만 † 정 찬 성 ††
(Seman Oh) (Chansung Jung)

요 약 자바 프로그래밍 언어는 인터넷 및 분산 네트워크 환경에서 효과적으로 수행될 수 있도록 설계된 언어이다. 그러나 각 플랫폼에서 인터프리터 방식으로 실행된다는 단점을 가지고 있기 때문에, 자바 프로그램을 효율적으로 실행하기 위해서는 Bytecode를 목적 기계 코드인 SPARC 코드로 변환하는 코드 생성 시스템이 개발되어야 한다.

본 논문에서는 재목적 코드 생성 기법을 이용하여 Bytecode를 SPARC 코드로 변환하는 코드 생성 시스템을 구현하였다. 이를 위해 Bytecode로부터 SPARC 코드 생성 규칙을 기술한 Bytecode 테이블을 작성하였고, 클래스 파일을 입력으로 받아 Bytecoed를 코드 확장시에 적합한 형태로 변환하는 정보추출기를 구현하였다. 정보추출기가 Bytecode 명령어의 피연산자에 대한 상수 기억장소의 엔트리를 결정한 후, 코드 확장기가 변경된 Bytecode를 Bytecode 테이블에 따라 SPARC 코드로 변환한다. 또한, 재목적 코드 생성 시스템은 다양한 목적 기계 코드를 생성하기 위해 체계적으로 재구성될 수 있다.

Abstract Java programming language is designed to run effectively on internet and distributed network environments. However, because it has a deficit to be executed by the interpreter method on each platform, to execute Java programs efficiently the code generation system which transforms Bytecode into SPARC code as target machine code must be developed.

In this paper, we implement a code generation system which translates Bytecode into SPARC code using the retargetable code generating techniques. For the sake of code expander, we wrote a Bytecode table describing a rule of SPARC code generation from Bytecode, and implemented the information extractor transforming Bytecode to suitable form during expanding of source code from class file. The information extractor determines constant pool entry of each Bytecode instruction operand and then the code expander translates the Bytecode into SPARC code accoring to the Bytecode table. Also, the retargetable code generation system can be systematically reconfigured to generate code for a variety of distinct target computers.

1. 서 론

자바 프로그래밍 언어는 인터넷 및 분산 환경 시스템에서 효과적으로 응용 프로그램을 작성할 수 있도록 설계된 언어로서 이 기종간의 실행 환경에 적합하도록 설계된 Bytecode를 중간 언어로 사용한다. 자바의 전단부(Front-end)는 자바 소스 프로그램을 입력으로 받아 중간 언어인 Bytecode를 가지고 있는 클래스 파일을 생성한다.

Bytecode를 실행하는 방법은 그림 1과 같이 인터프리터와 JIT를 혼용하는 방식과 자바Back-end 방식으로 구분된다. 과거에는 Bytecode를 실행할 때 인터프리터 방식만을 사용하였으나, 요즘은 실행 속도를 개선하기 위해 JIT와 결합하여 실행된다.

자바 가상 기계는 클래스 파일이라는 특별한 형식을 입력으로 받아 실행하고, 클래스 파일은 자바 가상 기계

· 본 연구는 한국과학재단의 핵심전문연구(과제번호:971-0903-025-2)지원에 의한 것임.
† 종신회원 : 동국대학교 컴퓨터공학과 교수
smoh@dgu.ac.kr
†† 비 회 원 : 동국대학교 컴퓨터공학과
csjung@dgu.ac.kr
논문접수 : 1999년 8월 10일
심사완료 : 2000년 5월 3일

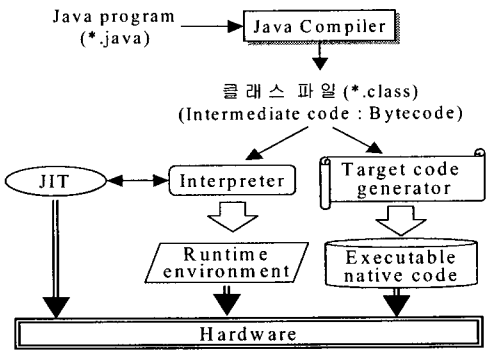

그림 1 자바의 실행 과정

명령어(Bytecode)와 심벌 테이블, 그 외에 부수적인 정보를 가지고 있다. 또한, 자바 가상 기계에서 보안성을 위하여 클래스 파일의 코드가 엄격한 형식과 구조적인 제약을 가지고 있도록 하고 있다[5][9].

Bytecode는 실행될 연산자를 기술하기 위한 한 바이트 크기의 opcode와 연산자에 의해 사용되어질 인자나 데이타를 기술하는 피 연산자들로 구성되어진다. 이러한 Bytecode는 이 기종 기계에서 공통적으로 실행될 수 있다는 장점을 갖지만 각 플랫폼에서 인터프리터 방식으로 실행되므로 실행 속도면에서 단점을 가지고 있다. 인터프리터 방식으로 실행하는 방식보다 목적 코드로 변환하여 실행하는 방식이 실행 속도가 훨씬 향상된다. 따라서 자바 프로그램을 효율적으로 실행하기 위해 인터프리터와 JIT 방식을 혼용함으로써 이 기종 기계에서 공통적으로 실행되며 인터프리터만을 사용하는 시스템에 비해 실행 속도가 향상된다[11]. 이때 JIT 방식은 중간 코드를 목적 코드로 변환하는 과정이 필요하다. 또한, 자바의 후단부를 이용하는 방식도 중간 코드를 목적 코드로 변환하여 실행하므로 변환 과정이 요구된다.

컴파일러 자동화 도구인 ACK (Amsterdam Compiler Kit)는 컴파일러의 후단부를 자동화하기 위한 도구의 하나로서 이식성과 재목적성이 매우 높은 컴파일러를 만들기 위해 80년대 초 네델란드의 Amsterdam에 위치한 vrije 대학의 Andrew S. Tanenbaum을 중심으로 개발되었다. 이러한 ACK에서 지원되는 재목적 코드 생성 기법으로는 코드 확장기를 이용하는 방법과 코드 생성기를 이용하는 방법이 있다.

ACK에서는 중간 코드로서 EM을 사용하여 목적 코드로서 SPARC 코드를 생성하기 위한 주된 노력을 스택 지향(stack orientation) 구조로부터 레지스터 지향(register orientation) 구조로 변환하는 과정에 두고 있다. 이를 위해 코드 확장 시스템을 이용하여 EM 각각의 명령어를 입력으로 받아 해당 루틴에서 목적 코드로 확장한다[4][10].

SPARC (Scalable Processor ARChitecture)는 전형적인 RISC(Reduced Instruction Set Computer) 프로세서 구조로서 레지스터 지향적 구조를 갖는다. 모든 산술과 논리 연산은 레지스터에 위치한 피 연산자(Operand) 사이에서 수행된다. 적재와 저장 연산은 메모리로부터 레지스터의 내용을 적재하고 저장하는데 사용되어지고, 한 시점에 32개의 레지스터를 프로그래머가 사용할 수 있다[7].

이러한 배경을 바탕으로, 본 논문에서는 Bytecode를 목적 기계 코드로의 변환을 위해 ACK의 코드 확장 시스템을 이용하여 Bytecode를 입력으로 받아 SPARC 코드로 확장하는 코드 확장기(Code Expander)를 구현한다. ACK의 코드 확장 기법을 적용한 이유는 EM 코드와 Bytecode가 스택을 기반으로 한 중간 코드로서 의미적으로 유사하기 때문이다.

2. 재목적 코드 생성

재목적 코드 생성은 소스 프로그램에 대한 중간 코드를 다양한 목적기계를 위한 기계 코드로 번역하는 과정을 정형화된 방법을 통하여 자동으로 구성하는 것이다. 따라서 코드 생성 과정을 목적 기계 의존적인 부분과 목적 기계 독립적인 부분으로 나누어 정형화하는데 목적을 두고 있다. 이를 위해 다음 그림 2와 같은 재목적 코드 생성 시스템을 구현한다.

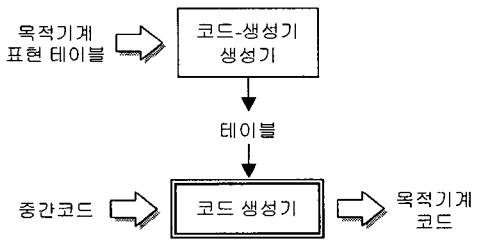

그림 2 재목적 코드 생성 시스템

재목적 코드 생성 시스템의 목적 기계 의존적인 부분에서는 목적 기계 표현 테이블을 입력으로 받아 목적 기계 정보 및 중간 코드에 대한 목적 코드 생성 정보를 저장하고 있는 테이블을 출력해준다. 또한 목적 기계 독립적인 부분에서는 중간 코드를 입력으로 받아 테이블 정보를 참조하여 하나의 공통된 코드 생성 알고리즘을

이용하여 실질적인 목적 기계 코드를 생성한다.

2.1 코드 생성 시스템

코드 생성기를 이용한 목적 코드 생성 방식은 중간 코드에 나타날 수 있는 다양한 패턴 정보를 참조해서 목적 코드를 생성하는 방법으로서 코드-생성기 생성기와 코드 생성기로 구성되어 진다. 코드-생성기 생성기는 목적기계 정보 및 코드 생성 규칙을 기술한 목적기계 표현 테이블을 입력으로 받아 목적 코드 생성시에 참조될 수 있는 정보를 생성한다. 코드 생성기는 중간 코드를 입력으로 받아 코드-생성기 생성기에 의해 생성된 정보를 참조하여 실질적인 목적 코드를 생성하게 된다[3][4].

2.2 코드 확장 시스템

코드 확장기를 이용한 목적 코드 생성 방식은 중간 코드를 목적기계 코드로 마크로 확장하는 방법으로 코드-확장기 생성기와 코드 확장기로 구성되어 진다. 코드 확장기 생성기는 중간 코드를 목적 기계 코드로 확장하기 위한 테이블을 참조하여 코드 확장 정보를 생성한다. 코드 확장기는 코드-확장기 생성기에 의해 생성된 코드 확장 정보를 이용하여 목적 기계 코드로 마크로 확장한다. 코드 확장 시스템에 의해 목적 코드를 생성하는 방식은 빠른 시간 내에 코드를 생성한다는 점과 코드 생성 시스템에 비해 구현이 쉽다는 장점이 있지만 코드 생성 시스템 방식에 비해 코드의 질이 떨어진다는 단점이 있다[3][4][11].

2.3 ACK

ACK는 컴파일러 후단부를 자동화하기 위한 도구로서 이식성과 재목적성이 매우 높은 컴파일러를 만들기 위한 실질적인 도구이며, 1960년에 처음 제안된 UNCOL(UNiversal Computer Oriented Language)의 개념에 기본을 두고 있다.

ACK에서 중간 코드 EM을 입력으로 받아 목적기계 코드인 SPARC 코드를 생성하기 위해 다음 그림 3과 같은 코드 확장 시스템을 이용하고 있다.

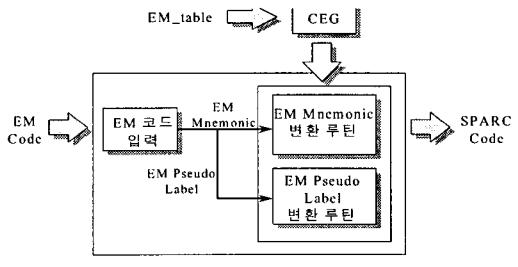

그림 3 ACK의 SPARC 코드 생성

ACK에서 중간 코드 EM을 SPARC 코드로 변환하기 위하여 코드의 질이 좋은 코드 생성 시스템 대신 코드 확장 시스템을 이용하는 이유는 다음과 같다. 첫째, 코드 생성기에 비해 구현이 용이하고, 코드 생성 시간이 빠르다. 둘째, 기계 종속적으로 설계된 컴파일러와 경쟁하기에는 전단부의 정보 손실이 많다. 마지막으로 SPARC과 같은 구조에서는 코드 확장기를 이용하여 생성된 목적 코드의 질이 반드시 떨어진다고 볼 수 없기 때문이다[4][10][11].

3. 자바 Bytecode를 위한 재목적 코드 생성기의 구현

3.1 시스템 개요

ACK에서 중간 코드를 입력으로 받아 목적 기계 코드인 SPARC 코드로 생성하기 위해 코드 확장 기법을 이용한다. 본 논문에서는 ACK의 중간 코드인 EM과 Bytecode가 스택 지향 구조를 갖는다는 점에서 유사하므로 ACK의 코드 확장 기법을 이용하여 그림 4와 같이 Bytecode를 입력으로 받아 목적 기계 코드인 SPARC 코드로 확장한다.

코드-확장기 생성기는 Bytecode를 SPARC 코드로 변환하는 테이블인 Bytecode 테이블을 입력으로 받아 코드 확장 정보를 생성한다. 정보 추출기는 Bytecode를 입력으로 받아 변형된 Bytecode 형식을 생성하며 코드 확장기는 코드 확장 정보를 참조하여 변형된 Bytecode를 목적 기계 코드인 SPARC 코드로 확장한다.

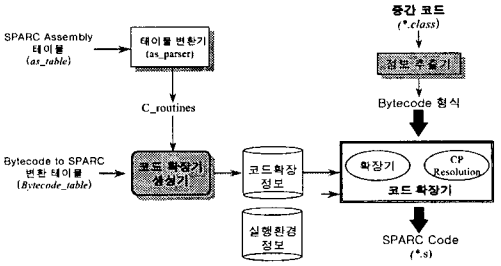

그림 4 코드 확장 시스템 개요

3.2 클래스 파일 및 Bytecode

클래스 파일은 8비트 스트림으로 구성되며, 16비트, 32비트, 64비트 크기는 2개, 4개, 8개의 연속적인 바이트를 읽어서 구성한다. 여러개의 바이트로 구성된 데이타 아이템은 상위 바이트가 먼저 오는 "Big-endian"의 순서로 저장된다.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 mdthods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info
    attributes[attributes_count];
}
```

그림 5 ClassFile 구조체

```
Code_attribute{
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {       u2 start_pc;
            u2 end_pc;
            u2 handler_pc;
            u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

확장될 Bytecode가 저장된곳

그림 6 코드 구조체

클래스 파일은 C 언어에서 사용하는 구조체를 이용하여 그림 5와 같은 하나의 ClassFile 구조체로 표현된다. 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. 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 Bytecode가 저장되어 있다. Code 속성을 갖는 구조체는 그림 6과 같다.

Bytecode는 피 연산자가 컴파일 시간에 결정되며 실행 시간에 계산되어 결과가 스택에 저장되는 스택 지향적 구조를 가진다. Bytecode의 데이타형에는 정수형에 속하는 byte, short, int, long, char 형이 있고, 실수형에 속하는 float, double 형이 있다.

3.3 SPARC 구조 및 코드

SPARC 구조는 32비트 RISC 구조를 갖는 기계로서 기본적인 연산을 수행하는 정수 처리부와 부동 소숫점 연산을 수행하는 실수 처리부로 CPU가 구성되어 있으며 병행적으로 동작한다. 또한 명령어와 데이타를 위한 가상 기억 장치 캐쉬를 제공하는 메모리 관리 장치와 32비트의 명령어 데이타 버스로 구성되어 있다.

SPARC는 프로시듀어의 호출과 반환 시에 발생하는 레지스터들의 저장과 복구에 관련된 부담을 줄이기 위해 레지스터 윈도우 개념을 사용한다. 레지스터 윈도우는 8개의 입력 레지스터, 8개의 지역 레지스터, 8개의 출력 레지스터와 같이 논리적으로 3개의 그룹으로 분류된다. 한 윈도우의 출력 레지스터는 다음 윈도우의 입력 레지스터로 사상된다. 또한, SPARC는 RISC 프로세서 구조이지만 적재 지연 슬롯이 발생하지 않도록 설계되어 있다.

SPARC의 연산자는 적재와 저장 연산, 산술/논리/이동 연산, 보조처리기 연산, 제어 이동 연산, 부동 소숫점 연산을 위한 명령어로 크게 분류할 수 있다. 또한, SPARC는 9개의 데이타 형이 있다. 정수로서 byte, unsigned byte, halfword, unsigned halfword, word, unsinged word 형을 가지고 있고, ANSI/IEEE 754-1985 실수형으로서 single, double, extended 형을 갖는다.

3.4 정보 추출기

정보 추출기는 클래스 파일에 대한 정보를 가지고 있는 테이블에서 그림 6과 같은 Code 속성 부분을 읽어들여 변형한다. 즉, Code 속성 부분에 있는 Bytecode를 SPARC 코드로 확장하기 위해 의사 명령을 추가하고 지역변수 표현법을 EM과 비슷한 형태로 만든다. 이는 Bytecode 테이블 구현시 Bytecode를 EM 코드와 유사한 형태로 변형함으로써 코드 생성에 필요한 테이블을 쉽게 구현할 수 있기 때문이다. 정보 추출기에 의해 변

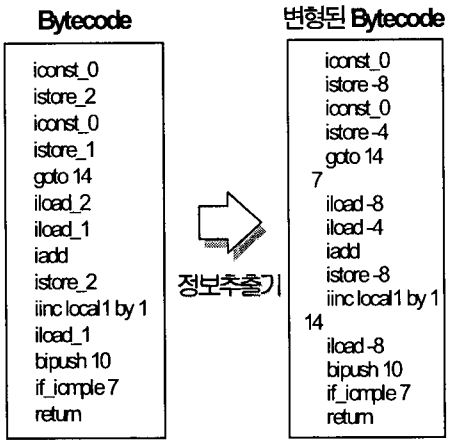

그림 7 정보추출기의 Bytecode 변형

형된 Bytecode의 예는 그림 7과 같다.

그림 7에서는 Bytecode 명령어인 istore_1을 정보 추출기에 의해 지역변수 표현법을 EM에서 표기하는 것과 같은 형태인 istore -4로 바꾸었고, goto와 같은 명령어를 위해서 레이블을 생성하였다. 레이블 생성은 Bytecode에서 goto 명령어와 같이 인수가 그림 6의 code 부분에 있는 Bytecode의 상대위치를 나타내면, 인수에 해당하는 Bytecode의 위치에 goto 명령어의 인수를 이용하여 레이블을 생성한다.

3.5 SPARC 레지스터 할당

Bytecode는 스택 지향 구조이고, SPARC 코드는 레지스터 지향 구조이다. 때문에 Bytecode를 SPARC 코드로 변환하기 위해서는 스택 지향 구조를 레지스터 지향 구조로 변환하는 레지스터 할당 과정이 필요하다.

메모리 참조와 레지스터 참조는 속도면에서 많은 차이가 있기 때문에 메모리에 대한 참조를 가능한 최소화하여 레지스터 참조 비율을 늘리는 것은 목적 코드의 실행 시간을 줄이는데 중요한 몫을 차지한다. 더욱이 많은 최적화 기법은 임시 변수를 만들어 내므로 이 임시 변수의 참조 비용은 사용된 최적화 기법의 성능에 큰 영향을 미친다. 이러한 이유 때문에 레지스터 할당은 컴파일러 구현에 있어 중요한 부분으로 인식되어 왔고 다양한 방법이 제안되어 왔다.

본 논문에서는 레지스터 할당 단계에서 ACK의 레지스터 할당 단계의 단점인 제약적 생존 범위의 미흡한 처리를 알고리즘 1과 같이 Chow의 Priority-Based Coloring 알고리즘[2]의 제약적 생존 범위 분할 방식을 적용하여 해결하였다.

(1) 제약적 및 비제약적 생존 범위 구분
(2) 모든 제약적 생존 범위를 대상으로 (a)-(c)를 반복 수행한다. 한번 수행될 때마다 하나의 생존 범위에 하나의 레지스터가 할당된다. 더 이상 할당할 수 있는 레지스터가 없거나 레지스터를 필요로 하는 생존 범위가 없을 때 마친다.
  (a) priority 값이 계산되지 않은 생존 범위의 priority 값을 계산하고, 계산된 priority 값이 음수이거나 레지스터를 할당받을 수 없는 생존 범위가 undo 조건을 만족하지 않으면 버린다. 만약 undo 조건을 만족한다면 이미 레지스터를 할당 받은 생존 범위 중 레지스터를 할당받을 수 없는 생존범위보다 이득 값이 작은 생존 범위들의 레지스터 할당을 취소한다.
  (b) 가장 큰 priority 값을 갖는 생존 범위 lr에 레지스터 할당.
  (c) lr과 간섭 관계를 갖는 생존 범위 중 할당 될 수 있는 레지스터가 없는 생존 범위를 찾아 생존 범위 알고리즘을 적용하여 분할한다.
(3) 비제약적 생존 범위에 레지스터를 할당한다.

알고리즘 1 레지스터 할당 알고리즘

위 알고리즘에서 제약적 생존 범위는 간섭 그래프 중 노드의 degree가 사용 가능한 레지스터의 수보다 큰 경우이고, 비 제약적 생존 범위는 노드의 degree가 사용 가능한 레지스터 수 보다 작은 경우이다. 또한, undo 조건은 레지스터를 할당받을 수 없는 생존 범위의 이득 값과 가장 최근에 레지스터를 할당받은 생존 범위의 이득 값이 같거나 큰 경우를 말한다.

재목적 코드 생성 시스템에서 실질적인 레지스터 할당은 생성될 SPARC 코드가 결정되고 레지스터 사용이 결정된 시점인 전역 최적화의 끝부분에서 수행된다. 목적코드 생성시에 레지스터 할당을 위한 정보를 저장하기 위해 다음과 같은 자료 구조를 이용한다.

```
struct reg_info{
    int     offset;
    int     size;
    int     pri;
    reg_t   reg,reg2;
} reg_Info;
```

또한, 이러한 정보를 이용하여 실질적으로 레지스터를 할당하기 위하여 사용중인 레지스터에 대한 정보를 표시하는 정보 플래그(inuse)를 둔다. inuse 플래그의 값이 0이 아닌 경우는 사용하려고 하는 레지스터가 이미 다른 정보를 저장하고 있어 사용할 수 없음을 표시하고, 1인 경우에는 해당 레지스터를 사용할 수 있음을 표시한다.

Bytecode로부터 SPARC 코드 생성시에 지역 변수에 대한 레지스터 할당 요구에 대해서 레지스터를 사용할 수 있는지를 결정하기 위해 find_local() 함수를 이용하여 레지스터 할당 정보를 검색하게 된다. 즉, 해당 Bytecode 명령어에 대한 SPARC 코드 생성시 이미 명령어 수행에 필요한 레지스터가 할당되어 있으면 레지스터를 사용할 수 있고 아직 레지스터가 할당되어 있지 않은 경우에는 사용 가능한 레지스터를 검색하여 새로운 레지스터를 요구한다. 이때, 사용 가능한 레지스터는 순차적으로 정렬되어 있어야한다.

3.6 Bytecode 테이블 기술

Bytecode 테이블은 코드-확장기 생성기에 의해 참조되어 중간 코드인 Bytecode를 목적기계 코드인 SPARC 코드로 확장하는데 필요한 정보로 구성된다.

```
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            ::= """[labe":"]
[INSTR]"""
INSTR   ::= mnemonic[operand( , operand)*]
```

그림 8 Bytecode 테이블 문법

Bytecode 테이블을 기술하는데 필요한 문법은 그림 8과 같다. Bytecode 테이블에는 주로 스택 지향 구조인 중간 코드를 레지스터 지향 구조인 목적기계 코드로 변환하기 위해 레지스터 할당을 위한 코드와 확장될 목적 코드가 기술된다. 그림 9는 Bytecode 테이블 문법을 이용하여 기술한 Bytecode 테이블의 일부분이다.

3.7 코드 확장기

코드 확장기는 코드-확장기 생성기에 의해 생성된 코드 확장 정보를 참조하여 변형된 Bytecode를 SPARC 코드로 확장한다.

정보 추출기에 의해 변형된 Bytecode를 코드 확장기가 Bytecode 테이블에 의해 생성된 코드 확장 정보를 이용하여 SPARC 코드로 번역한다. 예를들어 Bytecode인 iload2는 정보 추출기에 의해 iload -8로 변형되고, 변형된 Bytecode는 Bytecode 테이블을 참조하여 코드 확장기 생성기에 의해 생성된 코드 확장 정보를 이용하여 SPARC 코드인 ld [%fp+-8], %g1로 변환된다.

3.8 시스템 세부 구현

```
/* Pushing Constants onto the Stack      */
C_bipush  ==>
code_combiner(
    ;
    push_const($1))
.
/* Loading Local Variable onto the Stack */
C_iload  ==>
code_combiner(
;
        {
            reg_t S1;
            if (S1 = find_local($1, 0)) {
                soft_alloc_reg(S1);
                push_reg(S1);
            } else {
                soft_alloc_reg(reg_lb);
                push_reg(reg_lb);
                inc_tos($1);
                push_const(4);
                C_los(4);
            }
        })
.
/* Storing Stack Value onto Local Variable */
C_istore  ==>
code_combiner(
;
        {
                reg_t S1;
                if ((S1 = find_local($1, 0))) {
                        pop_reg_as(S1);
                } else {
                        soft_alloc_reg(reg_lb);
                        push_reg(reg_lb);
                        inc_tos($1);
                        push_const(4);
                        C_sts(4);
                }
        })
.
```

그림 9 Bytecode 테이블

본 논문에서는 ACK에 기반을 둔 코드 확장 시스템을 이용하여 Bytecode를 SPARC 코드로 변환하는 시스템을 그림 10과 같이 구현하였다. 이를 위해 Bytecode 테이블을 기술하였으며 Bytecode의 특성에 따라 as_table과 기계 종속적인 루틴을 구현하였다. 또한 Bytecode 테이블에 대한 어휘 분석기 및 구문 분석

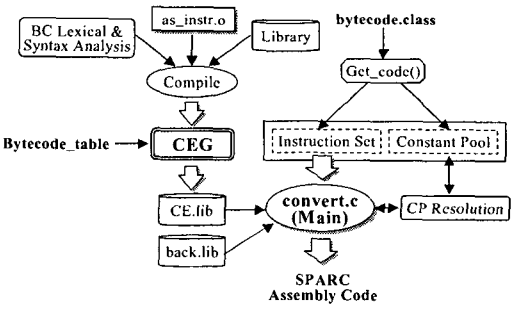

그림 10 시스템 세부 구현

기를 구현하고 as_table의 정보를 이용하여 코드-확장기 생성기를 생성하였다.

코드-확장기 생성기는 Bytecode에 대한 SPARC 코드로의 변환 규칙이 기술된 Bytecode 테이블을 입력으로 받아 각 Bytecode 명령어에 대한 코드 확장 루틴을 생성한다. 코드 확장 루틴과 함께 기계 종속적인 루틴은 코드 확장 시스템의 구성 요소가 된다. 정보 추출기는 자바 클래스 파일을 입력으로 받아 Bytecode 명령어의 코드 확장을 위해 필요한 정보인 코드 테이블과 동적 링킹을 지원하기 위해 설계된 심벌 테이블과 같은 상수 기억 장소를 테이블로 구성한다. 코드 확장기는 코드 확장기 생성기에 의해 생성된 코드 확장 정보를 이용하여, 입력되는 각 Bytecode 명령어의 엔트리 타입에 따라 상수 기억 장소의 엔트리를 결정(CP Resolution)하면서 SPARC 어셈블리 코드를 생성한다. 상수 기억 장소의 엔트리를 결정할 때 접근(Access)하고자 하는 아이템에 대한 권한을 검사한다.

SPARC 기계에 종속적인 루틴 작성시 JVM은 스택 구조에 기반을 둔 가상 기계이므로 스택을 이용하여 매개변수를 전달한다. 특히, JVM은 다양한 기계로의 이식을 용이하게 하기 위해서 프레임 안에 유지하고 있는 상태 변수들에 대해서 일반적인 레지스터를 사용하는 대신 스택과 지역 변수를 사용한다. 그러나 SPARC는 레지스터 기반 기계이므로 상태 정보를 유지하기 위해서 특정 레지스터를 이용한다. 따라서, 코드 확장기는 스택 포인터 레지스터(%sp)와 프레임 포인터 레지스터(%fp)의 상대적 주소 값을 이용해 적재와 저장 연산을 수행하는 SPARC 기계의 스택으로 대체한다.

4. 실험 결과

성능 측정을 위해 Bytecode를 자바 가상 기계에서의 실행 시간과 코드 확장기에 의해 생성된 목적 기계 코드의 실행 시간을 비교하였다. 실행 시간 비교는 JDK 1.2를 기준으로 하여 비교하였다.

성능을 측정하기 위한 테스트 프로그램으로서 Hanoi, Queens, Quick sort, Bubble sort 프로그램을 선택하였으며, 자바로 기술된 프로그램을 JDK 1.2의 Javac로 컴파일하여 중간 파일인 클래스 파일을 생성하였다. 생성된 중간 파일을 본 논문에서 제시한 재목적 코드 생성기에 의해 생성된 코드의 실행 시간과 비교하기 위해 JDK 1.2의 java로 실행시켰을 때의 실행 시간을 측정하였다. 비교 평가 환경은 목적 기계로서 Sun의 SPARC Station 5를 사용하였고 OS는 Solaris 2.6을 이용하였다.

실행 시간 측정 시 JDK 1.2의 경우는 클래스 파일을 입력으로 받아 실행되는 시간을 측정하였고, 재목적 코드 생성기의 경우에는 클래스 파일을 입력으로 받아 SPARC 코드를 생성하는 변환 시간을 제외하고 SPARC 코드를 생성한 후 SPARC 코드가 실행되는 시간을 측정하였다.

JDK 1.2가 JIT 방식을 사용하고 있다는 사항을 감안하여 실험 대상 프로그램의 반복적인 수행시간이 적은 경우와 많은 경우로 나누어 실행 시간을 측정하였다. 즉, 입력되는 데이타 집합의 크기를 조정하여 비교하였다.

그림 11의 실행 결과는 입력되는 데이타의 집합이 작아서 프로그램의 수행이 많이 반복되지 않는 경우에 해당된다. 그림 12의 경우는 입력되는 데이타의 집합을 크게 하여 프로그램의 반복 수행 시간이 긴 경우이다. 이와 같이 프로그램의 반복 수행되는 시간이 길어질 수록 JDK 1.2에서 수행한 결과나 재목적 코드 생성기에 의해 목적코드로 변환되어 수행되어지는 시간은 거의 비슷해진다. 이와 같은 현상이 발생하는 이유는 JDK 1.2에서 프로그램 실행 중 JIT에 의해 Bytecode가 목적코드로 바뀌어 실행되기 때문에 프로그램의 실행 시간이 거의 비슷하게 나온다.

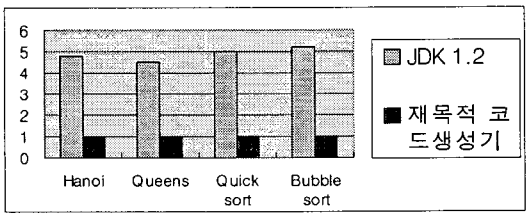

그림 11 실행시간 비교 1

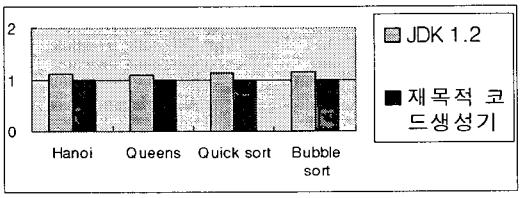

그림 12 실행시간 비교 2

5. 결론 및 향후 연구 과제

컴파일러 자동화 도구인 ACK에서는 코드 확장 시스템을 이용하여 SPARC 코드를 생성한다. 코드 확장 시스템을 이용한 SPARC 코드 생성 기법은 코드 생성 시

스템에 비해 코드의 질은 다소 떨어지는 반면, 빠른 속도로 코드를 생성한다는 장점과 구현하기 용이 하다는 장점을 가지고 있다.

본 논문에서는 Bytecode를 SPARC 코드로 번역하기 위해 재목적 코드 생성 가능한 ACK의 코드 확장 시스템에 기반하여 코드 확장 시스템을 설계하고 구현하였다. Bytecode로 부터 SPARC 코드 확장에 필요한 정보를 Bytecode 테이블에 기술하였으며 Bytecode 테이블을 참조하여 코드-확장기 생성기에 의해 코드 확장 정보를 생성했다. 코드 확장기는 중간 코드인 Bytecode를 입력으로 받아 코드 확장 정보를 참조하여 실질적으로 SPARC 코드를 생성하는 부분이다.

중간 코드로부터 목적 코드를 생성하기 위해 재목적 가능한 코드 생성 기법을 이용함으로써 Bytecode를 각 기계에 적합한 목적 코드 생성시에 용이할 것이다. 또한, JIT에서 목적 코드 생성시에 본 논문에서 제시한 코드 생성 기법을 이용하여 목적 코드를 생성할 수 있다.

생성된 코드의 실행을 위해 자바의 실행 환경을 각 기계의 실행 환경으로 변환하는 연구가 진행중이며, 보다 양질의 코드를 생성하기 위해 코드 최적화에 대한 연구와 중간 코드의 패턴을 이용하는 코드 생성 기법에 대한 연구도 진행 중에 있다. 또한, 본 논문에서 목적 기계 코드로서 SPARC 코드를 생성한 재목적 가능한 코드 생성 시스템을 이용하여 Pentium 코드를 생성하는 연구도 진행중이다.

참 고 문 헌

[1] Andrew S. Tanenbaum, Hans van Staveren, Ed G. Keiser and Johan W. Stevenson, "Description of a machine architecture for use with block structured language," Dept. of mathmetics and Computer Science, Vrije Universiteit, 1983.

[2] Chow F. and Hennessy J. "Register allocation by priority-based coloring," Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction, ACM, pp.222-232, 1984.

[3] Franckassshoek, Koen Langendoen, "The Code Expander Generator," Dept. of Mathmetics and Computer Science, Vrije Universiteit, 1990.

[4] Hans van Staverren, "The table driven code generator from ACK 2nd. Revision," report-81, netherlands Vrije Universiteit, 1989.

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

[6] Peter van der Linden, "just JAVA," Prentice-Hall, 1996.

[7] Richard P. Paul, "SPARC ARCHITECTURE, ASSEMBLY LANGUAGE PROGRAMMING, AND C," Prentice-Hall, 1994.

[8] The SPARC Architecture Manual, SUN Micro., 1987.

[9] Tim Lindholm, Frank Yellin, "The Java Virtual Machine Specification" Addison-Wesley, 1996.

[10] 윤영식, 고광만, 오세만, "속성 EM 트리를 이용한 SPARC 코드 확장기의 설계 및 구현", 한국정보과학회 '96년 봄 학술 발표 논문집, 제23권 1 호, pp.377- 380, 1996.

[11] 정찬성, 황순명, 오세만, "Bytecode에서 SPARC 코드로의 코드 확장", 한국정보과학회 '97년 가을 학술 발표 논문집, 제24권 2호, pp.367-370, 1997.

오 세 만

1977년 서울대학교 사범대학 수학과 졸업. 1979년 한국과학기술원 전산학과 석사학위 취득. 1985년 한국과학기술원 전산학과 박사학위 취득. 1985년 ~ 1988년 동국대학교 전산학과 조교수. 1988년 ~ 1989년 미국 USL대학 교환교수. 1994년 ~ 현재 동국대학교 컴퓨터공학과 교수. 관심분야는 프로그래밍 언어론, 컴파일러 구성론임.

정 찬 성

1996년 2월 원광대학교 컴퓨터공학과 학사 졸업. 1998년 2월 동국대학교 컴퓨터공학과 석사 졸업. 2000년 2월 동국대학교 컴퓨터공학과 박사 수료. 현재 드림인텍(주) 기술연구소 연구원. 관심분야는 프로그래밍언어, 컴파일러