스트링 패턴 매칭 기법을 이용한 중간 코드 변환기의 설계 및 구현

Design and Implementation of Intermediate Code Translator using String Pattern Matching Technique

고 광 만*
Kwang-Man Ko

요 약

자바 언어의 실행 속도를 개선하기 위해 전통적인 컴파일 방법을 사용하여 바이트코드를 특정 프로세서에서 수행될 수 있는 목적기계 코드로 변환하는 다양한 연구가 진행중이다. 패턴 매칭을 이용한 코드 생성 기법은 코드 확장 기법에 비해 양질의 코드를 생성할 수 있는 장점을 가지고 있다. 본 연구에서는 바이트코드로부터 효과적으로 네이티브 코드를 생성하기 위해 레지스터 기반의 중간 언어를 효율적으로 생성할 수 있는 정형화된 패턴 기술 방법과 패턴 매칭 기법에 대해 제시한다. 또한 기술된 정형화 패턴을 활용하여 양질의 레지스터 기반 중간 코드를 생성하는 중간 코드 변환기를 설계하고 구현하였다.

Abstract

The various researches are investigated for transforming byte code into objective machine code which can be implemented in the specific processor using classical compiling methods to improve the execution speed of the JAVA language. The code generation techniques using pattern matching can generate more high-quality code than code expansion techniques. We provide, in this research, the standardized pattern describing methods and pattern matching techniques that can be used to generate the register-based inter-language which is for the effective native code generation from byte code. And we designed and realized the inter-code transformer with which we can generate the high-quality register-based inter-code using standardized pattern described formerly.

1. 서 론

자바 언어 시스템에서는 자바 언어를 플랫폼에 독립적으로 실행시키기 위해 가상 기계 코드인 바이트코드를 사용하며 인터프리터를 이용하여 실행하고 있다. 이로 인해 작은 크기의 자바 응용 프로그램 수행에는 실행 속도 문제가 중요한 요소가 아니지만 대형 프로그램의 수행에는 실행 속도가 현저히 저하되는 단점을 가지고 있다. 이러한 실행 속도 문제를 개선하기 위해 바이트코드를 위한 프로세서를 제작하여 직접 하드웨어로 실행시키는 자바 칩을 이용하는 방법, JIT 방식으로 실행 시간에 필요에 따라 컴파일하여 실행하는 방법, 전통적인 컴파일 방식을 사용하여 바이트코드를 수행하고자 하는 특정 프로세서에 적합한 네이티브 코드를 생성하는 실행하는 방법 등이 사용되고 있다[7].

바이트코드는 플랫폼 독립적인 특성으로서 컴파일러 자동화 도구인 ACK에서 사용된 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]. 또한 바이트코드로부터 특정 프로세서에 적합한 네이티브 코드를 생성하기 위한 다양한 연구가 진행되어 왔다. CACAO[8]는 Alpha 프로세서를 위한 64비트 JIT 컴파일러로서 바이트코드를 입력으로 받아 Alpha 프로세서를 위한 네이티브 코드를 생성한다. 네이티브 코드

* 상지대학교 컴퓨터정보공학부 교수
kkman@mail.sangji.ac.kr

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 네이티브 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 네이티브 코드를 생성하기 위해 자바의 중간 언어인 *.class 파일을 입력으로 받아 레지스터 기반의 중간 언어로 변환한다. 이를 위해 첫째, 바이트코드로부터 레지스터 기반 언어로의 효율적인 변환을 위한 패턴을 정형화된 방법으로 기술한다. 둘째, 정형화된 패턴을 입력으로 받아 패턴 매칭에 적합한 테이블 정보를 생성한다. 셋째, 테이블 정보를 참조하여 실질적인 패턴 매칭을 통해 양질의 레지스터 기반 언어를 생성하는 패턴 매칭기를 구현한다.

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 언어에 대해 소개한다. 제 3장에서는 본 연구에서 구현하고자 하는 전체 시스템 모델 및 중간 언어 번역기 모델과 각 요소의 특성에 대해 기술한다. 구체적으로 3.2절에서 실제적으로 바이트코드를 입력으로 받아 코드 확장 방식으로 레지스터 기반 중간 언어를 생성하는 과정과 생성된 결과를 기술한다. 3.3절에서는 중간 언어의 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 중간 언어

자바 컴파일러에 의해 생성되는 클래스 파일[10]은 플랫폼 독립성을 위해 설계된 8비트 크기의 바이트 스트림으로 구성되어 있으며 2개, 4개, 8개의 연속적인 바이트 스트림으로 구성되어 있으며 ClassFile 구조체형식을 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.

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 결정되

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

본 연구에서는 스택 기반 바이트코드 입력을 레지스터 기반 중간 언어로 변환하기 위해 오프라인 자바 컴파일러인 Jcc에서 사용하는 gasm 중간 언어[9]로 변환한다. gasm은 move, add, compare 등과 같은 명령어를 가지고 있으며 개략적인 형식은 그림 1과 같다.

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.

```
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 형식

2.2 코드 확장 방식과 패턴 매칭 코드 생성

코드 확장 방식에서는 가상 기계에 대한 코드를 생성하고 실제 목적 기계에 대한 코드의 생성은 마크로 확장(macro expansion)이나 스키마(schema)를 이용한다. 이 기법에서는 코드 생성 방법과 목적 기계에 대한 특징이 혼합되어 있어 다른 목적 기계에 대한 코드 생성시에 코드 생성 부분의 수정과 동시에 목적 기계의 특징을 수정해야 하는 단점을 가지고 있다. 또한 각각의 중간 언어에 대해 대응되는 목적 코드간의 관계를 1:1 방식으로 변환한다[3].

패턴 매칭 방법을 이용한 코드 생성 방법은 양질의 코드 생성을 위해 중간 언어에 대한 분석을 수행한 후 하나 이상의 명령어로 구성된 패턴에 대해 목적 코드를 생성하는 방식이다. 따라서 패턴 매칭을 이용한 코드 생성 방식에서는 먼저 중간 언어의 패턴에 대한 변환될 목적 코드간의 관계가 기술되어 실질적인 코드 변환 시에 참조될 수 있도록 해야한다. 패턴 매칭을 이용한 코드 생성은 Aho에 의해서 제안된 "brute-force"방법이 있으며 최적의 코드를 생성하기 위해 동적 프로그래밍 방법을 이용한다. 또 다른 형태로 Ripken 방법에서는 원시 프로그램상에 존재하는 제어의 흐름을 분석하여 속성화된 중간 표현을 이용하여 코드를 생성하는 방법을 제시하였다. 테이블을 이용한 코드 생성 방법은 Gramham과 Glanville에 의해 제안되었으며 테이블을 이용한 코드 생성의 또 다른 방법으로 Cattell은 중간 표현으로써 트리 구조에 기반을 둔 TCOL을 사용하였으며 목적 기계의 코드를 생성하기 위해 순환적으로 트리를 순회할 수 있는 알고리즘을 사용하였다. 트리-재구성(Tree-Rewriting)을 이용한 코드 생성 방법은 효율적인 코드 생성을 위해 TWIG라 부르는 트리 조작 언어를 이용해서 목적 기계의 코드를 생성하는 방법이다. TWIG를 위한 컴파일러는 트리-매칭 방법을 이용하고 있으며 중복된 패턴 매칭이 발생하는 경우, 최적화된 코드 생성을 위해 동적

프로그래밍 기법을 이용한다. 목적 기계의 명령어를 선택하기 위해 트리-재작성 규칙을 이용한다 [1,2,4].

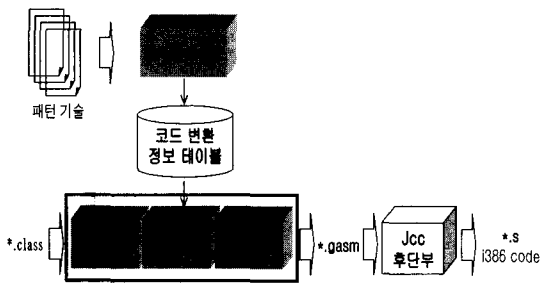
2.3 패턴 매칭 테이블

컴파일러 자동화 도구인 ACK에서는 중간 코드를 입력으로 받아 양질의 목적기계 코드를 생성하기 위해 목적기계 표현 테이블을 기술하였다. 목적기계 표현 테이블에는 다양한 목적기계 정보 및 중간 코드를 입력으로 받아 생성하고자 하는 목적기계 코드에 대한 생성 규칙이 정형화된 방법으로 기술되어 있다. 특히, 중간 코드에 대한 패턴 기술 부분은 1:1, 1:N, N:1, N:M 방식으로 다양하게 기술되어 있으며 정확한 패턴 매칭이 발생되지 않을 경우를 대비하여 수정된 형태의 코드 생성이 가능하도록 하였다. 먼저 중간 언어에 대한 목적기계 코드 패턴을 기술하기 위해서는 중간 언어가 가질 수 있는 명령어 명칭과 피연산자가 특징에 따라 기술된다. 또한 현재 명령어의 주소 표현 방식과 일치하지 않을 경우 이전시키는 방법을 정의하는 절에서는 move, from, to, gen 지정어를 이용하여 융통성을 높이고 있다. 또한 코드 규칙의 패턴과 정확히 일치하지 않을 때 다른 명령어로 변경시키는 방법이 기술되어 있다. 마지막으로 이미 기술된 정보를 참조하여 실질적인 코드 변환을 위해 중간 코드 패턴에 대한 목적 코드로의 변환 규칙 정의한다. 실질적인 목적 코드 생성 시에 의해 패턴 매칭을 수행하는 경우에는 가장 긴 패턴에 대해 우선적으로 패턴 매칭을 수행하게 되며 동일한 길이를 가진 패턴에 대해서는 첫 번째 발견되는 패턴을 실행하게 된다[3].

3. 패턴 매칭 중간 언어 변환기

3.1 시스템 모델

패턴 매칭 중간 언어 변환기는 자바 바이트코드를 입력으로 받아 i386 네이티브 코드를 생성하기 위해 오프라인 자바 컴파일러인 Jcc의 후단부 입력에 적합한 gasm 코드를 생성한다. 이를 위해 스택 기반 중간 코드인 바이트코드를 입력으로 받아 컴파일러 자동화 도구인 ACK의 코드 확장기법에서 적용되었던 스트링 패턴 매칭 기법을 사용하여 레지스터 기반 중간 언어인 gasm 코드를 생성한다. 바이트코드로부터 생성된 gasm 코드는 Jcc 후단부에서 제공하는 기계 정보(Machine Description; MD) 정보와 라이브러리를 참조하여 i386 네이티브 코드를 생성한다. 본 논문에서 구현하고자 하는 시스템 모델은 그림 2와 같다.



(그림 2) 전체 시스템 구성도

패턴 기술(pattern description) 부분에서는 바이트코드를 gasm 코드로 변환하는 패턴이 기술되며 테이블 생성기에 의해 코드 변환 시에 필요한 정보로 생성된다. 약 230개로 구성된 바이트코드의 명령어를 적재/저장 명령어, 지역 변수에 값을 저장하는 명령어, 배열을 관리하는 명령어, 산술/논리 연산을 위한 명령어, 자료형 변환 명령어, 분기 및 제어 관리 명령어, 객체 관리 및 메소드 호출 명령어 등으로 구분하여 유사한 특성을 갖는 바이트코드 명령어 패턴을 기술하였다. 테이블 생성기는 코드 생성에 필요한 정보를 추출하는 부분으로서 패턴 기술에 대한 어휘 분석 및 구문 분석 동작을 수행한 후에 패턴 매칭에 적합하도록 유사한 특성을 갖는 그룹 및 기술되는 바이트코드 명령어의 개수에 따라 순차적으로 저장되어 있다. 실질적인 코드 생성은 바이트코드 추출기, 패턴 매칭기, 코드 출력기로 구성된 중간 언어 변

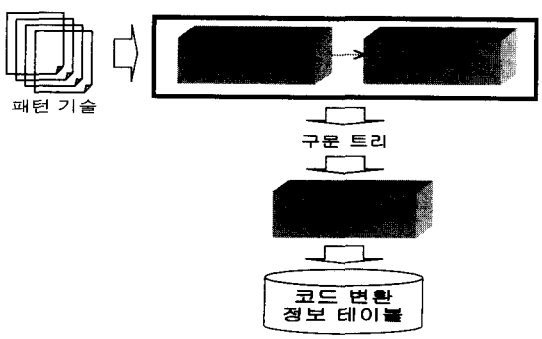
환기에 의해 이루어진다. 바이트코드 추출기는 클래스 파일을 입력으로 받아 클래스 파일의 상수 풀 정보를 분석하여 바이트코드를 추출한다. 패턴 매칭기는 입력되는 바이트코드 스트림을 분석하여 해당되는 최적의 패턴을 코드 변환 정보 테이블을 참조하여 스트링 패턴 매칭을 수행한다. 코드 출력기는 추출된 패턴 정보를 이용하여 실질적으로 gasm 코드를 생성하는 부분이다. 생성된 gasm 코드는 Jcc의 후단부로 입력되어 최종적으로 i386 네이티브 코드를 생성한다.

3.2 패턴 기술 및 테이블 생성기

패턴 기술 부분은 바이트코드를 레지스터 기반 중간 언어인 gasm 코드로 변환하는데 필요한 핵심적인 부분으로 바이트코드에 대한 gasm 코드의 매핑 정보를 기술한다. 기술된 패턴으로부터 테이블 생성기를 이용하여 코드 변환 정보 테이블을 생성하는 과정은 그림 3과 같다.

양질의 gasm 코드를 생성하기 위해 기본적으로 230여 개의 바이트코드에 대한 코드 변환 정보가 1:1 방식으로 우선적으로 기술한 다음 바이트코드 패턴에 대한 gasm 매핑 패턴을 순차적으로 기술한다. 패턴 기술은 크게 바이트코드 절(Bytecode_Section), gasm 코드 절(Gasm_Section), 패턴 절(Pattern_Section)로 다음과 같이 구성되어 있다.

Root ::= Bytecode_Section Gasm_Section Pattern_Section ;



(그림 3) 코드 변환 정보 테이블 생성

바이트코드 절에서는 바이트코드에 대한 모든 정보를 기술하는 부분으로서 각각의 바이트코드가 갖는 피연산자의 종류까지 자세하게 속성이 기술되는 부분이다. 바이트코드 절은 바이트코드의 특성에 따라 크게 적재/저장 명령어, 산술/논리 연산을 위한 명령어, 자료형 변환 명령어, 상수 관리 명령어, 스택 제어 명령어, 분기 및 제어 관리 명령어, 배열을 관리하는 명령어, 객체 및 배열 생성 명령어, 객체 조작 명령어, 메소드 호출 명령어 등으로 구분하여 기술하며 패턴 절에서 패턴을 기술시에 활용된다. 실제적으로 230여 개 바이트코드에 대한 정보를 저장하기 위해 그림 4와 같은 구조체를 정의하여 정보를 저장한다.

```
typedef struct OpcodeList {
                    BYTE_CODE_OPCODE code ;
                    int operands ;
                    int args ;
                    const char* name ;
} OpcodeList ;
```

(그림 4) 바이트코드 정보 저장을 위한 구조체

gasm 절에서는 gasm 명령어 형식에 대한 다양한 특성 정보를 기술한다. gasm에서 허용되는 레지스터의 종류 및 크기는 그림 5와 같은 구조체에 정의되어 있다.

```
typedef struct GasmRegisters {
  int bX; // general byte registers(8 bit)
  int sX; // general short registers(16 bit)
  int iX; // general integer registers(32 bit)
  int lX; // general long long registers(64 bit)
  int gX; // general machine-reg-wide registers(32/64 bit)
  int fX; // general float registers
  int dX; // general double registers
} GasmRegisters;
```

(그림 5) gasm 레지스터 종류 및 크기

패턴 절에서는 바이트코드에 대응하는 gasm 코드 패턴이 정형화된 형식으로 기술된다. 정수형 덧셈 연산을 위해 사용되는 iload, iadd, istore 각 명령어에 대한 변환 패턴은 그림 6과 같다.

```
iload => *(%Mem_Addr_X) = Value
iadd => %Mem_addr_X
        = *(%Mem_Addr_X) + *(%Mem_Addr_X)
istore => move %ix, *(%Mem_addr_X)
```

(그림 6) 바이트코드에 대한 gasm 매핑 패턴

실제적으로 Java 언어의 입력에 대해 변환된 바이트코드는 본 연구에서 기술된 패턴 정보를 참조하여 그림 7과 같이 변환된다.

```
'int i = 10;      => * %(fp+0) = 10
int a = I++; => * %(fp+4) = (%g0 = * %(fp+0),
                        * %(fp+0) = * %(fp+0) + 1, %g0)
int b = ++i;          => * %(fp+8) = (* %(fp+0),
                        * %(fp+0) = * %(fp+0) + 1)
```

(그림 7) 패턴 정보를 이용한 gasm 코드 변환

테이블 생성기는 코드 생성에 필요한 정보를 추출하는 부분으로서 패턴 기술에 대한 어휘 분석 및 구문 분석 동작을 수행한다. 구문 분석을 위한 문법 기술은 파서 생성기인 bison 입력에 적합하도록 작성하였으며 구문 분석 후에 패턴 기술에 대한 정보를 구문 추상화 구문 트리(abstract syntax tree)에 저장하였다. 추상화 구문 트리를 순회하면서 입력되는 바이트코드 스트림과 패턴 매칭에 적합하도록 유사한 특성을 갖는 그룹 및 기술되는 바이트코드 명령어의 개수에 따라 순차적으로 배열로 구성된 테이블에 저장하였다. 실질적으로 테이블 정보를 생성하는 주 함수는 하나의 패턴이 인식되면(reduce) 패턴 정보를 pattern[ ]에 저장한다. 이러한 반복 과정을 통해서 전체적인 패턴을 각각 저장하게 되면 해쉬 테이블을 이용하여 효과적으로 전체적인 패턴 저장을 관리한다.

3.3 중간 코드 변환기

중간 코드 변환기는 크게 바이트코드 추출기, 패턴 매칭기, 코드 출력기로 구성되어 있다. 바이트코드 추출기는 클래스 파일을 입력으로 받아 클래스 파일의 상수 풀 정보를 분석하여 바이트

(그림 8) 바이트코드 추출기

코드를 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 8과 같다.

따라서 바이트코드 추출기는 실질적인 중간 코드 변환을 위한 전처리 과정으로 간주된다. 바이트코드 추출기를 통해 출력된 각각의 바이트코드는 다음 단계인 패턴 매칭기에 의해 실질적인 중간 코드 변환이 이루어진다.

패턴 매칭기는 입력되는 바이트코드 스트림을 분석하여 해당되는 최적의 패턴을 코드 변환 정보 테이블을 참조하여 스트링 패턴 매칭을 수행한다. 스트링 패턴 매칭을 위해서는 먼저 입력 스트림을 읽어들여 바이트코드를 추출한다. 추출된 바이트코드를 순서적으로 읽어들여 가장 먼저 발견되는 바이트코드를 코드 변환 정보 테이블에서 검색한다. 코드 변환 정보 테이블은 인덱스 테이블과 실질적인 패턴을 저장하고 있는 정보 테이블로 구성되어 있다. 먼저 인덱스 테이블은 그림 9와 같은 과정을 거쳐서 구성된다.

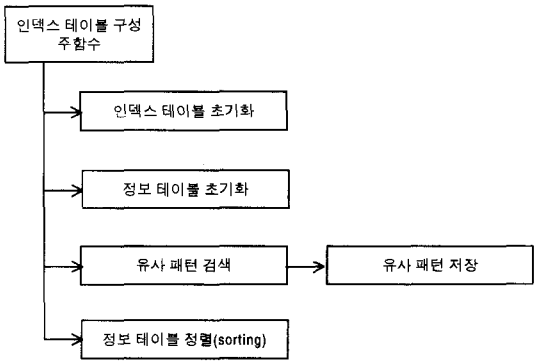
패턴 매칭기에 의해 수행되는 스트링 패턴 매칭 기법은 입력 스트림을 순서적으로 읽어 각각의 바이트코드를 인덱스 테이블 및 정보 테이블을 참조하여 비교한다. 실질적으로 스트링 패턴 매칭을 수행하는 핵심 알고리즘은 그림 10과같다.

3.4 실험 결과 및 분석

본 논문에서 바이트코드를 입력으로 받아 Jcc 후단부 입력인 gasm 코드를 스트링 패턴 매칭 기법

을 적용하여 생성하였다. Jcc 후단부는 gasm 코드를 입력으로 받아 i386에서 수행할 수 있는 네이티브 코드를 생성하며 실질적으로 프로그램 실행 결과를 확인할 수 있다. 본 논문에서 구현한 중간 코드 변환기의 타당성을 검증하기 위해 그림 11과 같은 Java 언어를 입력으로 받아 바이트코드 추출기를 통해 바이트코드를 생성하였다.

본 연구에서 구현한 중간 코드 변환기는 그림 11의 바이트코드를 입력으로 받아 그림 12와 같은 gasm 코드를 생성하였으며 생성된 gasm 코드를 다시 Jcc 후단부에 입력하여 i386 네이티브 코드를 생성하였다. 생성된 네이티브 코드의 올바른실행 결과를 통해 본 연구에서 구현한 중간 코드변환기의 타당성을 입증하였다.



(그림 9) 코드 변환 정보 테이블 구성

```
while( fgets(line, sizeof(line), in) != NULL) {
  token = strtok(line,fieldsep);
     if( strcmp(token, BytePattern[count])==0) {
       count++;
          while((token=strtok(NULL, fieldsep)) != NULL)
{
               if(strcmp(token, BytePattern[count])==0 ) {
                     count++;
                     token=NULL;
                  }
               else break ;
            }
       }
        . . .
}
```

(그림 10) 스트링 패턴 매칭 알고리즘

4. 결론 및 향후 과제

Java 응용 프로그램의 실행 속도 문제를 개선하기 위해 바이트코드를 위한 프로세서를 제작하여 직접 하드웨어로 실행시키는 자바 칩을 이용하는 방법, JIT 방식으로 실행 시간에 필요에 따라 컴파일하여 실행하는 방법, 전통적인 컴파일 방식을 사용하여 바이트코드를 수행하고자 하는 특정 프로

Java 소스 프로그램	바이트코드

int foo() {
  int j=0;
  for(int i=0; i<10; I++) {
  j++;
  }
  return j;
}
	int foo() {
	  //0 0:iconst_0
	  //1 1:istore_1
	  //2 2:iconst_0
	  //3 3:istore_2
	  //4 4:goto 13
	  //5 7:iinc 1 1
	  //6 10:iinc 2 1
	  //7 13:iload_2
	  //8 14:bipush 10
	  //9 16:icmplt 7
	  //10 19:iload_1
	  //11 20:ireturn
	  }

(그림 11) Java 소스 프로그램 및 바이트코드

```
% g21799 = param pointer this ;
. . .
local int _36 = 0 ;   // int j
local int _37 = 0 ;   // int i
        if(local int _37 < 10 ) { do
           (% g21800 = local int & _36, (% i21801 =
                              *(0 + % g21800, int),
            *(0 + % g21800, int) = *(0 + % g21800,
                          int)signed + 1, % i21801) ) ;
           (% g21802 = local int & _37, (% i21803 =
                              *(0 + % g21802, int ),
            *(0 + % g21802, int) = *(0 + % g21802,
                          int)signed + 1, % i21803 ) ) ;
        } while ( ( ) local int _37 < 10 ) ;


. . .
% i21804 = local int _36 ;
return % i21804 ;
}
. . .
```

(그림 12) gasm 코드 생성

세서에 적합한 네이티브 코드를 생성하는 실행하는 방법 등이 사용되고 있다.

바이트코드로부터 특정 프로세서에 적합한 네이티브 코드를 생성하기 위한 다양한 연구가 진행되어 왔다. CACAO[8]는 Alpha 프로세서를 위한 64비트 JIT 컴파일러로서 바이트코드를 입력으로 받아 Alpha 프로세서를 위한 네이티브 코드를 생성한다. NET 컴파일러[7]은 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 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 Java 응용 프로그램의 실행 속도 개선을 위해 바이트코드로부터 직접 네이티브 코드를 생성하기 위해 자바의 중간 언어인 *.class 파일을 입력으로 받아 레지스터 기반의 중간 언어로 변환하였다. 이를 위해 첫째, 바이트코드로부터 레지스터 기반 언어로의 효율적인 변환을 위한 패턴을 정형화된 방법으로 기술하였다. 둘째, 정형화된 패턴을 입력으로 받아 패턴 매칭에 적합한 테이블 정보를 생성하였다. 셋째, 코드 변환 정보 테이블을 참조하여 스트링 패턴 매칭 기법을 적용하여 레지스터 기반 중간 언어인 gasm 코드를 생성하는 패턴 매칭기를 구현하였다. 본 연구를 통해서 생성된 gasm 코드는 Jcc의 후단부에 입력되어 실질적으로 i386 네이티브 코드를 생성하였다. 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.

Acknowledgement

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

참 고 문 헌

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

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

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

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

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

[7] 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 Micro-architecture, Dec., 1996.

[8] 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.

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

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

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

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

◐ 저 자 소 개 ◑

고 광 만

1991년 원광대학교 컴퓨터공학과 졸업(학사)
1993년 동국대학교 대학원 컴퓨터공학과 졸업(석사)
1998년 동국대학교 대학원 컴퓨터공학과 졸업(박사)
1998년 3월~2001년 8월 광주여자대학교 정보통신학부 교수
2001년 9월~현재 : 상지대학교 컴퓨터정보공학부 교수
관심분야 : 프로그래밍언어론, 컴파일러구성론, 모바일 컴퓨팅 etc.
E-mail : kkman@mail.sangji.ac.kr