바이트코드 최적화 프레임워크의 설계

김영국, 김경수, 김기태, 조선문, 유원희
인하대학교 전자계산공학과
e-mail: odin1004@hotmail.com

Design of Bytecode Optimization Framework

Young-Kook Kim, Kyung-Soo Kim, Ki-Tae Kim,
Sun-Moon Jo, Weon-Hee Yoo
Dept. of Computer Science and Engineering, Inha University

요 약

자바는 객체지향 언어이고 바이트코드로 번역 이후에는 플랫폼에 독립적으로 가상머신에 의해 실행될 수 있기 때문에 소프트웨어 개발과 유지보수에 많은 장점을 갖는다. 이러한 특징으로 인해 플랫폼에 독립적인 소프트웨어 개발에는 자바가 많이 이용된다. 그러나 바이트코드로 작성된 프로그램은 가상기계에서 인터프리터 방식으로 수행된다. 때문에 프로그램의 실행속도가 느리게 실행되는 문제점을 가지고 있다. 실행속도의 문제점을 해결하기 위한 여러 가지 방법들이 연구가 진행중이다.

본 논문은 자바 바이트코드가 가상기계에서 인터프리터 방식으로 수행할 때 바이트코드의 크기를 줄여 해석하는 부담을 줄이기 위해서 바이트 코드를 최적화하는 프레임워크를 구성한다. 프레임워크를 이용하여 바이트코드를 3주소 형태의 CTOC-T(Class To Optimizer Classes-Three Address Code)로 변환하여 프로그램을 분석을 할 수 있다. 또한 CTOC-T는 3주소 형태이므로 3주소 최적화 기법을 적용하여 최적화된 바이트코드를 생성하는 프레임워크를 설계한다.

1. 서론

자바는 객체지향 언어이기 때문에 최신에 객체지향 기술을 적용할 수 있다. 또한 자바 소스 프로그램을 컴파일하여 바이트코드를 생성한다. 바이트코드는 프로그램의 수정과 재 컴파일 과정이 없이도 가상머신에서 실행된다.

그러나 자바는 가상머신을 이용하여 플랫폼에 독립적이다. 가상머신은 바이트 코드를 인터프리터 방식으로 수행하기 때문에 실행속도가 느리다는 단점을 가진다. 실행속도의 단점을 극복하기 위해서 다양한 방법이 연구되고 있다. 첫 번째는 바이트코드 실행을 위한 하드웨어 개발로 자바 칩 개발이 있다. 두 번째는 JIT방식으로 실행시간에 메소드 단위로 컴파일하는 방법이 있다[1]. 세 번째는 기존의 컴파일 방식을 이용하여 중간코드인 바이트코드를 특정 프로세서에서 수행될 수 있는 목적코드로 바꾸는 컴파일러의 후단부를 이용하는 방법으로 CACAO[2], Toba[3], JCC[4]등이 있다. 네 번째 바이트코드는 스택을 기반으로 하는 코드이므로 효율적인 코드로 변환하는 방법으로 DashO[5], JOIE[6]등이 있다[7].

그러나 위와 같은 연구에서 개별적인 고유한 최적화 기법과 구현된 프로그램을 이용해서만 최적화가 가능했다. 최적화 기법을 적용하기 위해서는 바이트코드 추출기부터 최적화를 위한 부분까지 모두 구현을 해야만 한다.

본 논문에서 제시하는 바이트코드 최적화 프레임워크는 바이트코드를 3주소 중간코드 형태로 변환시킴으로써 3주소 코드 최적화 알고리즘을 직접 대입하여 사용 가능하게 해준다.

본 논문은 2장에서 기존의 연구에 대해서 알아본다. 3장에서는 바이트코드 최적화 프레임워크를 설계한다. 마지막으로 4장에서는 결론과 향후 연구 방향에 대해서 기술한다.

2. 관련 연구

2.1 중간 코드 변환기

중간 코드 변환기는 바이트코드 추출기, 코드 확장기, 코드 출력기로 구성된다. 바이트코드 추출기는 클래스 파일을 입력으로 받아 바이트코드를 추출한다. 바이트코드 추출기는 Sun사에서 제공하는 javap 명령어의 "-c" 옵션을 이용하여 바이트코드를 출력한다. 코드 확장기법은 바이트코드에 해당하는 변환 루틴을 호출하여 코드를 생성한다. 그러나 코드확장 기법은 빠르게 변환이 가능하다는 장점을 갖지만 생성된 코드의 질이 효율성이 떨어질 수 있으므로 최적화 동작이 수행 되어야한다는 단점을 갖는다[8]. 바이트 코드 추출기로부터 생성된 각각의 바이트코드에 대해 해당하는 루틴을 호출하는 형식은 [그림 2-1]와 같다.

```
switch( bytecode ) {
  case ICONST_0: constant_push(integer, 0, 32);
                 break;
  ...
  case POP:      pop( 1 );
                 break;
 ...
}
```

[그림 2-1] 코드변환 루틴 호출 예제

ICONST_0항목은 호출 형식을 “constant_push(type, value, size)”를 갖는다. 매개변수로 type, value, size는 각각 입력 값의 타입과 값 그리고 크기를 표현한다. 그리고 POP항목은 호출형식을 “pop(count)”를 갖는다. 매개변수로 count는 스택에서 명령어를 꺼내는 개수를 표현한다.

2.2 바이트코드 최적화 도구

DashO는 클래스파일의 크기를 줄이는 방법을 이용한다. 클래스파일을 분석해서 참조되는 패키지(package)만을 클래스파일 내에 포함시켜서 클래스파일의 크기를 줄이는 방법을 사용한다. 줄여진 클래스파일을 전체를 압축하여 다운로드하여 사용할 수 있는 방법을 제공한다[5].

JOIE는 동적인 클래스 로딩기능을 개선하였다. 동적인 클래스 로딩은 자바프로그램에서 다른 자바프로그램에 대해서 자바 가상기계에 로딩을 필요로 할 경우 로드와 동시에 해제한다. 이런 동적인 로딩기능을 효율적으로 할 수 있는 방법과 메소드 분석을 용이하게하는 방법도 제시하고 있다[6].

2.3 바이트코드를 네이티브코드 변환 도구

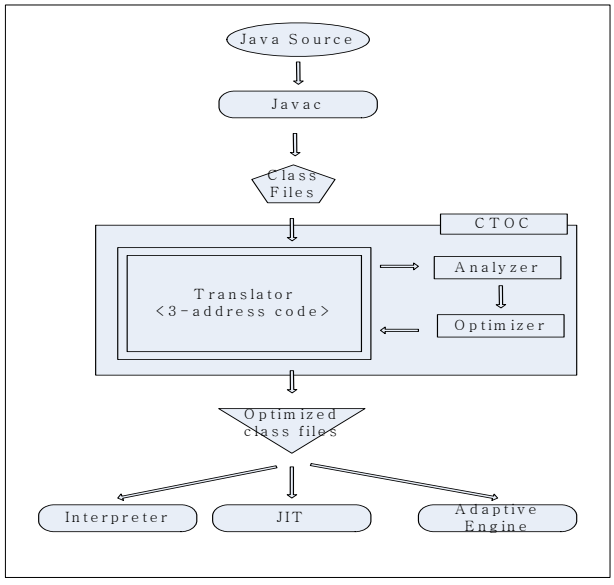
CACAO는 바이트코드를 Alpha프로세서의 네이티브 코드로 변환하는 시스템이다. 변환하는 동안 스택 기반 바이트코드를 레지스터 기반의 중간코드로 변환한다. 지역변수와 스택의 위치는 제한된 32비트 주소 타입의 의사 레지스터 코드로 교환되고 제거된다. 빠른 레지스터 할당 알고리즘을 이용하여 의사 레지스터 코드를 머신 레지스터로 사상되는 방법을 사용한다[2].

Toba는 자바 바이트코드를 C언어로 변환한 후에 C컴파일러를 이용하여 네이티브 코드를 생성한다. Toba의 구성은 바이트코드를 C언어로 변환시키는 변환기, C언어 컴파일러, 가비지 컬렉터, 스레드 패키지, 자바 API지원 부분들을 포함하고 있다[3].

JCC는 지역 최적화 컴파일러이다. JCC는 자바 소스로부터 직접 실행이 가능하다. JCC시스템은 대부분 사용하는 파서 생성기와 컴파일러의 전단부와 후단부를 포함하고 있다. 또한 중간 표현으로 레지스터와 스택을 모두 사용하는 3주소형태의 중간 코드를 사용한다[4].

3. 바이트코드 최적화 프레임워크의 설계

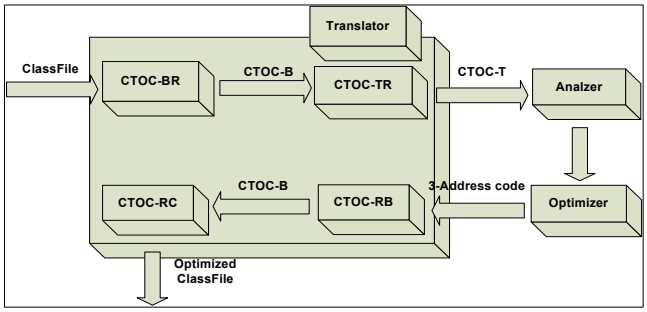
본 논문에서 제안하는 바이트코드 최적화 프레임워크는 CTOC(Class To Optimizer classes)라고 부른다. CTOC의 전체적인 구성은 translator, Analyzer, Optimizer로 3부분으로 나뉘고 [그림 3-1]과 같다.



[그림 3-1] CTOC의 구성도

3.1 Translator

Translator는 네 부분으로 나누어진다. CTOC-BR (Class To Optimize Classes - Bytecode tRanslator)은 바이트코드를 추출해서 CTOC-B (Class To Optimizer Classes-Bytecode)로 변화하는 부분이다. CTOC-TR(Class To Optimizer Classes-Three address code tRanslator)은 CTOC-B를 3주소 형태의 CTOC-T(Class To Optimizer Classes-Three Address Code)로 변환하는 부분이다. CTOC-RB (Class To Optimizer Classes-Return Bytecode)은 3주소 형태의 CTOC-T를 CTOC-B로 변환하는 부분이다. CTOC-RC(Class To Optimizer Classes -Return ClassFile)은 CTOC-B를 클래스파일로 변환시키는 부분이다. 자세한 내부 구성은 [그림 3-2]와 같다.



[그림 3-2] CTOC의 내부 구성도

바이트코드는 타입이 없는 명령어이다. 때문에 분석하기 어려운 문제점을 가지고 있다. 이러한 문제점을 해결하기위해 CTOC-B로 변환한다. 변환 방법은 바이트코드 명령어에 타입을 붙여서 표현하고 각각의 변수에 대한 타입으로 var라는 타입을 갖는다. var타입은 변수를 담기위한 자료구조의 형식이다. 변환방법의 원리는 바이트코드에 대해서 CTOC-B로 단순한 패턴매칭기법을 이용하여 변환할 수 있다.

예를 들어 [그림 3-3]의 iadd를 I.add 형태로 타입이 있는 형태로 변환한다.

iload_1	I.load l1
iload_2	I.load l2
iadd	I.add
istore_3	I.store l3
(a) 바이트코드	(b) CTOC-B

[그림 3-3] 중간 코드 변환의 예

CTOC-TR을 이용해서 CTOC-B를 3주소 형태인 CTOC-T로 변환한다. CTOC-T로 변환하면 기존의 3주소 최적화 기법을 적용할 수 있고, 고급언어와 형태가 비슷하기 때문에 분석하기 쉽다.

그러나 CTOC-B는 스택기반이기 때문에 스택을 사용하던 부분들에 대한 임시변수들을 생성해야한다. 임시변수의 수는 스택을 참조하는 수와 같고 스택변수는 변수의 이름 앞에 기호 '&'를 붙여서 표현한다. 예를 들어 [그림 3-4]와 같이 나타낼 수 있다.

var l1,l2,l3	int &r0, &r1, a, b, c
I.load l1	r0 = a
I.load l2	r1 = b
I.add	r0 = r1 + r2
I.store l3	z = r0
(a) CTOC-B	(b) CTOC-T

[그림 3-4] 3주소 형태 변환의 예

3주소 형태의 코드를 바이트코드로 바로 변환을 할 경우 원래의 코드와 같은 실행을 한다는 보증을 할 수 없다. 따라서 원래의 코드와 같은 실행을 보증하기 위해서 CTOC-B로 변환 후에 원래의 클래스파일로 환원시킨다.

3.2 Analyzer

Analyzer는 3주소 형태의 표현을 최적화하기 위해서 필요한 정보를 얻기 위해서 사용한다. analyzer는 흐름분석을 통해 제어 흐름과 데이터 흐름에 대한 정보를 Optimizer에게 제공한다. 따라서 Analyzer는 제어 흐름을 분석하여 기본 블록을 생성한다. 기본 블록은 기본 블록 리덕션(Reduction)을 행하여 기본 블록의 수를 줄인다. 리덕션은 두 단계로 진행 된다. 첫 번째 단계에서는 제어흐름이 명백하게 기본블록이 나누어질 때까지 반복된다. 두 번째 단계에서는 첫 번째 단계에서 기본 블록의 개수를 줄이는데 장애가 되는 요인을 제거하고 첫 번째 단계를 다시 적용하여 기본블록의 수를 줄인다. 따라서 Analyzer는 기본 블록을 이용한 최적화를 적용할 수 있는 자료를 Optimizer에게 알려준다. 또한 사용-정의 고리와 정의-사용 고리, 그리고 도달 정의에 대한 정보를 Optimizer에 제공한다.

3.3 Optimizer

Optimizer는 3주소 형태의 두 가지 최적화 방법을

제공한다. 자동으로 최적화를 적용하는 방법과 직접 3주소 형태의 최적화 기법을 적용하는 방법을 제공한다. 자동으로 최적화를 적용하는 방법은 기존의 최적화 기법을 이용해서 최적화된 코드를 생성하는 것을 의미한다. 기존에 최적화 기법인 지역적 최적화 방법과 전역적인 최적화 방법을 적용할 수 있다. 최적화 기법은 전역적 공통부분 없애기, 귀납변수 없애기, 공통부분식의 제거, 의미가 보존된 함수 변환, 복사 전파, 죽은 코드 없애기, 루프 최적화, 코드 위치 바꾸기등을 적용한다. 직접 최적화 기법을 적용하는 경우는 CTOC-T에 대해서 집접 새로운 3주소 최적화 알고리즘을 적용하여 최적화된 코드를 얻을 수 있다.

4. 결론 및 향후 연구

본 논문에서 제안하는 바이트코드 최적화 프레임워크는 자바에서 문제점으로 제시되고 있는 실행시간 문제를 해결할 수 있도록 설계하였다.

기존의 바이트코드 최적화 툴은 최적화 툴에서 사용하는 최적화 기법을 이용하여 최적화하므로 최적화 기법을 연구하기 위해서는 컴파일러의 후반부 구현하여 적용해야 하는 문제점이 있었지만, 바이트코드 최적화 프레임워크는 최적화 알고리즘을 바로 적용할 수 있는 기반을 제공해준다.

앞으로 향후 연구과제는 바이트코드 최적화 프레임워크를 구현하고 프레임워크에서 사용할 수 있는 API의 설계하고 구현할 예정이다.

참고문헌

[1] John Meyer, Troy Downing, "Java Virtual Machine", O′ RELLAY, 1997

[2] A. Krall and R. Grafl, "CACAO - A 64 bit Java VM Just-in-time Compiler", Appeared at PPoPP'97 Workshop on Java for Science and Engineering Computation, 1997

[3] Todd A. Proebsting, Greg Townsend, Patrick Bridges, "Toba: Java For Applications A Way Ahead of Time(WAT) Compiler", COOTS97, pp. 41-53, 1997

[4] Ronald Veldema, "JCC, a native Java compiler",Technical report, 1998

[5] preEmptive Solutions, "DashO Whitepaper", http://preemptive.com/downloads/documentation.html, 2002-2004

[6] Geoff Cohen (Duke/IBM), Jeff Chase (Duke), David Kaminsky (IBM),"Automatic Program Transformation with JOIE", in Proceedings of the 1998 USENIX Annual Technical Symposium, 1998

[7] Taiana Shpeisman, Mustafa Tikir, "Generating Efficient Stack Code for Java", Technical report, University of Maryland, 1999

[8] Frank Yellin, "The JIT Compiler API", http://java.sun.com/docs/jit_interface.html, 1996