바이트코드 프레임워크 설계

Design of Bytecode Framework

김영국*, 김기태*, 조선문*, 이갑래**, 유원희*
인하대학교 컴퓨터정보공학과*, 김천 과학 대학**

Kim Young-Kook*, Kim Ki-Tae*, Jo Sun-Moon*,
Lee Kab-Lae**, Yoo Weon Hee*
Department Computer Science &
Information Inha University*
Kimcheon Science College**

요약

자바 바이트코드는 스택기반 코드이다. 스택기반 코드는 스택 접근 명령어를 사용하기 때문에 분석과 최적화를 어렵게 한다. 따라서 스택기반 코드 최적화에서 생기는 문제점인 코드의 단편화, 타입정보의 상실, 불필요한 적재와 저장이 나타날 수 있다. 바이트코드의 최적화와 분석의 어려운 문제점의 해결 방안으로 바이트코드 프레임워크를 설계한다.

본 논문은 바이트코드의 최적화와 분석의 문제점을 지적하고, 기존의 바이트코드 최적화 기술에 대한 연구 내용을 서술한다. 바이트코드의 분석과 최적화를 단순화하기 위한 대안으로 바이트코드 프레임워크를 제안한다.

Abstract

Java bytecode is stack-base code. Stack-base code makes analysis and optimization hardly because use stack access imperative. Therefore, fragment of code that is problem that occur in stack-base code optimization, loss of type information, unnecessary Load and Store can appear. Optimization and analysis of bytecode design bytecode framework by solution way of problem that is difficult.

This paper indicates optimization of bytecode and hangup of analysis, and describe research contents about existent byte code optimization technology. This propose byte code framework by the alternative to simplify analysis and optimization of byte code.

I. 서 론[1)]

자바는 객체지향 언어이다. 또한 플랫폼에 독립적이기 때문에 프로그램의 유지보수의 비용이 적게 드는 매력을 가진다. 자바는 플랫폼 독립적인 설계상의 특징으로 바이트코드의 중간표현과 가상기계 두 가지를 들 수 있다. 바이트코드는 어셈블리와 비슷한 표현을 사용하며 스택을 기반으로 한다. 따라서 자바 바이트코드는 스택 접근 명령어를 정의하고 있다. 가상기계는 바이트코드를 인터프리터 방식으로 네이티브 코드를 생성하여 프로그램을 실행한다. 가상기계는 인터프리터방식을 이용하기 때문에 코드의 크기가 커질수록 실행에 많은 비용이 드는 단점을 가진다. 따라서 실행에 드는 비용 중에 시간을 줄이기 위한 많은 연구가 진행 중이다.

실행시간에 드는 비용을 줄이기 위한 방법으로 JIT

1) 본 연구는 한국과학재단 목적기초연구(31746)지원으로 수행되었음.

컴파일러[1], 네이티브 코드로 변환 방식, 그리고 바이트코드 최적화하는 방식 등이 있다. JIT컴파일 방식은 실행시간에 메소드 단위로 컴파일하는 방식을 말한다. 네이티브 코드로 변환을 이용하는 방식은 JCC[2], Toba[3], CACAO[4]등이 있다. 그리고 바이트코드 최적화 방식에는 크게 전역최적화와 지역최적화를 이용하는 방식이 있다.

그러나 바이트코드 최적화기법은 바이트코드 단위에서 실행하기 때문에 기존의 주로 연구되었던 3주소 최적화 기법을 바로 적용할 수 없는 문제점을 가진다. 또한 바이트코드는 코드의 단편화와 타입 정보의 상실 그리고 스택접근 명령어가 프로그램의 분석을 어렵게 하는 단점을 가진다.

본 논문은 2장에서 바이트코드의 최적화와 분석의 문제점을 지적하고, 기존의 바이트코드 최적화 기술과 중간코드 변환기에 대해서 서술한다. 3장에서는 바이트코드의 분석과 최적화를 단순화하는 바이트코드 프레임워크를 제안한다. 마지막으로 4장에서는 결론과 향후 연구 방향에 대해서 기술한다.

II. 관련 연구

1. 바이트코드 최적화와 분석의 문제점

바이트코드의 최적화의 문제점은 코드의 단편화, 타입정보의 상실, 스택접근 명령어가 프로그램의 분석을 어렵게 만든다.

코드의 단편화 문제점은 복잡한 수식의 경우 여러 조각으로 나누어서 코드상의 넓게 분포되어 나타나는 경우를 말한다. 다음 [그림 2.1]에서 나타나 있다.

[그림 2.1]을 보면 바이트코드에서 자바소스에서 int 타입으로 선언되었던 변수들의 타입 상실을 볼 수 있다. 이런 내용들은 바이트코드의 분석을 어렵게 하는 이유 중의 하나이다.

그리고 [그림 2.1]에서 바이트코드를 보면 모든 값들은 스택에 넣어서 연산을 한다. 따라서 스택 접근 연산이 많아지게 된다.

(a) 자바 소스	(b) 바이트코드

int x=1, y=1;
 	
if (x!= 0) x+=y*3;
else y=y*2;
	0: iconst_1
	1: istore_1
	2: iconst_1
	3: istore_2
	4: iload_1
	5: ifeq 17
	8: iload_1
	9: iload_2
	10: iconst_3
	11: imul
	12: iadd
	13: istore_1
	14: goto 21
	17: iload_2
	18: iconst_2
	19: imul
	20: istore_2
	21: return

▶▶ 그림 2.1 코드단편화의 예제

2. 바이트코드 최적화 기술

바이트코드의 최적화 기술은 기본 블록을 이용하며 기본 블록간의 흐름분석을 통해서 전역최적화를 적용하고, 기본 블록 내에서 최적화를 적용하는 지역최적화 방법이 있다.

최적화의 주요원리는 의미가 보존된 함수의 변환, 공통부분식의 제거, 복사전파, 죽은 코드 없애기, 루프 최적화, 코드 위치 바꾸기, 귀납변수와 값싼 연산 만들기가 있다.

첫째 의미가 보존된 함수의 변환은 컴파일러가 함수들의 의미를 변하지 않으면서 프로그램을 변형시키는 것을 의미한다. 둘째 공통부분식 제거는 어떤 식 E가 프로그램에서 한번 이상 나타나며 변수 E의 값은 한번 계산된 후 변하지 않는 값을 계산식과 대치하여 사용하는 방법을 의미한다. 셋째 복사전파는 죽은 코드를 판별할 수 있는 기회를 제공하기 때문에 복사전파는 죽은 코드 없애기랑 같이 사용된다. 넷째 죽은 코드 없애기는 특정한 지점 이후에 사용되지 않는 문장을 제거한다. 다섯째 루프 최적화는 루프내의 수행시간을 줄이는 방법을 말한다. 여섯째 코드 위치 바꾸기는 루프 내 코드 중에 루프를 수행하는 동안

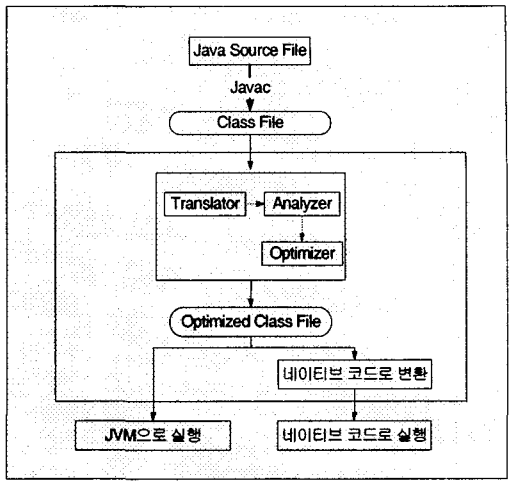
항상 같은 값을 생성하는 식이 있다면 루프 앞으로 이동하는 것을 의미한다. 일곱째 귀납변수의 제거에서 귀납변수는 앞의 변수 값에 의해서 변수의 값을 유추 할 수 있는 변수를 제거하는 것을 의미한다. 여덟째 값싼 연산 만들기를 사용하여 명령어의 의미가 같은 명령어로 바꾸는 것을 의미한다[5,6].

3. 중간코드 변환기

중간 코드 변환기는 바이트코드 추출기, 코드 변환기, 코드 변환 테이블로 구성된다. 첫째 바이트코드 추출기는 클래스 파일을 입력으로 받아 바이트코드를 추출한다. 둘째 코드 변환기는 코드 변환 테이블을 참조하여 바이트코드에 해당하는 코드를 생성한다. 셋째 코드 변환 테이블은 코드 변환의 정보를 저장하는 부분으로서 바이트코드에서 유사한 기능을 갖는 명령어 그룹을 구성한다. 그러나 코드확장기법은 빠르게 변환이 가능하다는 장점을 갖지만 생성된 코드의 질이 효율성이 떨어질 수 있으므로 최적화 동작이 수행 되어야한다는 단점을 갖는다[7].

III. 바이트코드 프레임워크

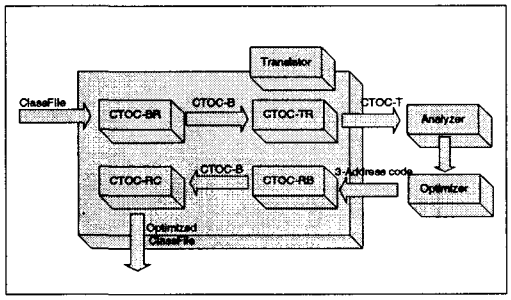
바이트코드 프레임워크는 본 논문에서는 CTOC (Class To Optimizer Class)라고 부른다. CTOC는 Translator, Analyzer, Optimizer, 네이티브 코드로 변환등으로 구성 되어있고 [그림 3.1]에 나타나있다.



▶▶ 그림 3.1 바이코드 프레임워크

1. Translator

Translator는 CTOC-BR(Class To Optimizer Classes-Bytecode tRanslator), CTOC-TR (Class To Optimizer Classes-Three address code tRanslator), CTOC-RB (Class To Optimizer Classes-Return Bytecode), CTOC-RC(Class To Optimizer Classes-Return Classfile) 등의 4단계로 구성된 모습은 [그림 3-2]와 같다.



▶▶ 그림 3.2 Translator의 내부구성도

CTOC-BR은 클래스파일에서 바이트코드를 추출할 때 타입정보를 생성한다. 추출한 타입정보를 이용하여 CTOC-B(Class To Optimizer Classes-Bytecode)를 생성해낸다. CTOC-B는 니모닉 코드

에 타입을 붙여서 사용하기 때문에 바이트코드와 비슷한 모양을 갖는다. CTOC-B의 예를 들면 iadd라는 바이트코드의 표현을 I.add로 나타낼 수 있다. 또한 바이트코드에서 CTOC-B로 변환할 때의 문제점은 dup종류나 swap의 경우 추상 스택 해석을 통해서 타입을 추론해서 표현한다.

CTOC-TR은 CTOC-B를 CTOC-T(Class To Optimizer Classes Three address code)로 변환한다. CTOC-T는 타입이 있는 3주소코드이기 때문에 스택 대신에 임시변수를 생성해주기 위한 최대 스택 크기는 클래스 파일에서 메소드 단위로 계산해 놓은 최대 스택 크기를 이용하여 메소드 단위로 3주소 코드를 생성해낸다.

CTOC-RB은 CTOC-T를 CTOC-B로 변환한다. CTOC-T에서 CTOC-B로 변환할 때 생기는 불필요한 구문들은 핍홀 최적화를 수행하여 3주소에서 바이트코드로 변환할 때 생기는 문제점을 해결한다.

CTOC-RC은 CTOC-B를 클래스파일로 환원한다.

2. Analyzer

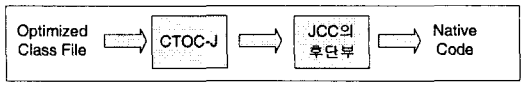
Analyzer는 프로그램 분석과 최적화에 필요한 정보를 생성한다. Analyzer는 흐름분석을 통해 제어 흐름과 데이터 흐름에 대한 정보를 생성하고 생성된 정보를 Optimizer나 프로그램 분석을 위해서 사용된다. 따라서 Analyzer는 제어 흐름을 분석하여 기본 블록을 생성한다. 기본 블록은 리더를 설정하여 기본 블록을 생성한다. 생성한 기본 블록은 리덕션(Reduction)을 실행하여 기본 블록의 수를 줄인다. 리덕션은 두 단계로 진행된다. 첫 번째 단계에서는 제어흐름에 알맞은 기본블록으로 나누어질 때까지 반복된다. 두 번째 단계에서는 첫 번째 단계에서 기본 블록의 개수를 줄이는데 장애 요인을 제거한다. 그 후에 첫 번째 단계를 반복 적용하여 기본블록의 수를 줄인다.

3. Optimizer

Optimizer는 3주소 형태의 최적화 방법을 제공한다. 자동으로 최적화를 적용하는 방법과 직접 3주소 형태의 최적화 기법을 적용하는 방법을 제공한다. Optimizer에서 제공하는 최적화 기법은 기존의 최적화 기법을 이용해서 최적화된 코드를 생성한다. 기존에 최적화 방법을 적용할 수 있다. 직접 최적화 기법을 적용하는 경우는 CTOC-T에 대해서 직접 새로운 3주소 최적화 알고리즘을 적용하여 최적화된 코드를 생성한다.

4. 네이티브 코드로 변환

네이티브 코드로 변환의 구성은 [그림 3-3]과 같다.



▶▶ 그림 3.3 Native Code로의 변환의 구성도

CTOC-J(Class To Optimized Classes-JCC)는 최적화된 클래스 파일을 JCC의 중간 표현으로 변환하는 부분이다. CTOC-J는 CTOC-BR을 이용하여 바이트코드를 추출하여 CTOC-B를 생성한다.

CTOC-B를 코드변환 테이블을 이용하여 각각의 바이트코드에 대응하는 코드 확장 기법을 사용하여 JCC의 중간표현인 gasm코드를 생성한다. 생성된 gasm코드는 JCC의 후단부를 이용하여 네이티브 코드를 생성한다.

IV. 결론 및 향후 연구

본 논문에서 제안하는 바이트코드 프레임워크는 자바에서 가장 문제시 되고 있는 실행시간에 대한 비용을 줄이는 방법을 제안한다.

기존의 사용하던 바이트코드 최적화 툴은 각각의 최적화 방법을 사용하므로 사용자가 직접 최적화 알고리즘을 적용할 수가 없었다. 따라서 바이트코드 프

레임워크는 최적화 알고리즘을 바로 적용할 수 있는 기반을 제공해준다.

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

▌참고문헌▐

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

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

[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] 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.

[5] Steven S. Muchnick, "Advanced Compiler Design and Implementation", Morgan Kaufmann, 1997.

[6] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ulman "Compilers principles, Techniques and Tools", Addison-Wesley, 1988.

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