韓國 컴퓨터情報學會 論文誌
第11卷 第1號, 2006. 3.

2006-11-1-1-12

바이트코드 분석을 위한 중간코드에 관한 연구

김 경 수*, 유 원 희**

A Study on Intermediated code for Analyzing Bytecodes

Kyung-Soo Kim*, Weon-Hee Yoo**

요 약

자바 언어는 객체지향 언어이며 다양한 개발 환경과 이식성에 맞는 언어로써 각광을 받고 있다. 하지만 자바 언어는 실행속도가 느리다는 단점을 가지고 있다. 이러한 이유는 자바 프로그래밍 환경에서 자바 가상 기계 코드인 바이트코드가 인터프리터 방식으로 사용되기 때문이다. 따라서 프로그램의 수행에는 실행 속도가 현저히 저하되는 단점이 발생하게 된다. 또한 자바 언어는 컴파일러를 통해 생성된 클래스 파일에 프로그램의 수행과 관련된 정보가 숨겨져 있다. 클래스 파일의 분석으로 바이트코드를 위한 효율적인 분석 및 최적화를 할 수 있다. 본 논문에서는 자바 클래스 파일의 정보들을 이용해 자바 바이트코드 분석을 하려 한다. 분석을 위해 정적 단일 배정문 형태로 변환하게 되는데 정적 단일 배정문 형태는 정의-사용 체인에서 변형된 형태이다. 정적 단일 배정문 형태는 각각의 타입들을 오직 한번만 배정하고 재명명함으로써 프로그램을 정적으로 분석 할 수 있게 한다. 정적 단일 배정문 형태는 최적화와 분석을 위한 효과적인 중간 코드이다.

Abstract

Java language creates class files through Java compiler. Class files include informations involved with achievement of program. We can do analysis and optimization for efficient codes by analyzing class files. This paper analyzes bytecodes using informations of Java class files. We translate stack-based Java bytecodes into 3-address codes. Then we translate into static single assignment form using the 3-address codes. Static single assignment form provides a compact representation of a variable's definition-use information. Static single assignment form is often used as an intermediate representation during code optimization. Static single assignment form renames each occurrence of a variable such that each variable is defined only once.

▸ Keyword : 정적 단일 배정문(Static Single Assignment), 최적화(optimazation), 바이트코드(Bytecode)

• 제1저자 : 김경수
• 접수일 : 2006.01.12, 심사완료일 : 2006.03.13
* 인하대학교 컴퓨터공학과 석사, ** 인하대학교 전자계산공학과 교수
※ 이 논문은 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(R05-2004-000-11694-0)

Ⅰ. 서 론

프로그램 분석에 있어 데이터 흐름 분석이 중요하다. 정의와 사용 패턴은 데이터 흐름에서 필요한 부분이다. 데이터 흐름 분석에서 변수의 정의는 새로운 값이 배정될 때 발생하고 변수의 사용은 변수의 값이 참조될 때 발생한다. 프로그램 내에서 정의와 사용을 정의-사용 체인[1]이라고 불린다. 프로그램의 모든 변수는 정의되고 사용된다. 또한 각 변수는 타입을 갖게 된다. 정적 단일 배정문 형태(SSA Form: Static Single Assignment Form)는 정의-사용 체인(definition-use chain)에서 변형된 형태이다. 정적 단일 배정문 형태는 코드의 분석 및 최적화를 하기 위한 중간 표현으로써 각각의 변수들이 가지고 있는 값뿐만 아니라 각각의 타입들을 오직 한번만 배정하고 재명명함으로써 프로그램을 정적으로 분석할 수 있게 한다. 즉, 정의-사용 체인에서 정의 되어지는 배정문의 왼쪽에 유일한 이름을 설정함으로써 각각의 사용에 대한 정확한 정보를 갖게 하는 것이다. 따라서 정적 단일 배정문 형태를 이용하여 프로그램의 분석뿐만 아니라 죽은 코드 제거, 상수 전파, 타입추론, 부분 중복 제거[2]등과 같은 최적화 기법을 적용할 수 있다.

본 논문에서는 정적 단일 배정문 형태를 구하기 위해 지배자 경계(DF: Dominance frontier)[3]를 계산해야 한다. 지배자 경계를 계산하는 이유는 최소한의 병합노드를 구하기 위해서이다. 지배자 경계를 계산하기 위해서 제어 흐름 그래프와 지배자 트리(dominator tree)가 필요하다. 지배자 트리는 각 블록간의 지배관계를 알기 위해서 구하는 것으로 제어 흐름 그래프를 이용해서 만들게 된다. 지배자 경계를 이용하여 ∅-함수를 삽입 할 위치를 결정하게 된다. ∅-함수 삽입 후에 새로 정의된 변수들에 대해서 변수 이름을 바꿔주게 되면 정적 단일 배정문 형태가 된다.

본 논문은 다음과 같이 여섯 장으로 구성된다. 2장은 관련연구로 자바 클래스 파일 기본 블록 그리고 정적 단일 배정문에 대한 내용을 다룬다. 3장에서는 바이트코드를 3-주소 코드로 변환하는 내용으로 라인 정보 테이블, 3-주소 코드로 변환 대한 내용을 다룬다. 4장에서는 본 논문에서 사용하게 될 정적 단일 배정문 형태를 설계하고 설계에 필요한 지배자 트리와 지배자 경계를 구한다. 5장에서는 정적 단일 배정문 형태를 구현하고 성능 평가를 한다. 6장에서는 본 논문의 결론을 제시하고 향후 과제를 서술한다.

Ⅱ. 관련연구

2.1 자바 클래스 파일

자바 클래스 파일은 바이너리 형태로써 자바 가상기계에서 실행되어 실행 결과를 생성한다. 이러한 클래스파일의 구조는 실행에 필요한 모든 정보를 저장[4]하고 있으므로 그 내용 및 구조[5]를 이해하는데 많은 기초적인 지식을 필요로 한다. 바이너리 형태로 구성된 클래스파일은 자바 가상기계에서 올바른 수행을 위한 많은 정보를 가지고 있으며 [9] 실질적으로 클래스 파일의 구조 및 정보를 추출하기 위한 도구들이 사용되고 있다. 자바 파일 이름과 클래스 파일의 이름은 같게 되며, 자바 소스 파일 안에 있는 클래스의 개수만큼 생성되게 된다. 즉, 소스 파일 안에 아무리 많은 클래스가 정의되어 있더라도 생성되는 클래스 파일은 정의된 클래스 파일 당 하나씩 존재 하게 되는 것이다. 클래스 파일은 8비트 크기의 이진 흐름으로 구성되며 데이터 아이템들은 순차적으로 클래스 파일에 저장이 되게 된다. 모든 16비트와 32비트 크기들은 2개 혹은 4개의 8비트를 각각 읽음으로써 구성되며, 1바이트 이상 차지하는 몇 개의 연속된 바이트에 높은 바이트를 먼저 써 넣는 방식(big-endian)으로 만들어지게 된다.

```
ClassFile{    u4 magic;
u2 minor_version;
u2 major_version;
cp_infoconstant_pool
[constant_pool_count -1];
u2 access_flags;
u2 this_class;
u2 super_class;
u2 interfaces[interface_count];
u2 fields_count;
field_info fields[fields_count];
u2 methods_count;
field_info methods[fields_count];
u2 attributes_count;
attribute_info attributes
[attributes_count];}
```

그림 1. 클래스 파일의 구조
Fig 1. Structure of Class File

클래스 파일의 구조는 (그림 1)과 같다. (그림 1)에서 u1,u2,u4는 각각 unsigned 1,2,4바이트 값을 의미한다.

바이트코드는 8비트 연산 코드를 갖는 200여개가 넘는 명령어로 구성되어 있으며 스택을 기반[6]으로 실행된다. 즉, 실행을 위해 오퍼랜드 스택을 이용하고 명령어 실행 전 오퍼랜드들을 스택에 적재하고 결과 값도 스택에 적재한다. 레지스터는 없고 대신에 지역변수를 레지스터처럼 사용한다. 대부분의 바이트코드 명령어는 오퍼랜드 타입을 명시한다. 예를 들면 iadd 명령어는 정수 덧셈을 의미한다. 같은 방법으로 다른 명령어 isub, iload, istore 등에도 타입을 명시한다. 복잡한 바이트코드 명령어로는 메모리 할당, 모니터, 스레드 동기화, 메소드 호출 등을 위한 명령어를 들 수 있다.

2.2 기본 블록

기본블록은 제어가 정지나 분기의 가능성이 없는 연속적인 문장들의 집합을 말한다. 기본블록을 나누는 데는 리더[4]를 찾아야 한다. 리더의 정의를 살펴보면 먼저 첫 번째 문장은 리더이다. 두 번째로 무조건 분기문의 목표가 되는 문장은 리더이다. 세 번째로 분기문 또는 조건 분기문 바로 다음에 오는 문장은 리더이다. 각 리더마다 기본 블록은 다음리더나 프로그램의 끝을 만나기 바로 전까지 모든 문장으로 구성된다고 할 수 있다.

1. 메소드의 첫 번째 명령어와 메소드를 위한 모든 핸들러의 첫 번째 명령어는 리더이다.
2. 조건 없는 분기의 목표가 되는 각 명령어는 리더이다.
3. 조건 분기의 목표가 되는 각 명령어는 리더이다.
4. 테이블 점프 명령어와 같이 복합 조건 분기의 목표 중 하나인 경우 각 명령어는 리더이다.
5. 복귀 명령어와 조건 또는 비조건 분기의 바로 뒤에 나타나는 각 명령어는 리더이다.

그림 2. 자바 바이트코드 리더의 정의
Fig 2. Definition of Java Bytecode Leader

바이트코드는 스택을 기반으로 하는 코드이기 때문에 리더는 3-주소 코드에서의 리더와는 다르게 된다. (그림 2)는 바이트코드 리더의 정의를 나타내고 있다. 〈표 2〉는 바이트코드의 분기 명령어를 보여주고 있다. 본 논문에서는 라인 정보 테이블을 이용하여 바이트코드의 기본 블록을 나누게 된다.

2.3 정적 단일 배정문

정적 단일 배정문 형태는 정의-사용 관계에서 개선된 형태로써 프로그램 상에서 각 변수들이 서로 다른 값과 다른 타입을 가질 수 있기 때문에 변수는 값이나 타입에 따라 분리 될 수 있다. 이러한 형태를 정적 단일 배정문

형태라고 한다. 정적 단일 배정문 형태는 데이터 분석이

표 1. 바이트코드 분기 명령어
Table 1. Bytecode Branch Instruction

명령어 종류	바이트코드 명령어
조건 분기	ifeq, ifne, iflt, ifle, ifnull, ifnonnull, if_acmpeq, if_icmple, if_icmpge
비교	lcmp, fcmp, dcmpg, fcmpg
무조건분기	goto, goto_w
서브루틴	jsr, jsr_w, ret
예외처리	athrow
테이블 점프	tableswitch, lookupswitch
메소드	*return, ireturn, lreturn, freturn, dreturn*

나 코드 최적화를 위한 컴파일러 중간 표현으로 주로 사용된다. 정적 단일 배정문 형태를 이용하여 죽은 코드 제거, 상수 전파, 타입추론, 부분 중복 제거등과 같은 최적화 기법을 적용할 수 있다. (그림 3)은 정적 단일 배정문 형태를 변환하는 과정을 나타내고 있다.

1. 제어 흐름 그래프에서 지배자 트리를 구한다.
2. 지배자 트리와 제어 흐름 그래프를 이용하여 지배자 경계를 구한다.
3. 지배자 경계를 이용하여 ø-함수를 삽입한다.
4. 지배자 트리를 순행하면서 변수의 이름을 재정의 한다.

그림 3. 정적 단일 배정문 형태 변환 과정
Fig 3. Translator Procedure of Static Single Assignment Form

III. 바이트코드를 3-주소 코드로 변환

3.1 라인 정보 테이블

자바 바이트코드는 스택 기반의 코드이기 때문에 복잡한 수식의 경우 여러 조각으로 나누어서 코드상의 넓게 분포되어 나타나는 코드 단편화 문제점과 스택 접근 연산들을 사용하여 프로그램 분석이 용이 하지 않고, 단순한 변환을 복잡하게 만들 수 있다는 단점이 있다. 이처럼 분석이 용이 하지 않은 바이트코드를 분석하기 위해서는 분석이 용이한 형태로 변환해야 한다.

3-주소 코드로 변환하기 위해 바이트코드의 라인 정보와 실제 자바 소스의 명령어 라인 정보를 가지고 있는 라인 번호 테이블을 이용한다. 라인 번호 테이블은 클래스

자바소스	label을 추가한 바이트코드

public class Temp {
int f(boolean b){
int x;
x = 1;
if(b)
x = 2;
else
x = 3;
return x;
}
}
	label_0
	ldc 1
	istore x$2
	label_2
	iload b$1
	ifeq label_11
	label_6
	ldc 2
	istore x$2
	goto label_13
	label_11
	ldc 3
	istore x$2
	label_13
	iload x$2
	ireturn

그림 4. label 추가
Fig 4. Label Addition

파일 내부에 있는 정보로써 실제 자바소스와 바이트코드의 라인정보에 대해서 갖고 있다. 라인 번호 테이블에서 실제 바이트코드의 시작 위치에 대한 정보를 얻어 그 위치에 레이블을 추가하여 기본 블록을 생성하게 된다. (그림 4)은 레이블을 추가한 모습을 보여주고 있다. 레이블의 번호는 라인 번호 테이블에 있는 바이트코드 시작위치(start_pc)이다.

3.2 3-주소 코드

3-주소 코드는 제어 흐름 그래프를 만들기 전에 수행된다. 트리구조 3-주소 코드는 바이트코드를 트리 형태로 만들기 위한 BNF가 있어야 한다. (그림 5)는 BNF 코드 일부를 보여주고 있다. BNF 코드는 두 가지로 분류할 수 있다. 하나는 추상클래스인 Expr 클래스로부터 파생되는 표현식이고 다른 하나는 추상클래스인 Stmt 클래스로부터 파생되는 문장이다. 이 두개의 차이점은 Expr은 파생클래스들이 값과 타입을 가지는 반면에 Stmt는 3-주소 형태의 문장을 표현한다. 이때 스택의 내용을 참고하여 스택의 위치와 해당 위치에 들어있는 값을 읽어 들여 처리하게 된다. 앞에서 구한 레이블을 추가한 바이트코드를 바탕으로 레이블별로 문장을 읽어 들인 후 각각의 명령어 별로 처리한다. 읽어 들인 명령이 레이블인지 명령어인지를 확인한 후 명령어인 경우 명령어가 리더인지를 다시 한번 확인하게 된다. 명령어는 다음 레이블이 나올 때까지 계속 읽어 들인 후 저장하게 된다. 만약 다음 레이블이 나올 경우 현재 스택의 높이를 계산하고 마지막에 나온 명령어의 BNF를 이용하여 트리를 구성하게 된다.

(그림 6)은 (그림 4)에서 label_0에 있는 바이트코드

```
Expr → ConstantExpr | DefExpr | StoreExpr
ConstantExpr → ' Id ' | Num + F | Num
              + L | Num
StoreExpr → ( MemExpr := Expr )
DefExpr → MemExpr
MemExpr → MemRefExpr | VarExpr
VarExpr → LocalExpr | StackExpr
LocalExpr → Stack | Local Type Num
Stmt → ExprStmt | InitStmt | JumpStmt |
       LabelStmt
LabelStmt → Label
ExprStmt → eval Expr
InitStmt → INIT LocalExpr( )
JumpStmt → GotoStmt | IfStmt |
           ReturnExprStmt
IfStmt → IfZeroStmt
GotoStmt → goto Block
IfZeroStmt → if 0 ( Expr == | != | > | >= |
   < |<= null | 0 ) then Block else Block ReturnExprStmt →
return Expr
Block → <block Label >
Label → label_Num
```

그림 5. BNF 코드
Fig 5. BNF Code

ldc 1, istore x$2를 트리구조 코드로 표현한 것이다. ldc 1은 스택에 1을 넣게 되고, istore x$2는 스택에 있는 1을 지역 변수 배열 2번째에 넣게 된다. 결국 eval (Locali2_2 := 1) 과 같은 하나의 문장으로 표현 할 수 있다.

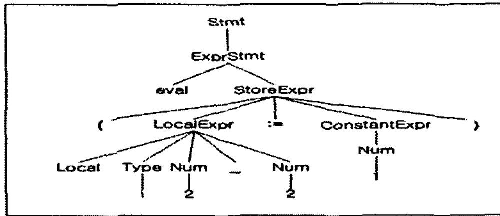

그림 6. eval(Locali2_2:=1) 트리구조
Fig 6. eval(Locali2_2:=1) Tree Structure

분기가 있는 명령어인 경우 기본블록과 제어 흐름 그래프를 만들기 위해서 필요하다. 분기가 있는 명령어일 경우 분기의 목표가 되는 레이블을 저장한다. 그 다음 레이블 간에 간선으로 연결하게 된다. 분기가 일어나는 레이블은 레이블 블록으로 만들어 기본블록을 이루게 된다.

(그림 7) 레이블을 붙인 바이트코드에서 트리구조 3-주소 코드를 표현한 상태의 모습을 나타내고 있다. 변수들의 정의가 존재 하는지를 알기 위해서 version 이라는 변수를 카운트해서 변수의 뒤에 붙이게 된다. 예를 들어 locali2_2 의 경우 2가 지역 변수로 몇 번째로 정의되었는지를 알 수 있게 된다. 하지만 locali2_undef의 경우 undef은 아직 정확히 사용이 되지 않은 것이기에 정의하지 않았다. 정의 되지 않은 부분은 5장에서 Ø-함수를 삽입으로 재정의 하게 된다. 레이블 간에 간선들은 연결한 후에 블록 간에 간선으로 연결하게 되면 제어 흐름 그래프가 만들어 지게 된다. 앞서 말한 것 같이 제어 흐름 그래프는 선행자와 후행자를 간선으로 가지고 있어야 한다. (그림 8)은 (그림 7)의 기본 블록을 제어 흐름 그래프로 나타낸 것이다. (그림 8)에서 3개의 노드가 추가 되어 있다. 이것은 시작노드(entry node), 종료노드(exit node), 초기화 노드(init node)가 추가 된 것이다. 시작노드는 그래프의 시작을 알리기 위해서 사용되고 종료노드는 그래프의 종료를 알리기 위해 사용된다.

<table>
<tr><td>label_0
ldc 1
istore x$2
label_2
iload b$1
ifeq label_11
label_6
ldc 2
istore x$2
goto label_13
label_11
ldc 3
istore x$2
label_13
iload x$2
ireturn
label_15</td><td>⟨block 0⟩
label_0
eval (Locali2_2 := 1)
label_2
If0 (Locali1_undef == 0) then
⟨block 11⟩ else ⟨block 6⟩

⟨block 6⟩
label_6
eval (Locali2_6 := 2)
goto label_13

⟨block 11⟩
label_11
eval (Locali2_4 := 3)
goto label_13

⟨block 13⟩
label_13
return Locali2_undef</td></tr>
<tr><td>label을 추가한 모습</td><td>트리구조 3-주소 코드</td></tr>
</table>

그림7. 트리구조 3-주소 코드
Fig 7. Tree Structure 3-address Code

초기화 노드는 그래프에서 사용되는 정보들을 초기화 시키는데 사용된다. 이 세 가지 노드들은 실제 제어의 흐름과는

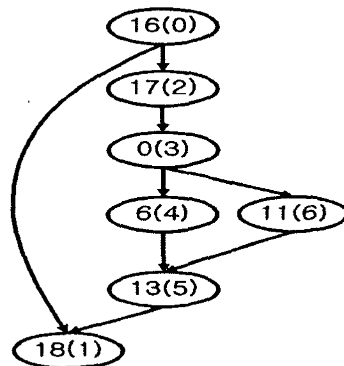

그림 8.제어 흐름 그래프
Fig 8. Control Flow Graph

상관없이 정보만을 가지고 있게 된다. 이 노드들은 다음 작업으로 지배자와 지배자 경계를 구할 경우 필요하게 된다. 제어 흐름 그래프는 지배자 트리와 지배자 경계를 구하기 위해 필요하다.

자기 자신을 다시 합집합 하는 이유는 (그림 9) 지배자의 특성에서 보듯이 모든 노드는 자기 자신을 지배하기 때문이다.

Ⅳ. 바이트코드 분석을 위한 정적 단일 배정문 설계4.1 지배자

지배자란 제어 흐름 그래프에서 초기노드로부터 노드 y까지 도달할 때 모든 경로가 x를 거쳐야 한다면 노드 x는 노드 y을 지배하며 이때 노드 x는 y을 '지배한다'라고 표현하고, 'x dom y'이라[7] 표현한다. (그림 9)는 지배자에 관한 특성을 나타낸 것이다.

- ο 노드 x는 y를 지배한다.(x dom y)
- ο 흐름그래프의 entry로부터 노드 y로 가는 모든 경로가 x를 포함한다.
- ο 모든 노드는 자기 자신을 지배한다.
- ο a가 b를 지배하고 b가 c를 지배하면 a는 c를 지배한다.
- ο a가 b를 지배하고 b가 a를 지배하면 a=b이다.

그림 9. 지배자 정의
Fig 9. Dominator Definition

지배자 트리는 지배자 관계를 트리로 표현한 것이다. 지배자 트리를 구하기 위해서는 우선 그래프에서 깊이 우선 신장 트리(DFST: depth-first spanning tree)를 구한 다음 지배자를 계산하게 된다. 알고리즘 1은 지배자 트리를 계산 하는 알고리즘[7]을 나타낸 것이다. 〈표 2〉는 (그림 8)의 지배자를 계산하기 위해 초기화된 모습과 지배자를 계산한 모습을 나타내고 있다.

〈표 2〉는 집합을 사용하여 지배자를 구한 것이다. 깊이 우선 신장 트리의 블록 탐색 순서로 프로그램에서도 집합 형태로 계산하게 된다. 알고리즘 1에서 초기화라고 되어 있는 부분은 시작(entry)노드를 제외한 각 노드들에 대해서 모든 노드들을 포함한다. 시작 노드를 제외한 모든 노드들에 대해 선행자의 지배자를 구한다. 구한 노드를 교집합 한 다음 자기 자신을 다시 합집합 한다.

$$D(n) := \{n\} \cup \left(\bigcap_{p \in pre(n)} D(p) \right)$$

```
지배자 계산 알고리즘
입력 : set of node
출력 : set of node
방법 :
N : in set of Node
Pred : in Node —> set of Node
r : in Node
begin
D, T : set of Node
n, p : Node
change := true
Domin : Node —> set of Node
Domin(r) := {r}
for each n ∈ N - {n} do
Domin(n) := N        // 초기화
od

repeat
change := false
for each n ∈ N - {n} do
T := N
for each p ∈ pred(n) do
T∩=Domin(p)
od
D := {n} U T
if D≠ Domin(n) then
change := true
Domin(n) := D
fi
od
until !change
return Domin
end
```

알고리즘 1. 지배자 계산 알고리즘
Alg 1. Dominator Compute Algorithm

지배자 계산은 한번에 계산되지 않고 여러 번 반복을 하게 된다. 알고리즘 1에서 보듯이 boolean 값의 change 변수를 true로 설정한 다음 지배자의 변화가 없을 때까지 반복을 함으로써 지배자를 계산하게 된다.

〈표 2〉에서 지배자 계산 부분은 계산된 지배자를 보여주고 있다. 하지만 지배자 계산으로 각 블록간의 지배관

표 2. 지배자 계산
Table 2. Dominator Compute

블록	초기화	지배자 계산
0 (entry)	{0(entry)}	{0(entry)}
1 (exit)	{0(entry),1,2,3,4,5,6}	{0(entry),1(exit) }
2 (17)	{0(entry),1,2,3,4,5,6}	{0(entry),2}
3 (0)	{0(entry),1,2,3,4,5,6}	{0(entry),2,3}
4 (6)	{0(entry),1,2,3,4,5,6}	{0(entry),2,3,4}
5 (13)	{0(entry),1,2,3,4,5,6}	{0(entry),2,3,5}
6 (11)	{0(entry),1,2,3,4,5,6}	{0(entry),2,3,6}

계를 좀더 정확한 블록간의 지배관계를 알기 위해서 직접 지배자를 구한다. 직접 지배자는 초기 노드 X에서 Y까지의 경로에서 노드 Y의 마지막 지배자를 의미한다. 알고리즘 2는 직접 지배자를 계산하는 알고리즘이다.

```
직접 지배자 계산 알고리즘
입력 : FlowGraph(blocks)
출력 : bitset of idom
방법 :
begin
  blocks := graph.nodes.iterator
  repeat
   if blocks.hasNext == true
     block := (Block) blocks.next
     i = graph.preOrderIndex(block)
      if i == root
           block.setDomParent(null)
      else
         blockDoms := dom(i)
            idom := new BitSet(size)
            idom.or(blockDoms)
            idom.clear(i)
             for j := 0, j < size, j++   do
               if i != j && blockDoms.get(j)
               domDomBlocks := dom(j)
               b := new BitSet(size)
                 b.or(domDomBlocks)
                 b.xor(ALL)
                 b.set(j)
                 idom.and(b)
             fi
          od
      fi
   fi
  until ! blocks.hasNext
end
```

알고리즘 2. 직접 지배자 계산 알고리즘
Alg 2. Immediate Dominator Compute Algorithm

알고리즘 2는 우선 root인지를 확인한다. 시작노드인 root는 직접 지배자가 없기 때문이다. blockdoms에 지배자를 넣게 된다. 그 후에 idom과 or연산을 하게 된다. or연산으로 dom에 모든 노드들을 넣게 되고 idom에서 자기자신을 clear시켜줌으로써 초기화를 하게 된다. 초기화 하게 되면 <표 3>에서 직접지배자 초기화 부분과 같게 된다. 자기 자신을 제외한 blockdoms에 노드들에 대해서 계산을 하게 된다. 해당 노드의 dom을 domDomblocks에 넣고 b에 다시 domDomblocks를 넣게 된다. 그 후 b와 ALL을 xor연산을 한 후 idom과 b를 and 연산하여 직접 지배자를 구하게 된다. <표 3>은 알고리즘 2를 이용하여 계산된 것이다.

(그림 10)는 제어 흐름 그래프에서 계산된 지배자 관계를 지배자 트리로 보여 주고 있다. <표 3>에서 노드 2의직접 지배자는 노드 0이 되고, 노드 3의 직접 지배자는 2 가 된다. <표 3>의 노드들을 연결하게 되면 지배자 트리를

표 3. 직접 지배자 계산
Table 3. Immediate Dominator Compute

Node	지배자 Dom(node)	직접지배자 초기화	직접 지배자
0 (entry)	{0(entry)}	∅	∅
1 (exit)	{0(entry),1 (exit)}	{0(entry)}	{0(entry)}
2	{0(entry),2}	{0(entry)}	{0(entry)}
3	{0(entry),2,3}	{0(entry),2}	{2}
4	{0(entry),2,3}	{0(entry),2,3}	{3}
5	{0(entry),2,3,5 }	{0(entry),2,3}	{3}
6	{0(entry),2,3,6 }	{0(entry),2,3}	{3}

구할 수 있게 된다. 따라서 (그림 10)에서 알 수 있듯이 지배자 트리를 구함으로써 블록간의 지배 관계를 알 수 있다.

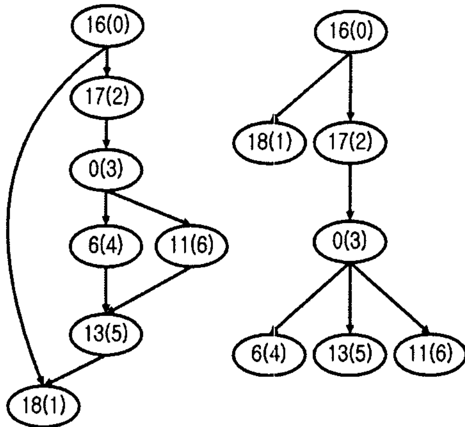

그림 10. 제어 흐름 그래프와 지배자트리
Fig 10. Control Flow Graph and Dominator Tree

4.2 지배자 경계

지배자 경계는 제어 흐름 그래프와 지배자 트리를 이용하여 ∅-함수를 삽입할 위치를 계산하는 것이다. 프로그램 상에서 조건 분기문인 while 문이나 if문과 같은 구조 또는 goto와 같은 무조건 분기문에서는 흐름 그래프에서 병합되는 경로가 존재하게 된다. 병합되는 블록에 ∅-함수를 삽입하게 되는데, 프로그램 에서 분기 또는 반복이 되는 곳에서 병합되는 위치 즉, ∅-함수 삽입하는 위치를 알 수가 없다. 따라서 모든 노드는 자신이 지배하지 않는 노드의 위치를 가지고 있을 필요가 있다. 이러한 ∅-함수를 삽입할 위치를 지배자 경계라고 한다. 즉, 지배자 경계는 제어 흐름 그래프에서 모든 노드 x에 대해서 필요한 ∅-함수를 삽입할 위치를 말한다.

```
지배자 경계 계산 알고리즘
입력 : Block(block), FlowGraph(graph)
출력 : LinkedList of df
방법 :
begin
local := new Block[graph.size()]
Iterator children;
 repeat
  if children.hasNext == true
    child := (Block) children.next
    df := calcFrontier(child, graph)
    e := df.iterator
    repeat
if children.hasNext == true
    child := (Block) children.next
    df := calcFrontier(child, graph)
          local[graph.preOrderIndex(dfChild)] := dfChild
        fi
       succs := graph.succs(block).iterator
      fi
    until ! e.hasNext
  fi
 until ! children.hasNext

repeat
  if succs.hasNext == true
      succ := (Block) succs.next();
      if (block != succ.domParent())
        local[graph.preOrderIndex(succ)] := succ;
      fi
  fi
until ! succs.hasNext

v := new LinkedList();
  for i := 0, i < local.length, i++ do
     if local[i] != null
          v.add(local[i])
     fi
  do
      block.domFrontier().clear();
      block.domFrontier().addAll(v);
  return v;
end
```

알고리즘 3. 지배자 경계 알고리즘
Alg 3. Dominator Frontier Algorithm

지배자 경계를 구하기 위해서 두 가지의 개념을 알아야 하는데 DFlocal과 DFup이다. DFlocal은 노드 n에 의해 직접 지배되지 않는 n의 선행 노드집합을 말하며 DFup은 노드 n에 의해 간접 지배되는 노드들에 의해 지

배되지 않는 지배자 경계의 노드 집합을 말한다. 이 두 개념으로 지배자 경계를 구하는 식은 다음과 같이 간단하게 표현할 수 있다.

$$DF(n) = DFlocal(n) \cup \bigcup_{Z \in children(n)} DFup(z)$$

알고리즘 3은 위의 식을 바탕으로 지배자 경계를 구하는 알고리즘을 나타낸 것이다. <표 4>는 지배자 경계를 구하기 위해 계산한 결과를 나타낸 것이다. <표 4>에서 DF(n)은

병합이 발생하고 Ø-함수를 삽입해야 하는 위치이다. {1, 5}는 〈block_18〉과 〈block_13〉을 나타낸다.

표 4. 지배자 경계 계산
Table 4. Dominator Frontier Compute

Node (block)	Succ(n)	DF(n)	DFup(n)	DFlocal(n)
0 (16)	{ 2 }	{ }	{ }	{ }
1 (18)	{ }	{ }	{ }	{ }
2 (17)	{ 3 }	{ 1 }	{ }	{ 1 }
3 (0)	{ 4 , 6 }	{ 1 }	{ }	{ 1 }
4 (6)	{ 5 }	{ 5 }	{ 5 }	{ }
5 (13)	{ 1 }	{ 1 }	{ 1 }	{ }
6 (11)	{ 5 }	{ 5 }	{ 5 }	{ }

V. 바이트코드 분석을 위한 정적 단일 배정문 구현 및 성능 평가

5.1 Ø-함수 삽입

정적 단일 배정문 형태를 보면 자바 바이트코드는 while문과 for 문이 없이 if와 같은 조건 분기문이나 혹은 goto와 같은 무조건 분기문만으로 이루어져있기 때문에 보다 간단하게 표현할 수 있다. 4장에서 계산한 지배자 경계를 통해 Ø-함수의 삽입할 위치를 계산 했다면 Ø-함수를 삽입해야 한다. 4장에서 계산한 지배자 경계를 다시 반복적으로 계산함으로써 Ø-함수가 추가될 위치를 찾게 된다. 반복적인 지배자 경계를 계산하게 되면 블록 〈block_13〉과 〈block_18〉인 것을 알 수 있다. 하지만 〈block_18〉은 종료(Exit)노드이기 때문에 Ø-함수를 삽입하지 않게 된다. 따라서 프로그램에서 Ø-함수를 삽입하는 부분은 〈block_13〉이 된다. 본 논문에서는 Ø-함수 삽입으로 Phistmt를 삽입하게 된다. Phistmt의 BNF코드는 3장에서 정의했다. 이제 Phistmt를 넣을 부분을 찾아야 한다. 〈block_13〉에서 아직 정의 되어 있지 않은 locali2_undef 부분에서 Phistmt를 설정하게 된다. Phistmt는 우선 locali2_undef := Phi()로 설정하게 된다. 그 다음 locali2_undef가 새로 정의된 변수이기 때문에 locali2_10을 붙이게 된다. Phi()부분에는 병합 노드의 선행자들을 추가하게 된다.

〈block_13〉의 선행자는 〈block_6〉과 〈block_11〉이고 이 두 블록에서 Phi()가 추가 된 모습은 Phi(locali2_undef ,locali2_undef)이 된다.

(그림 11)은 Ø-함수를 삽입한 모습을 나타내고 있다.

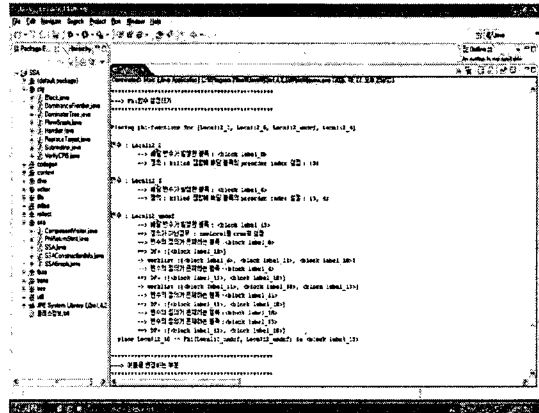

그림 11. Ø-함수 삽입한 모습
Fig 11. Ø-Function insertion

5.2 변수 이름 바꾸기

Ø-함수를 삽입했다면 변수 이름을 바꾸어야 한다. 이름을 바꾸게 되는 변수에 대해 그 변수가 정의되었는지를 확인해야 한다. 우선 변수가 정의될 때 마다 각각의 변수 스택을 두어 변수가 재정의될 때 마다 각 변수의 스택에 저장하게 된다. phi(locali2_undef , locali2_undef)에 있는 locali2_undef에 대해서 locali2가 각 블록에 정의되어 있는지를 확인한다. (그림 13)은 실제 프로그램을 수행 시 나온 결과물을 나태나고 있다.

(그림 10)의 제어 흐름 그래프와 지배자 트리를 이용하여 확인하게 된다. 우선 locali2는 〈block_0〉에서 locali2_2로 정의가 존재한다. 정의가 존재하게 되면 스택에 넣게 된다. 그리고 다음의 블록에서 다시 locali2가 정의되어 있는지를 확인하고 같은 방법으로 스택에 넣게 된다. 따라서 phi(locali2_undef , locali2_undef)에서 locali2_undef는 〈block_6〉에서 온 것으로 해당 변수를 현재 스택에 있는 locali2_6으로 넣게 된다. 같은 방법으로 뒤에 locali2_undef은 〈block_11〉에서 온 것으로 스택에 있는 locali2_4를 넣게 된다. 이렇게 변수들의 이름을 바꾸게 되면 다음과 같이 locali2_10 := phi(locali2_6 , locali2_4)와 같은 형태가 된다. 변수의 이름을 바꾼 후

〈block label_0〉 label_0 eval (Locali2_2 := 1) label_2 if0(Locali1_undef == 0) then 〈block label_11〉 else 〈block label_6〉 〈block label_6〉 label_6 eval (Locali2_6 := 2) goto label_13 〈block 11〉 label_11 eval (Locali2_4 := 3) goto label_13 〈block 13〉 label_13 return Locali2_undef	〈block label_0〉 label_0 eval (Locali2_2 := 1) label_2 if0 (Locali1_1 == 0) then 〈block label_11〉 else 〈block label_6〉 〈block label_6〉 label_6 eval (Locali2_6 := 2) goto label_13 〈block label_11〉 label_11 eval (Locali2_4 := 3) goto label_13 〈block label_13〉 label_13 Locali2_10:= Phi(label_6=Locali2_6, label_11=Locali2_4) return Locali2_10
3-주소 코드로 변환한 모습	완성된 정적 단일 배정문 형태

그림 12. 변환된 정적 단일 배정문 형태
Fig 12. Translated Static Single Assignment Form

해당 블록에 Phistmt를 추가 하면 된다. (그림 12)는 Phistmt를 추가한 모습으로 완성된 정적 단일 배정문 형태를 나태나고 있다.

5.3 성능 평가

성능 평가는 Pentium 4 1.8GHz, 메모리 512MB, 자바 컴파일러는 j2sdk 1.4.2_09에서 성능 평가를 했으며

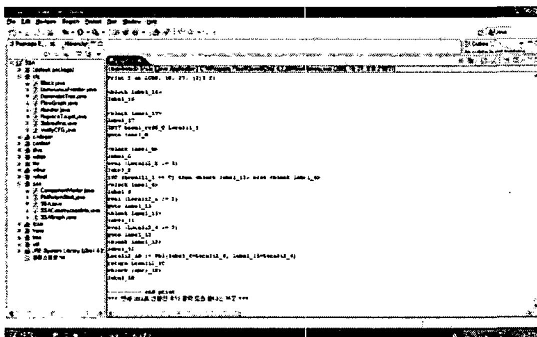

그림 13. 완성된 정적 단일 배정문 형태
Fig 13. Finished Static Single Assignment Form

eclipse 3.0을 사용하였다.. 예제 프로그램으로는 BubbleSort, SumOfSquareRoot, Fibonacci, Nestedloop, Ackermann 함수를[8] 사용하였다. 평가는 먼저 바이트코드의 라인 수, 제어 흐름 그래프의 라인 수, 제어 흐름 그래프의 노드 수, 정적 단일 배정문 형태 라인 수, 정적단일 배정문 형태 노드 수를 비교 하게 된다. 바이트코드의 라인 수와 제어 흐름 그래프의 라인 수는 3-주소 코드로 변환했을 경우 줄어든 라인 수를 비교할 수 있다. 또한 제어 흐름 그래프와 정적 단일 배정문의 노드와 라인 수의 비교는 삽입한 ∅-함수로 인해 발생되는 노드와 라인 수를 비교할 수 있다. 〈표 5〉는 실험에 대한 결과를 나타나고 있다.

표 5. 실험 결과
Table 5. Experimentation Result

프로그램	bytecode line	cfg line	cfg node	SSA line	SSA node
BubbleSort	79	56	117	64	131
SumOfSquareRoot	103	77	129	82	147
Fibonacci	76	54	108	69	116
Ackermann	48	26	53	26	53
Nestedloop	109	87	137	106	197

실험 결과로 알 수 있듯이 정적 단일 배정문 형태를 만들면서 노드의 개수와 라인수가 증가하게 된 걸 알 수 있다. 하지만 Ackermann 함수 프로그램에서는 증가분이 발생하지 않은 것을 알 수 있다. 그 이유는 Ackermann 함수 프로그램에서는 분기와 관련된 명령어들은 있지만 병합되는 노드들이 없기 때문에 ∅-함수 삽입이 일어나지 않았기 때문이다.

Ⅵ. 결론 및 향후 과제

제어 흐름 그래프를 바탕으로 바이트코드 분석을 위해 정적 단일 배정문 형태로 변환을 하게 된다. 정적 단일 배정문을 만들기 위해서는 지배자 경계 계산이 필요하다. 지배자 경계 계산은 제어 흐름 그래프와 지배자를 이용하여

계산하게 된다. 지배자는 블록간의 지배관계를 계산 구조를 알기 위한 것으로 제어 흐름 그래프를 이용하여 계산하게 된다. 계산된 지배자 경계를 이용하여 ∅-함수를 블록에 삽입하였고, 변수들의 이름을 바꾸므로 정적 단일 배정문 형태를 만들었다. 정적 단일 배정문은 정의-사용 관계에서 개선된 형태로써 프로그램에서 각 변수들이 서로 다른 값과 다른 타입을 가질 수 있기 때문에 변수는 값이나 타입에 따라 분리하는 것이다. 따라서 각 변수의 값과 타입들을 오직 한번만 배정하고 재명명함으로써 프로그램을 정적으로 분석할 수 있게 했다.

향후 연구로 본 논문에서 연구한 정적 단일 배정문을 이용하여 프로그램의 분석뿐만 아니라 죽은 코드 제거, 상수 전파, 타입추론, 부분 중복 제거등과 같은 최적화 기법을 적용할 수 있다.

참고문헌

[1] Don Lance, Roland H. Untch, Nancy J. Wahl, "Bytecode -based Java program analysis", Proceedings of the 37th annual Southeast regional conference, p 104-110, 1999.

[2] Carl McConnell, Ralph E. Johnson, "Using static single assignment form in a code optimizer", ACM Letters on Programming Languages and Systems, Vol.1(2), p 152-160, 1992.

[3] Stenve S. Muchinck, "Advanced Compiler Design and Implementation", Mogan Kaufmann Publishing, 1997.

[4] Ken Arnold, James Gosling, "The JavaProgramming Language", Addison-Wesley Publishing, 1996.

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

[6] Tim Lindholm, Frank Tellin, "The Java Virtual Machine Specification", Addison-Wesley Publishing, 1997.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", ACM Transactions on Programming Languages and Systems, Vol.13(4), p 451-490, 1991.

[8] benchmark http://dada.perl.it/shootout/craps.html

[9] 김영선, 최우승, "Java 코드를 이용한 UML 클래스 다이어그램 생성 도구의 설계", 한국 컴퓨터 정보학회 논문지, 제 5호 1권, 2000.

[10] 최준기, "최적화 컴파일러의 성능향상 기법에 관한 연구", 한국 컴퓨터 정보학회 논문지, 제 5호 1권, 2000.

저 자 소 개

김 경 수

2004년 2월 관동대학교 컴퓨터공학학사

2004년 ~ 2006 인하대학교 컴퓨터공학과 석사

〈관심분야〉 컴파일러, 바이트코드, 최적화

유 원 희

1975년 서울대학교 공과대학 응용수학과

1978년 서울대학교 대학원 계산학 전공(이학석사)

1985년 서울대학교 대학원 계산학 전공(애학박사)

1992년 ~ 1993년 University of California, Irvine 객원연구원

1979 ~ 현재 : 인하대학교 전자계산공학과 교수

〈관심분야〉 프로그래밍 언어, 컴파일러, 실시간 시스템, 병렬시스템