CTOC에서 자바 바이트코드를 위한 정적 단일 배정 형태

김 기 태[†] · 유 원 희[††]

요 약

자바 바이트코드는 많은 장점을 갖지만 수행 속도가 느리고 분석이 어렵다는 단점을 갖는다. 이를 극복하기 위해 바이트코드에 대한 분석과 최적화가 수행되어야 한다. 최적화된 코드를 위해 CTOC를 구현하였다.

바이트코드에 대해 분석과 최적화를 수행하기 위해서는 우선 CFG를 생성해야 한다. 바이트코드의 특성 때문에 기존의 제어 흐름 분석 기술을 바이트코드에 적합하게 확장해야 한다. 또한 정적으로 분석하기 위해 CFG를 SSA Form으로 변환한다. SSA Form으로 변환하기 위해서는 지배 관계, 지배자 트리, 직접 지배자, Ø-함수, 재명명, 지배자 경계 등 많은 정보에 대한 계산을 수행한다.

본 논문은 기존의 CFG로부터 SSA Form으로 변환을 위해 알고리즘과 변환 과정을 기술한다. SSA Form이 적용된 그래프는 추후에 타입 추론과 최적화를 위해 사용된다.

키워드 : CTOC, 자바 바이트코드, 제어 흐름 그래프, 정적 단일 배정 형태

Static Single Assignment Form for Java Bytecodes in CTOC

Ki-Tae Kim[†] · Weon-Hee Yoo[††]

ABSTRACT

Although the Java bytecode has numerous advantages, there are also shortcomings such as slow execution speed and difficulty in analysis. In order to overcome such disadvantages, bytecode analysis and optimization must be performed. We implements CTOC for optimized codes.

An extended CFG must be first created in order to analyze and optimize a bytecode. Due to unique bytecode properties, the existing CFG must be expanded according to the bytecode. Furthermore, the CFG must be converted into SSA Form for a static analysis, for which calculation is required for various information such as the dominate relation, dominator tree, immediate dominator, Ø-function, rename, and dominance frontier.

This paper describes the algorithm and the process for converting the 'existing CFG into the SSA Form. The graph that incorporates the SSA Form is later used for type inference and optimization.

Key Words : CTOC, Java Bytecodes, CFG(Control Flow Graph), SSA(Static Single Assignment) Form

1. 서 론

바이트코드는 자바 가상 기계(JVM)라 불리는 인터프리터에 동적으로 적재되고 수행된다[1]. 이러한 바이트코드는 유용한 특징을 많이 갖지만 스택 기반 코드이기 때문에 수행 속도가 느리고 분석이나 최적화에 적절한 표현은 아니다[2]. 또한 동적 링크 지원을 위해 고안된 상수 풀의 크기가 상대적으로 크다는 문제점을 갖는다. 따라서 클래스 파일이 네트워크와 같은 실행환경에서 효과적으로 실행되기 위해서는 코드 최적화가 요구된다.

※ 이 논문은 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(D00620)
† 정 회 원 : 인하대학교 컴퓨터공학부 강의전임강사
†† 종신회원 : 인하대학교 컴퓨터공학부 교수
논문접수 : 2005년 11월 4일, 심사완료 : 2006년 10월 12일

기존의 자바 또는 자바 바이트코드의 제어 흐름 분석과 최적화에 관련된 연구로는 Purdue 대학에서 수행한 Bloat 프로젝트나, Middle Tennessee 대학의 JAristotle 프로젝트, 그리고 McGill 대학에서 수행한 SOOT 프로젝트 등이 있다[3, 4, 5]. Bloat와 JAristotle 프로젝트에서는 로딩 동작을 효율적으로 할 수 있는 방법을 제시하고 바이트코드 내에서 사용되는 메소드의 분석을 용이하게 하는 방법을 제시한다. 하지만 이 프레임워크들은 바이트코드 형태에서 조작을 수행하는 특징을 가진다. SOOT 프로젝트는 3-주소 중간 표현을 이용해서 중요한 최적화를 실행한다. 하지만 바이트코드를 네이티브 코드로 생성하기 때문에 해당 기계에 의존적인 최적화 코드가 생성되는 특징을 가진다.

본 논문에서 제시하는 CTOC는 Bloat의 효율적인 로딩 동작과 SOOT의 3-주소 중간 표현을 사용한다. 하지만

CTOC에서는 바이트코드에 정보를 추가하여 분석을 용이하게 하고 트리 형태의 3-주소 중간 표현을 사용한다. 또한 CTOC는 최적화된 코드를 다시 바이트코드로 변환하기 때문에 플랫폼에 독립적인 특징은 계속 유지된다.

바이트코드의 분석과 최적화를 위해 가장 먼저 수행되는 것은 제어 흐름 분석이다. 또한 데이터 흐름 분석과 최적화를 위해서는 변수가 어디서 정의되고 어디서 사용되는지에 대해서 알아야 한다. 이러한 정보를 정의(def)와 사용(use)이라고 하는데 전통적인 컴파일러에서는 정의-사용 고리로 정의와 사용에 대한 정보를 유지한다[6, 7]. 변수를 정적으로 다루기 위해서는 변수들을 정의와 사용에 따라 분리해야 한다. 왜냐하면 동일한 변수라도 정의와 사용에 따라 다른 위치에서 다른 값과 다른 타입을 가질 수 있기 때문이다. 다라서 정적으로 값과 타입을 결정하기 위해서는 변수를 배정되는 것에 따라 분리해야 한다. 이를 위해 본 논문에서는 정의-사용 고리 대신 SSA Form을 사용한다[8]. SSA Form은 최근 데이터 흐름 분석이나 코드 최적화를 위해 컴파일러의 중간 표현으로 많이 사용된다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로 현재 개발 중에 있는 CTOC와 기본 블록 및 CFG에 대해 간략히 표현한다. 3장에서는 지배자 트리 생성, 지배자 경계 구하기, ∅-함수 추가, 변수 재명명, 반복적인 지배자 구하기와 같은 SSA Form으로 변환하는 단계에 대해 기술한다. 4장에서는 일반적으로 사용되는 예제 프로그램을 이용해서 실험을 수행한다. 5장에서는 결론과 향후 계획을 제시한다.

2. 관련 연구

2.1 CTOC

CTOC는 현재 개발 중인 최적화 프레임워크이다[9, 10]. CTOC는 McGill 대학의 SOOT 프로젝트와 Purdue 대학의 Bloat 프로젝트에 기반을 둔다[3, 5].

〈표 1〉 리더로 사용 가능한 바이트코드 명령어

명령어 종류	바이트코드 명령어
조건 분기 명령어	ifeq, ifne, iflt, ifge, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, ifnull, ifnonnull, if_acmpeq, if_acmpne
비교 명령어	lcmp, fcmpl, fcmpg, dcmpl, dcmpg
무조건 분기 명령어	goto, goto_w
서브루틴 명령어	jsr, jsr_w, ret
예외처리 명령어	athrow
테이블 점프 명령어	tableswitch, lookupswitch
메소드 반환 명령어	ireturn, lreturn, freturn, dreturn, areturn, return

2.2 기본 블록

기본 블록은 내부에 제어의 이동이나 분기가 존재하지 않고 블록 내부로의 제어 이동도 존재하지 않는 연속적인 코드의 집합이다. 전통적인 코드에서는 기본 블록의 첫 번째 문장을 리더(leader)라고 하며, 이 리더들의 집합을 결정한다[6]. 리더로 사용 가능한 바이트코드 명령어는 <표 1>과 같이 정리된다.

본 논문에서는 <표 1>에서 서술된 리더 관련 명령어와 바이트코드의 코드(CODE) 속성에 존재하는 라인 번호 테이블(LineNumberTable)을 이용하여 기본 블록을 생성한다.

2.3 제어 흐름 그래프

CTOC에서 제어 흐름 그래프를 생성하기 위해 (그림 1)(a)와 같은 예제 프로그램을 사용한다.

```
1:  public class Temp {
2:    int f(boolean b){
3:      int x;
4:        x = 1;
5:        if(b)
6:          x = 2;
7:        else
8:          x = 3;
9:        return x;
10:   }
11: }
```

```
public class Temp extends java. lang.Object{
public Temp();
  Code:
   0:  aload_0
   1:  invokespecial  #9;
   4:  return

int f(boolean);
  Code:
   0:  iconst_1
   1:  istore_2
   2:  iload_1
   3:  ifeq 11
   6:  iconst_2
   7:  istore_2
   8:  goto 13
   11: iconst_3
   12: istore_2
   13: iload_2
   14: ireturn
}
```

(그림 1) (a)예제 프로그램 (b)바이트코드

```
<block_16>
   label_16

<block_17>
   label_17
      INIT Local_ref0_0 Locali1_1
      goto label_0

<block_0>
   label_0
      eval (Locali2_2 := 1)
   label_2
      if0 (Locali1_UDef == 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_UDef

<block_18>
   label_18
```

(그림 2) 확장된 CFG

(그림 1) (a)의 소스에 javap -c 옵션을 적용한 결과는 (그림 1)(b)와 같다. (그림 1)(b)는 단순히 바이트코드이기 때문에 프로그램 분석과 최적화에 적절한 표현은 아니다. 따라서 확장된 새로운 CFG를 작성해야 한다. 새롭게 작성된 CFG는 SSA Form을 구성하는데 입력으로 사용된다. 확장된 CFG는 (그림 2)와 같다[9,10].

(그림 2)에서 <block label_0>은 기본 블록을 나타내고, 기본 블록 내부의 코드는 트리 형태로 구성되고, 3-주소 형태로 변경된 문장으로 표현된다. 즉, (그림 1) (b)의 iconst_1, istore_2의 두 명령어가 (그림 2)에서 eval (Locali2_2 := 1) 형태로 표현된다.

3. SSA 구현

동일한 변수라도 정의와 사용에 따라 다른 위치에서 다른 값이나 다른 타입을 가질 수 있기 때문에 정적으로 값과 타입을 결정하기 위해서는 변수는 배정되는 것에 따라 분리되어야한다. 이를 위해 본 논문에서는 SSA Form을 구현한다. SSA Form의 구현은 <표 2>와 같이 몇 개의 단계로 이루어진다.

〈표 2〉 SSA Form 변환 단계

단계	내 용
1단계	제어 흐름 그래프에서 각 변수에 대한 정보 획득과 지배자 트리 구하기
2단계	지배자 경계(DF) 구하기
3단계	∅-함수 추가하기
4단계	변수 재명명하기
5단계	반복적인 지배자 경계(DF+)가 위치하는 블록에 ∅-문장 추가하기

3.1 변수 정보 획득과 지배자 트리 구하기

SSA Form으로 변환하기 위해서는 (그림 2)의 CFG를 추적하면서 프로그램 내에서 정의 되었거나 사용되는 변수에 대한 정보를 획득해야 한다. CFG에 존재하는 트리의 방문을 위해 TreeVisitor 클래스를 생성하여 사용한다. 이 클래스는 블록별로 해당 문장에 대해 생성된 트리를 깊이 우선 방식으로 방문하면서 각 블록에 정의되거나 사용되는 변수를 찾는 일을 수행한다.

각 블록을 살펴보면 해당 블록에서 정의된 변수에 대한 정보를 찾을 수 있다. 변수에 대한 정보를 찾기 위해서는 해당 노드가 변수를 다루는 VarExpr 클래스로부터 파생된 문장이어야 한다. 변수에 대한 정보는 블록에 변수가 존재하면 해당 변수를 같은 이름을 가진 변수들과 함께 유지해야한다. 이를 위해 var와 allvars라는 연결 리스트를 사용한다. (그림 2)를 보면 <block_0>과 <block_13>에 아직 정의되지 않은 변수가 나타난다. 이들 변수 뒤에는 _UDef라는 표시가 나타난다. 이것은 아직 명확한 이름이 없다는 것을 표시한다. 이들에 대한 새로운 이름은 정적 단일 배정 변환이 수행되는 동안 이루어진다. 모든 블록을 방문한 후 locali2와 관련된 변수에 대한 allvars는 [locali2_2, locali2_4, locali2_UDef, locali2_6]과 같이 생성된다. locali2의 _2, _4와 _6은 변수를 정의할 때마다 다른 이름이 배정된다는 것을 나타낸다. 반면에 _UDef는 변수에 대한 정의 없이 변수가 사용된 경우를 나타낸다.

첫 번째 dom[0]과 블록의 blockDoms는 루트를 의미하는 {0}을 가진다. <표 3>은 생성된 dom[i]를 보인다.

변수에 대한 정보를 획득 한 후 ∅-함수를 삽입하기 위해서는 지배자 경계를 계산해야 한다. 지배자 경계를 구하기 위해서는 먼저 지배자 트리를 생성 한다. 지배자 트리는 지배자의 정보를 쉽게 표현하는 구조이다. 초기 노드는 루

〈알고리즘 1〉 dom[i]를 구하는 알고리즘

```
Input : graph ∈ FlowGraph
Output : dom[i] ∈ Set of dominator
procedure buildDomTree(FlowGraph graph)
begin
    size ← graph.size()
    dom[] ← new BitSet[size]
    all ← new BitSet(size)
    for i ← 0 to i < size do
      all.set(i)
    endfor
    for i ← 0 to i < size do
      blockDoms ← new BitSet(size)
      dom[i] ← blockDoms
      if i != root
           blockDoms ∪ ALL
      else
           blockDoms.set(root)
      fi
    endfor
    boolean changed ← true
    while changed do
      changed ← false
      Iterator blocks ← graph.preOrder().iterator()
      while  blocks.hasNext() do
           block ← blocks.next()
           i ← graph.preOrderIndex(block)
           if (i == root)
              print("ROOT")
              continue
           fi
           oldSet ← dom[i]
           blockDoms ← new BitSet(size)
           blockDoms ∪ oldSet
           Collection preds ← graph.preds(block)
           Iterator e ← preds.iterator()
           while e.hasNext() do
             pred ← e.next()
             j ← graph.preOrderIndex(pred)
             blockDoms ∩ dom[j]
           endwhile
           preds ← (Collection) snkPreds.get(block)
           if preds != null
             e ← preds.iterator()
             while e.hasNext() do
               pred ← e.next()
               j = graph.preOrderIndex(pred)
               blockDoms ∩ dom[j]
             endwhile
           fi
           blockDoms.set(i)
           if ! blockDoms.equals(oldSet)
             changed ← true
             dom[i] ← blockDoms
           fi
      endwhile
    endwhile
end
```

트이고 각 노드는 그 자손들을 지배하는 구조를 가진다. 흐름 그래프에서 초기 노드로부터 노드 n까지 도달할 때 모든 경로가 노드 d를 거쳐야 한다면 노드 d는 n을 지배한다고 한다[6]. 이런 지배 관계를 표현하기 위해 각 블록의 지배 정보를 갖는 dom[i]와 blockDoms를 이용하여 각 블록의 지배자를 구한다. 더 이상 dom[i]에 변화가 없을 때까지 계속 수행하여 각 dom[i]의 값을 구한다. <알고리즘 1>은 Aho가 제시한 지배자 계산 알고리즘을 변형한 것이다[6].

<표 3>에서 <알고리즘 1>을 1회 수행한 후에 변환된 내용을 굵게 표현하였다. 이 경우 dom[i]에 변화가 발생하였기 때문에 계속 알고리즘이 수행된다. 3회 수행 후에는 더 이상 아무런 변화가 없기 때문에 각 블록에 대해 dom[i]의 집합을 구하였다. dom[i]는 지배자에 대한 정보이기 때문에 자기 자신을 포함한다. 지배 경계를 구하기 위해서는 블록간의 지배 관계가 먼저 계산되어야 한다. 우선 현재 블록에 대한 인덱스를 구한 후, 현재 블록의 직접 지배자 찾기를 수행한다. 직접 지배자란 초기 노드에서 n까지 경로에서 노드 n의 마지막 지배자를 의미한다. <알고리즘 2>는 직접 지배자인 idom을 구한다.

〈표 3〉 dom[i]를 구하는 과정

블록	초기화 후	1회 수행 후	2회 수행 후	3회 수행 후
0(16)	{0}	{0}	{0}	{0}
1(17)	{0,1,2,3,4,5,6}	{0,1}	{0,1}	{0,1}
2(0)	{0,1,2,3,4,5,6}	{0,1,2}	{0,1,2}	{0,1,2}
3(6)	{0,1,2,3,4,5,6}	{0,1,2,3}	{0,1,2,3}	{0,1,2,3}
4(13)	{0,1,2,3,4,5,6}	{0,1,2,3,4}	{0,1,2,4}	{0,1,2,4}
5(18)	{0,1,2,3,4,5,6}	{0,5}	{0,5}	{0,5}
6(11)	{0,1,2,3,4,5,6}	{0,1,2,6}	{0,1,2,6}	{0,1,2,6}

〈알고리즘 2〉 직접 지배자 idom을 구하는 알고리즘

```
Input : blocks ∈ FlowGraph
Output : idom
procedure findIdom(blocks)
begin
    while blocks.hasNext() do
        block = blocks.next()
        i ← graph.preOrderIndex(block)
        if i == root
            block.setDomParent(null)
        else
            blockDoms ← dom[i]
            idom ← new BitSet(size)
            idom ∪ blockDoms
            idom.clear(i)
            for j ← 0 to j < size do
                if i != j && blockDoms.get(j)
                    domDomBlocks ← dom[j]
                    b ← new BitSet(size)
                    b ∪ domDomBlocks
                    b ⊗ (ALL)
                    b.set(j)
                    idom ∩ b
                fi
            endfor
        fi
    endwhile
end
```

<알고리즘 2>의 경우, 그래프의 노드들에 해당하는 블록을 찾아 현재 블록의 전위 순서를 변수 i에 지정한다. 현재 i가 루트인가를 확인한 후 루트가 아닌 경우라면 직접 지배자를 찾는다. 직접 지배자를 계산하는 식은 (1)과 같이 현재 블록의 지배자에서 현재 블록의 지배자를 지배하는 것을 제거한 후 현재 블록을 제거한 결과와 같다.

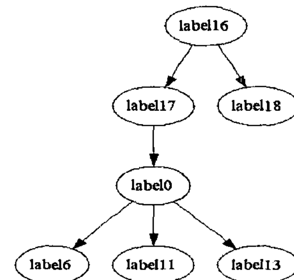
$$idom := dom(block) - dom(dom(block)) - block \quad \cdots(1)$$

<알고리즘 2>를 적용하여 결정된 직접 지배자는 <표 4>와 같다.

〈표 4〉 직접 지배자

블록(전위 순서)	16(0)	17(1)	0(2)	6(3)	13(4)	18(5)	11(6)
직접 지배자	∅	0	1	2	2	0	2

<block_16>은 루트이기 때문에 직접 지배자가 존재하지 않는다. 따라서 ∅로 표시된다. 구해진 idom을 현재 블록의 부모로 설정하여 지배 관계에 대한 정보를 저장한다. (그림 3)은 <표 4>의 정보를 이용해 생성한 지배자 트리이다.



(그림 3) 지배자 트리

3.2 지배자 경계 구하기

제어 흐름 그래프와 지배자 트리를 생성한 후 ∅-함수 삽입을 위해 지배자 경계를 구한다. while 문이나 if 문과 같은 구조에서 문장은 흐름 그래프에서 병합(join)되는 경로가 존재하며 그 위치에 ∅-함수가 삽입된다. 일반적으로 루프가 시작되는 곳에서 ∅-함수가 삽입되는 위치를 알 수 있는 방법은 없다. 따라서 모든 노드는 자신이 지배하지 않는 노드의 위치를 갖고 있을 필요가 발생한다. 이러한 ∅-함수를 삽입할 위치를 지배자 경계(dominance frontier)라고 한다. <알고리즘 3>은 지배자 경계를 구하는 알고리즘을 나타낸다.

<알고리즘 3>에서 dominanceFrontier(child, graph)는 트리의 잎 노드에서 df가 구해질 때까지 계속 재귀적으로 호출되어 진다. 지배자 경계를 DF[n]으로 표현할 때 (2)와 같은 식으로 표현된다.

$$DF[n] = DFlocal[n] \cup \bigcup_{c \in children[n]} DFup[c] \quad \cdots(2)$$

〈알고리즘 3〉 지배자 경계를 구하는 알고리즘

```
Input : block ∈ Block, graph ∈ FlowGraph
Output : df ∈ LinkedList
procedure dominanceFrontier(Block block, FlowGraph graph)
    local[] ← new Block[graph.size()]
    children ← block.domChildren().iterator()
(i) while children.hasNext() do
        child ← children.next()
        df ← dominanceFrontier(child, graph)
        e ← df.iterator()
        while e.hasNext() do
            dfChild ← e.next()
            if  block != dfChild.domParent()
             local[graph.preOrderIndex(dfChild)] ← dfChild
            fi
        endwhile
    endwhile
    succs ← graph.succs(block).iterator()
(ii) while succs.hasNext() do
        succ ← succs.next()
        if  block != succ.domParent()
            local[graph.preOrderIndex(succ)] ← succ
        fi
    endwhile
    v ← new LinkedList()
    for  i = 0 to i < local.length do
        if local[i] != null
            v.add(local[i])
        fi
    endfor
    block.domFrontier().clear()
    block.domFrontier().addAll(v)
    return v
end
```

〈표 5〉 지배자 경계

블록(Block)	선행자(succ)	직접지배자(idom)	지배자 경계(DF)
16	17, 18	∅	∅
17	0	16	18
0	6, 11	17	18
6	13	0	13
13	18	0	18
18	∅	16	∅
11	13	0	13

(2) 식에서 DFlocal[n]은 노드 n에 의해서 직접 지배되지 않는 n의 앞 노드 집합을 의미하고 <알고리즘 3>의 (i)부분이 DFlocal[n]를 표현한다. DFup[c]는 노드 n에 의해 간접 지배되는 노드들에 의해 지배되지 않는 지배자 경계의 노드 집합을 의미하고 <알고리즘 3>의 (ii)부분이 DFup[c]를 표현한다.

<표 5>에서 보듯이 <block_13>과 <block_18>이 지배자 경계로 계산되었다. <block_18>의 경우엔 종료 노드에 해당하기 때문에 ∅-함수를 삽입하지 않는다. 따라서 ∅-함수가 삽입 될 수 있는 위치는 <block_13>이 된다. 제어 흐름 그래프에서 보듯이 <block_13>은 병합되는 노드가 위치한 곳이기도 하다.

3.3 ∅-함수 추가하기

지배자 경계를 통해 ∅-함수가 삽입될 위치가 결정되면 ∅-함수를 삽입해야 한다. 구조적인 언어에서는 ∅-함수가 삽입될 수 있는 위치가 여러 곳에 발생할 수 있다. 왜냐하면 구조적인 언어에서는 특정 위치에 병합 노드가 발생하기 때문이다. 병합이 이루어지는 곳에 일반적으로 ∅-함수가 삽입된다. 바이트코드 경우엔 대부분의 분기가 조건 분기인 if와 무조건 분기인 goto에 의해 이루어지기 때문에 간단하게 표현될 수 있다.

본 논문에서는 ∅-함수를 PhiStmt로 표현한다. ∅-함수가 삽입될 위치를 찾고 그곳에 PhiStmt를 추가 한다. ∅-함수 삽입과 관련된 알고리즘은 <알고리즘 4>와 같다.

<알고리즘 4>에서 정의된 변수들의 집합(killed)과 비 지역변수(non_local)는 변수에 대한 정보를 나타낸다. killed에는 <block_0>의 Locali2_2처럼, 블록에 해당 변수에 대한 정의가 존재하는 경우 killed 집합에 해당 블록의 전위 인덱스를 취

〈알고리즘 4〉 ∅-함수를 삽입하는 알고리즘

```
Input : cfg ∈ FlowGraph, info ∈ SSAInfo
Output : df ∈ LinkedList
procedure PhiFunctions(FlowGraph cfg, SSAInfo info)
begin
    killed ← new BitSet(cfg.size())
    nonLocal ← false
    reals ← info.reals().iterator()
    while reals.hasNext() do
        real ← reals.next()
        block ← real.block()
        if real.isDef()
            killed.set(cfg.preOrderIndex(block))
        else if  ! killed.get(cfg.preOrderIndex(block))
            nonLocal ← true
            break
        fi
    endwhile
    if  ! nonLocal
        return
    fi
    df ← iteratedDomFrontier(info.defBlocks())
    while df.hasNext() do
        block ← df.next()
        if  block != cfg.sink()
            info.addPhi(block)
        fi
    endwhile
end
```

〈알고리즘 5〉 반복적으로 지배자 경계를 구하는 알고리즘

```
Input : blocks ∈ Collection
Output : idf ∈ HashSet
procedure idf(Collection blocks)
begin
    idf ← new HashSet()
    inWorklist ← new HashSet(blocks)
    worklist ← new LinkedList(inWorklist)
    while ! worklist.isEmpty() do
        block ← worklist.removeFirst()
        df ← block.domFrontier()
        iter ← df.iterator()
        while iter.hasNext() do
            dfBlock ← iter.next()
            idf.add(dfBlock)
            if  inWorklist.add(dfBlock)
                worklist.add(dfBlock)
            fi
        endwhile
    endwhile
    return idf
end
```

하고 <block_13>처럼 Local_i2_ UDef가 존재하는 경우엔, 현재 블록에 해당 변수에 대한 정의가 존재하지 않고 다른 블록에 정의가 존재하는 경우이기 때문에 non_local 변수를 true로 설정하여 다른 블록에 존재하는 정보를 이용하여 해당 변수를 정의한다. 이를 위해 이전에 계산된 지배자 경계 값을 이용해서 반복적인 지배자 경계(iterated dominance frontier)를 다시 구한다. 반복적으로 구해진 지배자 경계가 바로 Ø-함수가 추가될 위치를 나타낸다. <알고리즘 5>는 반복적으로 지배자 경계를 구하는 알고리즘이다.

알고리즘 적용 후 반환되는 반복적인 지배자 경계는 <block_13>과 <block_18>이 존재하는데 <block_18>은 종료 노드이기 때문에 Ø-함수가 삽입될 수 없다. 따라서 <block_13>이 Ø-함수가 삽입될 위치가 된다. Ø-함수가 삽입될 위치를 찾을 후에는 병합되는 문장을 의미하는 PhiJoinStmt 문장을 생성해야 한다. 우선 목표가 되는 변수를 확인한다. (그림 2)의 경우엔 Locali2_UDef가 된다. 다음으로는 PhiJoinStmt를 생성한다. Locali2_10 := Phi()의 형태로 나타난다. Locali2_10은 새롭게 이름이 명명된 변수이다. 아직 Phi()안에는 피연산자들이 존재하지 않는데 이것은 해당 블록의 선행자를 통해서 생성하게 된다. 첫 번째 선행자는 <block_11>인데 추가된 후의 모습은 Locali2_10 := Phi(Locali2_UDef)이다. 두 번째 선행자는 <local_6>인데 추가된 후의 모습은 Locali2_10 := Phi(Locali2_UDef, Locali2_UDef)가 된다. 즉, 합병되는 <block_13> 위치에 Locali2_10 := Phi(Locali2_UDef, Locali2_UDef)으로 PhiJoinStmt가 추가된다.

3.4 이름 바꾸기

생성된 PhiJoinStmt에 존재하는 2개의 Locali2_UDef 변수에 대해 재명명을 수행한다. 우선 각 블록에서 해당 변수 Locali2에 대한 정의가 각 블록에 존재하는가를 확인한다. 이때 제어 흐름 그래프와 지배자 트리를 이용한다. (그림 2)의 <block_0>의 경우 Locali2_2에 대한 정의가 존재한다. 존재하는 변수에 대한 정보를 스택의 꼭대기를 의미하는 필드인 top에 설정한다. 이 블록의 후행자에 해당하는 블록에서 다시 해당 변수에 대한 정의를 찾는다. 이때 변수에 대한 정의가 존재하면 다시 스택의 꼭대기에 새로운 변수에 대한 정보를 넣게 된다. <block_11>의 경우 스택의 꼭대기에 새로운 값인 Locali2_4 값이 존재하게 된다. <block_13>의 경우 PhiJoinStmt중 하나의 피연산자인 Locali2_UDef가 <block_11>로부터 온 경우인데, 해당 변수를 현재 스택에 존재하는 Locali2_4 값으로 대체해서 이름을 바꾼다. 여기까지 진행한 후 결과는 Locali2_10 := Phi(Locali2_4, Locali2_UDef)이다. 모든 블록에 대해 변수에 대한 재명명을 수행한 후 결과는 Locali2_10 := Phi(Locali2_4, Locali2_6)이 된다.

3.5 Ø-문장 추가하기

PhiJoinStmt문장에 모든 변수들에 대해 이름을 정한 뒤, PhiJoinStmt를 삽입해야 하는 블록에 해당 문장을 추가한다.

```
<block_16>
  label_16
<block_17>
  label_17
     INIT Local_ref0_0 Locali1_1
     goto label_0
<block_0>
  label_0
     eval (Locali2_2 := 1)
  label_2
     if0 (Locali1_1 == 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
     Locali2_10 := Phi(Locali2_4, Locali2_6)
     return Locali2_10
<block_18>
  label_18
```

(그림 4) 완성된 정적 단일 배정 형태

현재 모든 문장은 연결 리스트 형태로 존재한다. 따라서 (그림 2)에서 라벨 문장 즉, label_13 다음에 생성된 PhiJoin Stmt를 추가하게 된다. (그림 4)는 PhiJoinStmt가 추가된 후의 그래프를 나타낸다.

(그림 4)에서 보듯이 모든 블록과 블록내의 변수들은 변수들의 정의와 사용에 따라 새로운 이름을 갖게 된다. 이렇게 변형된 정보는 추후에 타입 추론과 최적화를 위한 중요한 자료가 된다.

4. 비교분석

실험은 펜티엄 4 2.4GHz, 메모리 512MB를 가진 PC에서 수행하였으며, 사용한 소프트웨어는 CTOC 작성과 테스트를 위해 자바 IDE인 eclipse 3.1을 사용하였다. 자바 컴파일러는 j2sdk1.4.2_03을 사용하였다.

데이터들은 실험 결과의 비교를 위해 Don Lance의 논문의 자료를 이용하였다[4]. <표 6>은 실험에 사용될 프로그램에 대한 간단한 설명이다.

<표 6>의 예제를 이용해서 실험한 항목은 각 프로그램의 원시 소스의 라인 수, 바이트코드의 라인 수, 코드 변경 후 라인 수, 기본 블록 수, 간선의 수, 전체 노드의 수 등이다. <표 7>은 실험 결과이다.

<표 7>에서 소스(no.)는 소스 코드의 라인 수를 의미하고, 바이트코드(no.)는 javap -c를 이용하여 생성된 바이트코드의 라인 수를 의미한다. 변경 후(no.)는 CTOC를 통해 기본 블록을 생성하기 위해 기존의 코드를 변경하는 과정에서 추가되거나 삭제된 후의 코드 라인 수를 의미한다. 기본 블록(ea)은 변경된 코드와 기본 블록을 위한 리더를 통해 생성된 기본 블록의 수를 의미한다. 간선(ea)은 기본 블록과 다른 기본 블록 사이의 관계를 표현하기 위해 사용된 간선의 수를 의미한다. 노드(ea)는 기본 블록 내에 명령어와 문

〈표 6〉 사용 예제와 간단한 설명

프로그램	설 명
SquareRoot	숫자의 제곱근 찾기
SumOfSquareRoots	주어진 숫자 n에 대해 1부터 n까지 제곱근의 합 구하기
Fibonacci	주어진 숫자 n에 대해 피보나치 숫자인 Fn 찾기
BubbleSort	버블 정렬을 이용하여 정수 배열 정렬하기
LabelExample	라벨화된 break와 continue 프로그램
Exceptional	try-catch-finally 예외처리

〈표 7〉 실험 결과

	소스 (no.)	바이트 코드 (no.)	변경 후 (no.)	기본 블록 (ea)	간선 (ea)	노드 (ea)
SquareRoot	37	94	60	15	18	99
SumOfSquareRoot	38	103	63	18	19	108
Fibonacci	42	76	69	18	22	86
BubbleSort	30	79	68	16	21	101
LableExample	28	51	59	13	16	58
Exceptional	41	99	149	26	29	143

〈표 8〉 CTOC와 다른 도구들의 비교

프로그램	도구	기본 블록 (ea)	간선(ea)
SquareRoot	CTOC	15	18
	Soot	8	10
	JAristotle	17	19
SumofSquareRoot	CTOC	18	19
	Soot	8	10
	JAristotle	17	19
Fibonacci	CTOC	18	22
	Soot	9	12
	JAristotle	19	22
BubbleSort	CTOC	16	21
	Soot	11	14
	JAristotle	15	18
LabelExample	CTOC	13	16
	Soot	11	14
	JAristotle	9	12
Exceptional	CTOC	26	29
	Soot	23	28
	JAristotle	24	27

〈표 9〉 정적 단일 배정 형태 변환 후 실험 결과

	CFG lines	SSA lines	%	CFG nodes	SSA nodes	%
Square Root	60	63	4.76	99	117	15.38
Sum Of Square Root	63	71	11.27	108	143	24.48
Fibonacci	69	77	10.39	86	126	31.75
Bubble Sort	68	76	10.53	101	133	24.06
Label Example	59	63	6.35	58	74	21.62
Exceptional	149	177	15.82	143	304	52.96

장을 인식하기 위해 사용된 노드의 개수를 의미한다.

<표 8>에서는 기존에 작성된 CFG를 작성하는 도구인 SOOT, JAristotle와 CTOC를 비교하였다. 비교는 기본 블록의 수와 간선의 수에 대해서 수행하였다.

<표 8>에서 CTOC는 다른 도구들 보다 많은 노드와 간선을 갖는 것을 확인할 수 있다. 이는 좀 더 미세 단위의 처리가 CTOC에서 가능하다는 의미이다. 따라서 기존의 노드에 좀 더 세밀한 정보의 추가에 의해 더 나은 분석과 최적화를 적용할 수 있다는 것을 의미한다. SOOT은 바이트코드의 정보를 이용하여 기본 블록을 생성하는 반면에 JAristotle은 소스 코드 수준에서 기본 블록을 생성하여 CFG를 구성한다.

<표 9>는 SSA Form으로 변환 후에 CFG와 비교한 결과이다.

<표 9>의 경우 SSA Form으로 변환된 이후 전반적인 라인수와 노드의 개수가 증가하는 것을 볼 수 있다. 이는 기존의 제어 흐름 그래프에 존재하지 않았던 ∅-함수의 추가에 의해서 발생하는 것이다.

5. 결 론

본 논문에서는 바이트코드 수준에서 프로그램을 분석하고 최적화에 대한 준비를 수행하였다. 바이트코드 수준에서 분석과 최적화를 수행하기 위해서 우선 CFG를 생성하였다. 바이트코드의 특성 때문에 기존의 제어 흐름 분석 기술을 바이트코드에 적합하게 확장하였다. 또한 정적인 분석을 위해 CFG를 SSA Form으로 변환하였다.

SSA Form으로 변환을 수행하기 위해 CFG, 지배자 관계, 지배자 트리, 직접 지배자, ∅-함수, 재명명, 지배자 경계 등 많은 정보에 대한 계산을 수행하였다.

또한 본 논문에서는 기존의 제어 흐름 그래프에서 SSA Form으로 변환하기 위한 알고리즘을 제시하고 변환 과정을 기술하였다. 이렇게 해서 생성된 SSA Form이 적용된 그래프는 추후에 타입 추론과 최적화를 위해 사용되어진다.

참 고 문 헌

[1] Tim Linholm and Frank Yellin, 'The Java Virtual Machine Specification,' The Java Series, Addison Wesley, Reading, MA, USA, Jan. 1997.

[2] James Gosling, Bill Joy, and Guy Steel, 'The Java Language Specification' The Java Series, Addison Wesely, 1997.

[3] Soot, http://www.sable.mcgill.ca/soot

[4] JAristotle, http://www.mtsu.edu/~java

[5] Bloat, http://www.cs.purdue.edu/s3/projects/bloat

[6] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, 'Compilers

Principles, Techniques and Tools,' Addison Wesley, 1986.

[7] Andrew W. appel, 'Modern Compiler Implementation in Java.' CAMBRIDGE UNIVERSITY PRESS, 1998.

[8] 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," Mar. pp.451-490, 1991.

[9] 김경수, 김기태, 조선문, 유원희, "CTOC에서 스택 기반 코드를 효율적인 중간 코드로 변환기 설계," 제 22회 한국정보처리학회 추계발표대회 논문집 제 11권 제 2호, pp.429-432, 2004. 11.

[10] 김기태, 이갑래, 유원희, "CTOC에서 정적 단일 배정문 형태를 이용한 지역 변수 분리," 한국콘텐츠학회 논문지 제 5권 제3호, pp.73-81, 2005. 6.

김 기 태

e-mail : kkt@inha.ac.kr

1999년 상지대학교 전자계산학과(학사)

2001년 인하대학교 전자계산공학과 (공학석사)

2003년 인하대학교 전자계산공학과 (공학박사수료)

2004년~2006년 인하대학교 컴퓨터공학부 강의전임강사

관심분야 : 컴파일러, 프로그래밍언어, 정보보안

유 원 희

e-mail : whyoo@inha.ac.kr

1975년 서울대학교 응용수학과(이학사)

1978년 서울대학교 대학원 계산학 (이학석사)

1985년 서울대학교 대학원 계산학 (이학박사)

1979년~현재 인하대학교 컴퓨터공학부 교수

관심분야 : 컴파일러, 프로그래밍언어, 병렬시스템