韓國 컴퓨터情報學會 論文誌
第13卷 第5號, 2008. 9.

2008-13-5-1-4

CTOC에서 루프 벗기기 구현

김 기 태*, 김 제 민**, 유 원 희***

Implementation of Loop Peeling in CTOC

Kim Ki Tae*, Kim Je Min**, Yoo Weon Hee***

요 약

최근 웹 어플리케이션 분야에서 많이 사용되고 있는 자바 바이트코드의 분석과 최적화 과정을 효율적으로 수행하기 위해 CTOC 프레임워크를 개발하였다. CTOC는 바이트코드에 대해 분석과 최적화를 수행하기 위해 E-Tree를 중간 표현으로 사용하는 eCFG를 생성한다. eCFG는 바이트코드에 대한 제어 흐름 분석에 적합하도록 확장한 제어 흐름 그래프이다. 또한, 바이트코드를 정적으로 분석하기 위해 E-Tree를 SSA Form으로 변환한다. 이러한 변환 과정 중 프로그램의 많은 부분에서 루프가 발견된다. 하지만 기존의 CTOC에서는 루프에 대한 처리를 수행하지 않은 상태에서 직접 SSA Form으로 변환을 수행하였다. 하지만 SSA Form으로 변환 이전에 루프를 처리하면 더욱 효율적인 SSA Form을 생성할 수 있게 된다. 따라서 본 논문에서는 루프에 대한 처리를 효율적으로 하기 위해 E-Tree를 SSA Form으로 변환하는 과정 이전에 eCFG에서 루프를 발견하고 이와 관련된 루프 트리를 생성한 후 루프 벗기기를 수행하는 과정을 보인다.

Abstract

The CTOC framework was implemented to efficiently perform analysis and optimization of the Java bytecode that is often being used lately. In order to analyze and optimize the bytecode from the CTOC, the eCFG was first generated. Due to the bytecode characteristics of difficult analysis, the existing bytecode was expanded to be suitable for control flow analysis, and the control flow graph was drawn. We called eCFG(extended Control Flow Graph). Furthermore, the eCFG was converted into the SSA Form for a static analysis. Many loops were found in the conversion program. The previous CTOC performed conversion directly into the SSA Form without processing the loops. However, processing the loops prior to the SSA Form conversion allows more efficient generation of the SSA Form.

This paper examines the process of finding the loops prior to converting the eCFG into the SSA Form in order to efficiently process the loops, and exhibits the procedures for generating the loop tree.

▸ Keyword : CTOC(CTOC), 정적 단일 배정 형태(SSA Form), 루프(loop), 루프 트리(loop tree), 루프 벗기기(loop peeling)

• 제1저자 : 김기태
• 접수일 : 2008. 4. 18, 심사일 : 2008. 5. 27, 심사완료일 : 2008. 9. 25.
* 인하대학교 컴퓨터정보공학부 강의전임강사 ** 인하대학교 정보공학과 박사과정
** 인하대학교 컴퓨터정보공학부 교수
※ 이 논문은 인하대학교의 지원에 의하여 연구되었습니다.

I. 서 론

자바 바이트코드의 분석과 최적화를 위해 CTOC가 개발되었다[3, 4, 5, 6]. CTOC는 바이트코드에 라벨 정보를 추가하여 변환을 쉽게하고, E-Tree (Expression Tree)라 불리는 트리 형태의 중간 표현을 사용하여 바이트코드에 대한 분석과 최적화를 수행하는 프레임워크이다.

CTOC는 바이트코드에 대한 제어 흐름 분석을 수행하기 위해 기존의 제어 흐름 분석 기술을 자바 바이트코드에 적합하게 확장하였다. 바이트코드 수준의 분석을 위해 우선 eCFG(extended CFG)를 생성한다[3]. 생성된 eCFG는 유향 그래프(directed graph)이며, 제어 흐름 정보를 표현한다. CTOC의 eCFG는 일반적인 CFG에 시작 블록과 종료 블록뿐만 아니라 매개 변수와 같은 정보를 유지하기 위한 초기화 블록을 가진 확장된 CFG를 의미한다.

기존 CTOC는 루프에 대한 식별 없이 단순히 eCFG를 생성한 후 SSA Form으로 변환을 수행하였다[4, 5]. 하지만 일반적으로 프로그램 수행에 있어 많은 시간이 루프를 수행하는데 소비된다고 알려져 있다. 따라서 루프는 최적화를 구현하는데 아주 중요한 부분 중에 하나이다. 비록 최적화 수행 중에 루프의 외부 코드가 증가된다 할지라도 내부 루프의 명령어가 줄어든다면 프로그램의 수행 시간을 크게 줄일 수 있기 때문이다[7, 8, 9]. 따라서 프로그램 내에서 루프를 발견하고 변환을 수행하는 것은 중요한 작업이다. 또한 루프를 처리하는 시기도 중요하다. SSA Form을 사용하는 경우 SSA Form으로 변환한 후에 루프에 관한 처리를 수행하는 것보다 SSA Form 변환 전에 루프에 관한 처리가 수행되는 것이 더욱 효율적이다. 왜냐하면 루프를 처리하는 과정에서 코드가 옮겨지거나 새로운 변수를 생성하는 등 기존의 eCFG에 변화가 발생할 수 있기 때문이다. 게다가 변경된 eCFG를 SSA Form으로 변환하는 것이 SSA Form 변환 후 생성된 중간 코드를 이용해서 루프 처리를 수행하는 것보다 단순하기 때문이다.

루프를 발견하기 위한 대표적인 관련 연구로는 Tarjan의 인터벌 찾기 알고리즘이 있다[10]. 이것은 줄일 수 있는(reducible) 그래프로 제한되는 전통적인 루프 찾기 방법이다. Havlak은 Tarjan의 알고리즘을 확장하여 줄일 수 있는 그래프뿐만 아니라 줄일 수 없는(irreducible) 그래프까지 처리하는 방법을 제안 하였다[11]. 그리고 Sreedhar-Gao-Lee는 루프 중첩 숲(loop-nesting forest)을 생성하기 위해 Havlak과는 다른 방법을 제안하였다[12]. 그들은 DJ 그래프라 불리는 새로운 그래프를 사용하였다. DJ 그래프는 CFG와 지배자 트리를 묶어 하나의 구조로 다루는 그래프이다. Steensgaard는 그래프에서 루프 식별을 위해 하향식 방식을 사용하고, 외부 루프를 제일 먼저 찾는 방식을 사용하였다[13].

CTOC에서 사용한 루프의 식별은 Havlak의 방법과 Steensgaard의 방법을 기반으로 작성한다. 루프를 식별한 후 루프 트리를 생성하고 이를 이용하여 루프 반전(loop inversion)과 루프 벗기기(loop peeling)를 수행한다. 루프 트리(loop tree)란 루프의 헤더를 루트로 갖는 특수한 트리이다. 루프 반전과 루프 벗기기는 Wolfe에 의해 제안된 방법으로 루프를 다루기 위해 사용되는 방법이다.

본 논문에서는 생성된 루프와 루프 트리 그리고 루프 벗기기의 결과를 확인하기 위해 eCFG와 루프 트리를 작성하여 CTOC의 처리 결과를 확인한다. 또한 루프 처리로 인해 기존의 eCFG와 어떻게 달라지는지 확인한다. 본 논문은 루프 최적화 과정보다는 루프의 발견과 루프 트리 생성, 그리고 루프 벗기기 과정에 중점을 둔다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로 루프에 대한 정의와 루프 반전, 그리고 루프 벗기기에 대해서 간단히 설명한다. 3장은 루프 처리를 위해 사용할 간단한 예제와 생성된 제어 흐름 그래프 정보 그리고 생성된 eCFG를 보인다. 4장에서는 루프를 식별하는 과정과 루프 트리를 생성하는 과정을 기술한다. 5장에서는 루프 벗기기 수행 과정을 기술한다. 마지막으로 6장에서는 결론과 향후 계획을 제시한다.

II. 관련연구

일반적인 고급 언어의 경우의 루프와 루프 반전, 그리고 루프 벗기기는 그림 1과 같이 표현된다.

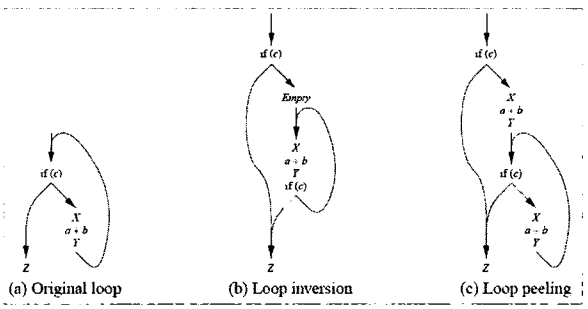

그림 1. 루프 벗기기 예제
Fig. 1 Example of Loop Pealing

그림 1에 대한 자세한 설명은 2.2와 2.3에 있다.

2.1 루프

프로그램의 실행 속도를 높이기 위해서는 제어 흐름 그래프에 존재하는 루프에 대한 식별이 요구된다. 제어 흐름 그래프에서 루프는 일반적으로 다음과 같이 정의된다. 첫째, 루프 안의 모든 노드는 강하게 연결(strongly connected)되어 있다. 즉, 루프의 어느 한 노드에서 다른 어떤 노드로든 갈 수 있는 경로가 존재한다. 둘째, 루프는 오직 하나의 입구(entry)가 존재한다. 루프 밖에서 루프 안으로 도달하기 위해서 항상 먼저 입구를 통과해야 하고, 이러한 입구는 오직 하나이다.

2.2 루프 반전

루프 반전은 만약 루프가 적어도 한 번 이상 수행된다면 코드 수행을 위해 루프 위에 코드를 삽입하기 위한 장소를 제공하는 변환이다. 이러한 변환은 고급언어에서 while을 do-while 구문으로 변환하는 것과 유사하다. 예를 들면 그림 1(a)에서 표현식 a + b를 루프 밖으로 끌어올리기를 원하는 경우라고 하자. 만약 루프가 적어도 한번 들어온다면 프로그램은 a + b를 루프 밖으로 올릴 수 있지만 if 구문이 거짓이라면 그렇지 않은 경우도 존재하게 된다. 따라서 구문을 단순히 루프 밖으로 옮길 수는 없다. 루프 반전은 빈 프리헤더 블록을 추가하여 a + b를 그림 1(b)와 같이 if 구문 위에 두는 프로그램으로 변환하는 것이다. 루프 반전은 유일한 루프 입구를 요구한다.

2.3 루프 벗기기

루프 벗기기는 새로운 제어 흐름 그래프에서 루프 몸체를 복사하고 간선을 수정함으로써, 루프의 첫 번째 반복을 추출한다. 간선을 수정하기 위해, 복사된 역간선(backedge)은 원래의 루프 헤더로 간선과 교체되고 루프 외부로부터의 간선은 새로운 루프 헤더를 가리키도록 만든다. 그림 1(c)은 (a)의 루프가 벗겨진 것을 보여준다. 루프 반전과 마찬가지로 루프 벗기기 역시 단일 입구를 가진 경우 수행 가능하다. 루프 벗기기는 루프 반전을 포함하는 개념이지만 루프 벗기기는 프로그램의 크기가 지수승으로 커진다는 단점이 존재한다. 따라서 CTOC에서는 프로그램의 내부 루프에 대해서만 루프 벗기기를 수행한다. 왜냐하면 내부 루프들이 가장 빈번하게 수행되기 때문이다.

III. eCFG

public void loop() {
int n = 5;
int x = 0;
for (int a=0; a<n; a++)
for (int b=0; b<n;
b++)
x++;
}
	public void loop();
	Code:
	0: iconst_5
	1: istore_1
	2: iconst_0
	3: istore_2
	4: iconst_0
	5: istore_3
	6: iload_3
	7: iload_1
	8: if_icmpge 35
	11: iconst_0
	12: istore 4
	14: iload 4
	16: iload_1
	17: if_icmpge 29
	20: iinc 2, 1
	23: iinc 4, 1
	26: goto 14
	29: iinc 3, 1
	32: goto 6
	35: return

(a) 중첩된 루프	(b) 역어셈블 된 바이트코드

그림 2. 중첩된 루프의 예제
Fig. 2 Example of Nested Loop

CTOC에서 루프 식별과 루프 트리 생성 그리고 루프 벗기기를 수행하기 위해 그림 2와 같은 중첩된 루프 예제 를 사용한다.

그림 2(a)는 중첩된 루프 예제이고 그림 2(b)는 그림 2(a)를 javap -c 옵션을 통해 생성한 역어셈블된 바이트코드이다. CTOC의 입력은 (a)의 소스 형태가 아니라 바이트코를 읽어 들인다. 실제 CTOC에서는 (b)의 역어셈블된 형태가 아닌 바이너리 형태의 .class 파일을 입력으로 사용한다. 여기서는 이해를 돕기 위해 역어셈블된 바이트코드로 표시한다.

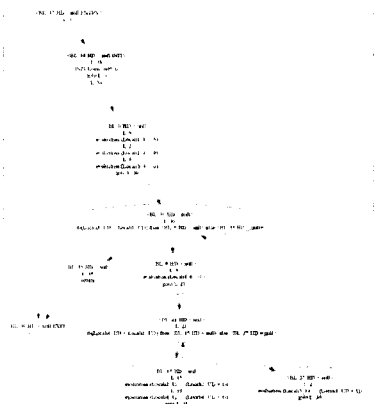

그림 3. eCFG
Fig. 3 eCFG

그림 3은 그림 2(b)의 정보를 이용하여 CTOC가 생성한 eCFG의 모습이다. 그림 3과 같은 eCFG를 생성하는 과정의 자세한 사항은 [3]을 참조한다. 그림 3에 생성된 eCFG는 블록과 블록 내용에 관한 정보와 블록들 간의 관계를 이용하여 생성된 그래프이다. 각 블록의 이름은 첫 번째 나타나는 레이블의 이름과 동일하기 때문에 〈BL_37〉, 〈BL_38〉 그리고 〈BL_39〉는 각각 시작, 초기화 그리고 종료 블록을 나타낸다. eCFG는 각 블록이 가지는 선행자와 후행자의 정보를 포함한다.

IV. 루프 트리 찾기

4.1 기본 정보 설정하기

루프 트리를 생성하기 위해서는 깊이 우선 탐색을 통해 방문한 노드에 대한 전위순서와 후위순서, 그리고 방문한 노드에 대한 정보들을 획득한다. 루프 트리를 생성하기 위해서는 모든 노드를 앞에서 생성한 전위순서로 방문한다.

그림 3에서 보듯이 시작 블록 〈BL_37〉과 종료 블록 〈BL_39〉을 제외한 경우, 각각의 기본 블록에는 선행자와 후행자가 존재한다. 시작 블록은 선행자가 존재하지 않고, 종료 블록은 후행자가 존재하지 않는다. CTOC에는 선행자 정보만을 이용하여 루프의 헤더를 결정한다. 선행자 정보는 깊이 우선 탐색 후 두 가지로 나눌 수 있는데, 하나는 선행자가 자신보다 작은 전위순서를 가진 경우이고 또 다른 하나는 선행자가 자신보다 큰 전위순서를 가진 경우이다. 루프를 식별하기 위해서는 역간선을 알아야 하기 때문에 이러한 선행자의 정보는 아주 중요하다.

순방향의 선행자인 FP(Forward Predecessor)와 역방향의 선행자인 BP(back Predecessor)를 집합 배열로 설정한 후 현재 존재하는 블록의 크기만큼 생성한다. 이때, FP는 현재 노드가 선행자 노드보다 깊이 우선 탐색 순서가 작은 순방향 선행자 집합을 나타낸다. 반면 BP는 현재 노드가 선행자 노드보다 깊이 우선 탐색 순서가 큰 역방향 선행자 집합을 나타낸다.

블록을 하나씩 방문하면서 각 블록에 대한 초기화 동작을 수행한다. 우선 모든 블록에 대해서 각 블록이 가지는 FP와 BP를 null로 초기에 설정한다. 그리고 모든 블록의 헤더를 시작 블록으로 설정한다. 하지만 시작 블록의 경우에는 선행자를 갖지 않기 때문에 시작 블록의 헤더는 null로 설정한다.

각 블록에 대해 FP와 BP를 설정하는 과정은 세 가지 경우로 나눌 수 있다. 첫 번째 경우는 그림 3에서 〈BL_37〉의 경우 시작 블록이기 때문에 이 블록에 대한 선행자는 존재하지 않는다. CTOC는 선행자에 대해서만 고려하기 때문에 이 경우 특별한 동작이 일어나지 않는다. 두 번째 경우는 〈BL_39〉와 같은 종료 블록의 경우이다. 이 블록의 선행자는 시작 블록인 〈BL_37〉과 〈BL_ 35〉가 해당된다. 선행자와 현재 블록의 관계를 확인하는 메소드인 isBackEdge(w, v)를 통해 현재 블록 w에 대해 선행자인 v가 조상인가를 확인하여, v가 조상인 경우는 자손에서 조상으로 역간선이 존재하는 경우이기 때문에 BP에 해당 블록에 대한 정보를 저장한다. 역간선이 아닌 경우라면 FP에 해당 선행자를 추가한다. 즉, 각 블록별로 순차간선과 역간선의 정보가 수집된다. 따라서 종료 블록인 〈BL_39〉의 경우에는 시작 블록 〈BL_37〉과 바로 앞의 〈BL_35〉이 종료 블록 〈BL_39〉의 FP에 저장된다. 반면 BP는 존재하지 않기 때문에 여전히 null이 설정된다. 세 번째의 경우는 두 가지 값이 다 설정되는 경우이다. 〈BL_30〉의 경우는 순차간선도 존재하고 역간선도 존재하는 경우이다. 〈BL_30〉의 선행자는 〈BL_0〉과 〈BL_27〉이다. 이중에 〈BL_0〉는 〈BL_30〉의 조상으로 존재하는 경우이기 때문에 이전과 같이 FP에 추가되지만, 〈BL_27〉의 경우는 〈BL_0〉보다 깊이 우선 탐색에서 늦게 나타나는 경우이기 때문에 역선행자에 해당한다. 따라서 BP에 추가된다. 따라서 〈BL_30〉을 수행한 후 FP는 〈BL_0〉가 설정되고 BP에는 〈BL_27〉이 설정 된다. 위와 같은 과정을 모두 수행한 후 각 블록의 FP와 BP의 내용은 표 1과 같다.

표 1 각 블록별 FP와 BP의 내용
Table 1 Contents of FP and BP

pre	Block	FP	BP	header
0	BL_37	∅	∅	null
1	BL_39	{BL_35, BL_37}	∅	BL_37
2	BL_38	{BL_37}	∅	BL_37
3	BL_0	{BL_38}	∅	BL_37
4	BL_30	{BL_0}	{BL_27}	BL_37
5	BL_9	{BL_30}	∅	BL_37
6	BL_21	{BL_9}	{BL_15}	BL_37
7	BL_15	{BL_21}	∅	BL_37
8	BL_27	{BL_21}	∅	BL_37
9	BL_35	{BL_30}	∅	BL_37

표 1에서 굵게 표현된 부분을 살펴보면, 전위순서 0인 〈BL_37〉의 경우 시작 노드이기 때문에 순차선행자도 역선행자도 존재하지 않는다. 또한 시작되는 부분이기 때문에 헤더도 null로 설정된다. 〈BL_39〉의 경우는 종료 블록이기 때문에 시작 블록으로부터 오는 간선과 모든 수행 후 종료 간선으로 오는 두 개의 선행자가 존재한다는 것을 확인 할 수 있다. 마지막으로 블록 〈BL_30〉과 〈BL_21〉은 순차선행자와 역선행자를 가진 경우를 보인다. 그리고 이 시점에서 시작 블록을 제외한 모든 블록은 초기화 동작에 의해 시작 블록 〈BL_37〉을 헤더로 갖게 된다.

4.2 루프 헤더 설정하기

각 블록별로 초기화 동작으로 선행자에 대한 정보를 획득한 후 eCFG에 루프가 존재하는지 확인하기 위해서는 전위순서의 역으로 정보를 찾아야 한다. 왜냐하면 각 내부 루프의 헤더 정보를 외부 루프의 헤더 정보보다 먼저 찾아내기 위해서이다. 알고리즘 1은 루프에서 헤더를 설정하는 알고리즘이다. 제안된 알고리즘은 Havlak과 Steensgaard의 알고리즘을 변형한 것이다[11, 13].

알고리즘 1 루프 헤더 설정 알고리즘
Algorithm 1 Loop Header Setting

```
Input : node ∈ Set of Node
Output : node ∈ Set of Node
Procedure setHeader()
begin
    for each node w ∈ reverse preorder
        body ← {}
        for each node v ∈BP[w]
            if (v not equal w)
                vn ← preOrderIndex(v)
                f ← findNode(vn)
                add f to body
            fi
        endfor
        if (body equal {})
            continue;
        fi
        worklist ← body
        while (worklist not equal {})
            select a node x ∈ worklist and delete from worklist
            for each node y ∈ FP[x]
                z ← findNode(y)
                if(z ∈ body) and (z not equal w)
                    add z to body and worklist
                fi
            endfor
        endwhile
    endfor
    for each node x ∈ body
        header[x] ← w
        union(x, w)
    endfor
end
```

만약 제어 흐름 그래프 내에 루프가 존재하는 경우라면 알고리즘 1을 사용하여 루프의 헤더를 설정 한다. 이를 위해 각 블록의 FP 정보와 BP 정보가 필요하다. 우선 블록을 전위 순서의 역순으로 불러온다. 고려되는 부분은 해당 블록에 BP의 정보가 존재하는 경우이면서 현재 블록과 현재 블록의 역선행자가 같지 않은 경우라면, 메소드 findNode(y)를 수행하고 이 때 찾아낸 블록을 루프 몸체에 해당하는 body에 추가한다. 역선행자의 전위순서 인덱스인 y에 대해 findNode(y)를 수행하면 해당 블록이 속한 블록을 알 수 있게 된다.

이후에 하나의 루프로 정보를 유지하기 위해서는 해당 루프에 해당하는 것끼리 병합해야 한다. 이를 위해서는 union(x, w) 메소드를 사용한다. 이 메소드는 하나의 루프에 속하는 노드들에 대해 헤더에 해당하는 노드를 기준으로 트리를 생성한다. union(x, w)은 x에 대해 w를 헤더로 설정한다.

동작이 수행된 후 FP, BP, 그리고 헤더의 정보는 표 2와 같이 변경된다.

표 2 각 블록별 변경된 FP, BP와 HD의 내용
Table 2 Modified Contents of FP, BP and HD

pre	Block	FP	BP	HD
0	BL_37	∅	∅	null
1	BL_39	{BL_35, BL_37}	∅	BL_37
2	BL_38	{BL_37}	∅	BL_37
3	BL_0	{BL_38}	∅	BL_37
4	BL_30	{BL_0}	{BL_27}	BL_37
5	BL_9	{BL_30}	∅	BL_30
6	BL_21	{BL_9}	{BL_15}	BL_30
7	BL_15	{BL_21}	∅	BL_21
8	BL_27	{BL_21}	∅	BL_30
9	BL_35	{BL_30}	∅	BL_37

표 2의 정보를 이용하여 루프 트리를 그리면 그림 4와 같다.

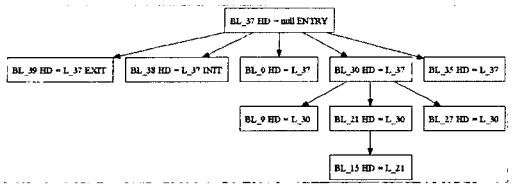

그림 4. 루프 트리
Fig. 4 Loop Tree

그림 4의 루프 트리를 보면 아래쪽에서부터 살펴보면 〈BL_15〉의 경우 헤더로 〈BL_21〉을 가진다. 이것의 의미는 〈BL_15〉와 〈BL_21〉이 하나의 내부 루프를 이룬다는 의미이다. 그리고 〈BL_21〉의 경우엔 헤더로 〈BL_30〉을 가진다. 하지만 이것의 의미는 〈BL_30〉, 〈BL_9〉, 〈BL_21〉, 〈BL_27〉이 또 다른 루프를 구성한다는 의미이다. 현재 루프 트리를 보면 깊이가 3인 것을 확인할 수 있다. 즉, 2개의 중첩된 루프를 가지며 특히 트리의 레벨은 내부 루프와 외부 루프를 나타내고 이 정보를 확인하면 루프의 개수와 중첩된 정보를 확인할 수 있게 된다. 이 정보를 이용하여 루프 반전과 루프 벗기기를 수행한다.

V. CTOC에서 루프 벗기기

5.1 루프 정보 모으기

루프 벗기기를 수행하기 위해서는 루프 트리를 생성하는 동안 작성된 루프의 레벨과 깊이에 대한 정보가 필요하다.

그림 5는 루프 트리에 대한 레벨과 깊이에 관한 정보를 다시 표현한 것이다.

```
1 : [Loop] : level=1 depth=2 header=<BL_21 HD = L_30>
  * [Elements] : [<BL_21 HD = L_30>, <BL_15 HD = L_21>]

2 : [Loop] : level=2 depth=1 header=<BL_30 HD = L_37>
  *[Elements] : [<BL_30 HD = L_37>, <BL_21 HD = L_30>,
          <BL_9 HD = L_30>, <BL_27 HD = L_30>]

3 : [Loop] : level=3 depth=0 header=<BL_37 HD = null ENTRY>
  *[Elements] : [<BL_30 HD = L_37>, <BL_0 HD = L_37>,
          <BL_37 HD = null ENTRY>, <BL_39 HD = L_37 EXIT>,
          <BL_35 HD = L_37>, <BL_38 HD = L_37 INIT>]
```

그림 5. 루프 트리의 레벨과 깊이에 대한 정보
Fig. 5 Information of Level and Depth of Loop Tree

그림 5에서 1: 부분에 대해 설명하면, 가장 내부에 위치한 루프는 레벨 1로 설정된다. 물론 루프가 존재하지 않는다면 0의 값을 갖게 된다. 그림 5에서는 level이 1로 가장 안쪽에 위치하고 있다는 것을 의미하고, 중첩된 정도를 나타내는 depth는 2로써 2번째로 중첩된 루프라는 의미를 가진다. 루프의 헤더를 의미하는 header는 〈BL_21 HD=30〉으로서 현재 루프의 헤더는 〈BL_21〉이고, 〈BL_21〉이 속한 루프의 헤더는 〈BL_30〉이라는 것을 나타낸다. 아래쪽에 나오는 Elements는 현재 루프를 구성하는 요소를 나타낸다. 가장 아래쪽에 위치한 블록의 구성은 그림 3에서 보이는 것과 같이 〈BL_21〉과 〈BL_15〉로 되어 있다.

5.2 루프 벗기기

루프 벗기기를 수행하기 위해서는 현재 루프의 헤더 정보를 가져와 현재 루프가 벗겨질 수 있는 경우인가를 확인해야 한다. 그림 3의 예제의 경우를 보면 현재 2개의 중첩된 루프가 존재한다. 본 논문에서는 가장 깊은 곳에 존재하는 루프를 벗기는 과정을 설명하고 전체 결과는 최종 생성되는 그래프를 통해서 확인한다.

루프 벗기기를 수행하기 위해서는 현재 루프 헤더의 후행자와 선행자 정보를 이용해야 한다. 가장 안쪽에 존재하는 루프는 〈BL_21〉과 〈BL_15〉로 구성된 경우이다. 이 루프의 헤더는 〈BL_21〉이다. 헤더의 후행자는 〈BL_15〉와 〈BL_27〉이다. 이때 〈BL_15〉는 현재 루프에 존재하는 후행자이고 〈BL_27〉은 루프 외부에 존재하는 블록이다. 후행자들이 루프 내부에도 존재하고 외부에도 존재하는 경우라면 루프를 반전시킬 수 있는 상태가 된다. 따라서 루프를 반전 시키고 빈 블록을 삽입하는 과정을 수행해야 한다. 새로운 빈 블록을 생성하기 위해 HashSet으로 된 copySet을 생성한다. 현재 루프 반전이 가능한 경우이기 때문에 copySet에 현재 헤더를 추가한다. 현재 copySet의 내용은 {〈BL_21〉}이 된다.

루프에 copySet으로부터 가져온 블록이 존재하는 경우라면 새로운 복사된 블록(copy)을 생성한다. 이 경우 현재는 아무런 내용도 없는 상태이기 때문에 copy는 null로 설정된다. 만약 현재 복사된 블록이 null인 경우라면 새로운 블록을 생성해야 한다. 새로 생성된 블록은 〈BL_40〉이 된다. 〈BL_40〉에는 블록을 생성하면서 블록의 시작 위치를 알려주는 라벨 정보와 루프 헤더의 마지막 문장이었던 if(Locali4_UD 〈 Locali1_UD) then 〈BL_15〉 else 〈BL_27〉 이 추가된 상태로 생성된다.

HashMap으로 된 copies에 현재 블록과 새로 생성된 블록을 추가한다. 현재 상태에서 copies의 내용은 {〈BL_21〉 = 〈BL_40〉}이 된다.

새롭게 블록이 생성되었기 때문에 생성된 새로운 블록을 추적(trace) 리스트에 추가한다.

표 3은 추적 리스트에 새로운 블록을 추가하기 전과 후의 결과이다.

표 3 추적 리스트의 내용
Table 3 Contents of Trace List

[illegible]	[illegible]
수행 전	⟨BL_38 INIT⟩, ⟨BL_0⟩, ⟨BL_9⟩, ⟨BL_15⟩, ⟨BL_21⟩, ⟨BL_27⟩, ⟨BL_30⟩, ⟨BL_35⟩
수행 후	⟨BL_38 INIT⟩, ⟨BL_0⟩, ⟨BL_9⟩, ⟨BL_40⟩, ⟨BL_15⟩, ⟨BL_21⟩, ⟨BL_27⟩, ⟨BL_30⟩, ⟨BL_35⟩

표 3의 수행 후를 보면 새로운 블록인 ⟨BL_40⟩이 추가된 것을 확인할 수 있다.

다음에 수행할 동작은 새로 추가된 블록에 간선을 추가하는 것이다. 현재 블록은 ⟨BL_21⟩이고 복사된 블록은 ⟨BL_40⟩인 상태이다. 현재 블록인 ⟨BL_21⟩의 후행자 정보와 후행자의 복사본이 존재하는지 확인한다. ⟨BL_21⟩의 후행자로 ⟨BL_15⟩와 ⟨BL_27⟩이 존재한다. 우선 ⟨BL_15⟩에 대해 살펴보면, 만약 후행자가 헤더가 아니고 후행자의 복사본이 null이 아니라면 복사복과 후행자의 복사본 사이에 간선을 추가하지만 그렇지 않은 경우라면 복사복과 후행자를 연결하게 된다.

따라서 ⟨BL_40⟩과 ⟨BL_15⟩사이에 간선이 추가된다. 두 번째 ⟨BL_27⟩의 경우 마찬가지로 헤더도 아니고 후행자의 복사본이 null이기 때문에 복사복과 후행자를 간선으로 연결하게 된다. 즉, ⟨BL_40⟩과 ⟨BL_27⟩사이에 간선이 추가된다.

다음은 현재 블록인 ⟨BL_21⟩의 선행자 정보를 이용하여 기존에 존재하던 간선을 제거한다. 이를 위해서는 우선 현재 블록의 선행자 정보를 가져온다. 현재 블록의 선행자는 ⟨BL_15⟩와 ⟨BL_9⟩이다. 이 때, 선행자가 현재 루프에 존재하는지 확인한 후 존재하지 않는 경우라면, 선행자와 복사된 블록 사이에 간선을 추가한다. 따라서 ⟨BL_9⟩와 ⟨BL_40⟩ 사이에 새로운 간선이 추가된다. 새로운 간선이 추가되었기 때문에 기존에 존재하는 ⟨BL_9⟩와 ⟨BL_21⟩ 사이의 간선은 제거되어야 한다. 간선을 제거한 후 선행자 블록을 방문하여 선행자 블록인 ⟨BL_9⟩에서 ⟨BL_21⟩으로의 간선이 ⟨BL_9⟩에서 ⟨BL_40⟩으로 바뀌었음을 표현 트리에 나타내어야 한다. 이를 위해 ReplaceTarget(block, copy) 클래스를 사용한다. 이때 block은 현재 블록이고 copy는 새로 생성된 블록이다.

표 4 loop peeling 결과
Table 4 Results of Loop Peeling

block	succs	preds
⟨BL_0 HD = L_37⟩	⟨BL_41 HD = null⟩	⟨BL 38 HD = L_37 INIT⟩
⟨BL_35 HD = L_37⟩	⟨BL_39 HD = L 37 EXIT⟩	⟨BL 30 HD = L_37⟩, ⟨BL_41 HD = null⟩
⟨BL_21 HD = L_30⟩	⟨BL_15 HD = L_21⟩, ⟨BL 27 HD = L 30⟩	⟨BL_15 HD = L_21⟩
⟨BL_38 HD = L_37 INIT⟩	⟨BL_0 HD = L_37⟩	⟨BL 37 HD = null ENTRY⟩
⟨BL_39 HD = L_37 EXIT⟩	∅	⟨BL_37 HD = null ENTRY⟩, ⟨BL_35 HD = L_37⟩
⟨BL_37 HD = null ENTRY⟩	⟨BL 39 HD = L 37 EXIT⟩, ⟨BL 38 HD = L_37 INIT⟩	∅
⟨BL_9 HD = L_30⟩	⟨BL_40 HD = null⟩	⟨BL_30 HD = L 37⟩, ⟨BL_41 HD = null⟩
⟨BL_40 HD = null⟩	⟨BL_15 HD = L_21⟩, ⟨BL 27 HD = L 30⟩	⟨BL_9 HD = L 30⟩
⟨BL_27 HD = L_30⟩	⟨BL_30 HD = L_37⟩	⟨BL_40 HD = null⟩, ⟨BL_21 HD = L_30⟩
⟨BL_41 HD = null⟩	⟨BL_9 HD = L_30⟩, ⟨BL 35 HD = L_37⟩	⟨BL 0 HD = L_37⟩
⟨BL_30 HD = L_37⟩	⟨BL_9 HD = L 30⟩, ⟨BL 35 HD = L 37⟩	⟨BL_27 HD = L_30⟩
<BL_15 HD = L_21>	<BL_21 HD = L_30>	<BL_40 HD = null>, <BL_21 HD = L_30>

ReplaceTarget 클래스를 통해 〈BL_9〉의 마지막 문장이 JumpStmt인 경우 goto L_21을 goto L_40으로 변경한다. 이렇게 가장 내부에 존재하는 루프에 대해 새로운 블록을 생성하고 새로운 블록과 간선을 연결하면 루프 벗기기가 완성된다.

외부에 존재하는 루프에 대해서도 동일한 과정을 수행하면 두 번째 루프 벗기기도 수행되어진다. 모든 과정을 수행한 후의 결과는 표 4와 같다.

표 4와 같은 Loop peeling의 결과가 적용된 eCFG를 그리면 그림 7과 같이 변경된 그래프가 작성된다.

VI. 결론 및 향후 계획

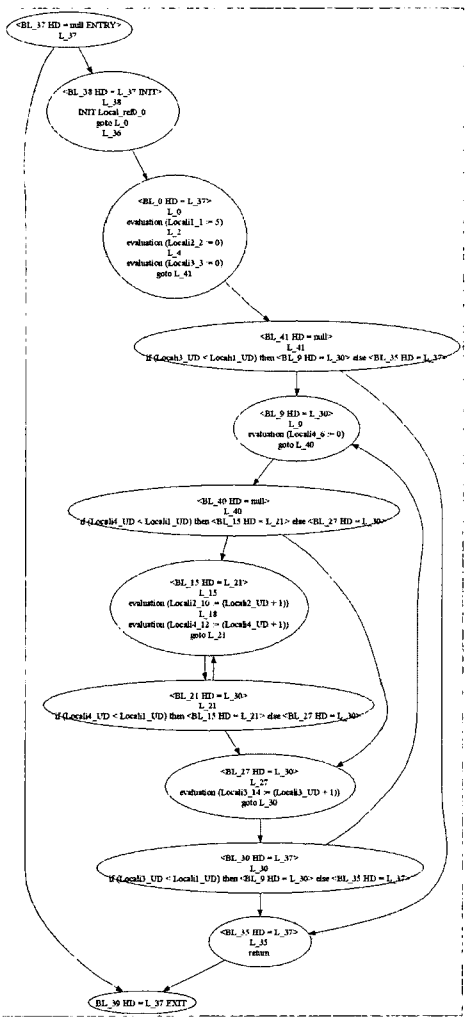

그림 6. 루프 벗기기가 수행된 eCFG
Fig. 6 eCFG which a loop peeling is performed

CTOC에서는 바이트코드를 적절한 형태로 변경한 후,변경된 코드에 대해 제어 흐름 분석을 수행하였다. 이때 사용되는 CTOC는 기존의 바이트코드에 정보를 추가하여 분석을 용이하게 하고 트리 형태의 중간 표현을 이용하여 최적화를 수행하는 프레임워크이다.

이전 연구에서는 루프에 대한 식별없이 단순히 eCFG를 생성한 후 SSA Form을 생성하였다. 하지만 프로그램에서 수행 중에 많은 시간이 루프를 수행하는데 소비된다. 비록 루프에서 외부 코드가 증가된다 할지라도 내부 루프의 명령어를 줄일 수 있다면 프로그램의 수행시간을 크게 줄일 수 있게 된다. 따라서 프로그램 내에서 루프를 발견하는 것은 중요한 작업이라고 할 수 있다.

따라서 본 논문에서는 기존의 eCFG에서 루프를 찾는 방법을 보였다. 기존의 eCFG에서 루프를 찾기 위해서 제일 먼저 깊이 우선 탐색을 통해 방문한 노드에 대한 전위순서와 후위 순서, 그리고 방문한 노드에 대한 정보들을 획득하였다. 이후 각 블록별로 순차선행자와 역 선행자에 대한 정보를 FP와 BP 집합을 통해서 수집하였다. 루프 헤더를 설정하는 과정에서 BP를 이용하여 그래프에 루프가 존재하는지를 확인할 수 있었다. 그리고 루프가 존재하는지가 확인된 후에는 관련된 루프를 묶는 과정에서 각 루프의 헤더를 설정하였다. 이후 헤더 정보를 이용하여 루프 트리를 생성하였다. 루프 트리를 확인하면 현재의 eCFG에 몇 개의 루프가 존재하며 또한 어느 정도 중첩되었는지 확인할 수 있었다. 이들 루프와 루프 트리 정보를 이용하면 추후에 루프에서 적용 가능한 최적화와 분석을 좀 더 효율적으로 적용할 수 있게 된다. 추후 연구에서는 루프 트리를 이용하여 CTOC에서의 루프 최적화와 분석을 적용할 것이다.

참고문헌

[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] 김기태, 유원희, "CTOC에서 자바 바이트코드를 이용한 제어 흐름 분석에 관한 연구", 한국콘텐츠학회 논문지 제6권 제1호, pp. 160-169, 2006

[4] 김기태, 유원희, "CTOC에서 자바 바이트코드를 위한 정적 단일 배정 형태", 정보처리학회논문지 D 제 13-D

권 제 7호, pp. 939-946, 2006
[5] 김기태, 유원희, "CTOC에서 코드 최적화 수행", 멀티미디어학회 논문지 제10권 제5호, pp. 687-697, 2007
[6] 김기태, 김지민, 김제민, 유원희, "CTOC에서 자바 바이트코드 최적화를 위한 Value Numbering", 한국컴퓨터정보학회논문지, 11권6호, pp. 19-26, 2006
[7] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques and Tools, Addison Wesley, 1986
[8] Andrew W. appel, Modern Compiler Implementation in Java. CAMBRIDGE UNIVERSITY PRESS, pp. 437-477, 1998
[9] Muchnick, S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco. 1997
[10] Tarjan, R. E. Testing flow graph reducibility. J. Comput. Syst. Sci. 9, pp. 355-365, 1974
[11] Havlak, P. Nesting of reducible and irreducible loops. ACM Trans. Program. Lang. Syst. 19, 4, 557-567, 1997
[12] Sreedhar, V. C., Cao, G. R., and Lee, Y. F. Identifying loops using DJ graphs. ACM Trans. Program. Lang. Syst. 18, 6, pp. 649-658, 1996
[13] Steensgaard, B. Sequentializing program dependence graphs for irreducible programs. Tech. Rep. MSR-TR-93-14, Microsoft Research, Redmond, Wash. 1993
[14] http://www.graphviz.org/

저 자 소 개

김 기 태(Kim Ki Tae)

1999년 상지대학교 전자계산학과
2001년 인하대학교 전자계산공학과(공학석사)
2008년 인하대학교 정보공학과(공학박사)
2008년 - 현재 인하대학교 컴퓨터정보공학부 강의전임강사
관심분야 : 컴파일러, 프로그래밍언어, 정보보안

김 제 민(Kim Je Min)

2006년 인하대학교 컴퓨터공학부
2008년 인하대학교 정보공학과(공학석사)
2008년 - 현재 인하대학교 컴퓨터정보공학과 박사과정
관심분야 : 컴파일러, 프로그래밍언어, 정보보안

유 원 희(Yoo Weon Hee)

1975년 서울대학교 응용수학과
1978년 서울대학교 대학원 계산학(이학석사)
1985년 서울대학교 대학원 계산학(이학박사)
1979 ~ 현재 : 인하대학교 컴퓨터공학부 교수
관심분야 : 컴파일러, 프로그래밍언어, 병렬시스템