자바 바이트코드 프레임워크 구현
Implementation of Java Bytecode Framework

김기태, 김제민, 유원희
인하대학교 IT 공과대학 컴퓨터정보공학부

Ki-Tae Kim(kimkitae@inha.ac.kr), Je-Min Kim(jeminya@gmail.com),
Weon-Hee Yoo(whyoo@inha.ac.kr)

요약

본 논문에서는 자바 바이트코드를 분석하고 변환할 수 있는 새로운 도구인 CTOC 프레임워크를 설계하고 구현한다. CTOC는 자바 바이트코드의 분석과 코드 변환을 수행할 수 있는 도구로, 바이트코드 분석과 변환 과정을 효율적으로 구현하기 위해 확장된 제어 흐름 그래프인 eCFG(extended Control Flow Graph)와 바이트코드를 효과적으로 처리할 수 있는 중간 코드인 E-Tree(Expression-Tree)를 사용한다. eCFG와 E-Tree를 자바 바이트코드에 대한 분석과 최적화 코드 변환 과정에 적용하여 CTOC의 유용성과 확장 가능성을 보인다.

■ 중심어 : | CTOC | 최적화 | 자바 바이트코드 |

Abstract

In this paper, we design and implement CTOC, a new bytecode analysis and translation tool. We also propose E-Tree, a new intermediate code, to efficiently deal with intermediate codes translated from bytecodes. E-Tree is expressed in a tree form by combining relevant bytecode instructions in basic blocks of eCFG to overcome the weaknesses of bytecodes such as complexity and analytical difficulty. To demonstrate the usefulness and possible extensibility of CTOC, we show the creation process of eCFG and E-Tree through practical bytecode analysis and translation and shows the optimization process of a bytecode program as an example of possible extensibility.

■ keyword : | CTOC | Optimization | Java Bytecode |

I. 서 론

자바 프로그램을 인터넷을 통해 내려 받은 경우에는, 대부분 소스 코드 없이 단지 바이너리 형태로된 바이트코드만을 얻게 된다[1][2]. 이러한 경우 내려 받은 파일이 내 컴퓨터에서 어떤 동작을 수행하는지 확인하기 위해 실행해보아 한다. 하지만 파일을 실행할 경우 실제 파일이 어떤 동작을 수행할지 모르기 때문에 상당히 위험한 결과를 초래할 수 있다. 따라서 바이트코드에 대해 수행 전에 정적 분석이 요구된다[3].

본 논문에서는 정적으로 자바 바이트코드를 처리하는 새로운 도구인 CTOC 프레임워크[4-7]를 구현하고 실험한다. 기존 연구들에서는 대부분의 경우 분석을 위해 역어셈블된 형태의 바이트코드를 생성하였다. 하지만 역어셈블 형태의 바이트코드는 표현이 명확하지 않고 단편적인 형식을 갖기 때문에 이해하기 어렵고 분석

접수번호 : #091112-002
접수일자 : 2009년 11월 12일
심사완료일 : 2010년 02월 12일
교신저자 : 김기태, e-mail : kimkitae@inha.ac.kr

이 복잡하다는 단점이 존재하였다. 이를 극복하기 위해 CTOC에서는 E-Tree를 제안한다. E-Tree는 표현식과 문장을 나타내기 위해 사용하는 중간 코드이름이다. E-Tree는 SSA Form(*Static Single Assignment Form*)으로 생성된다. E-Tree는 제어 흐름 그래프인 eCFG의 기본 블록에서 관련된 바이트코드를 묶어 트리 형태로 표현한 것이다.

CTOC는 4가지 구성 요소로 이루어진다. MBC, MeCFG, 그리고 MSSA는 eCFG와 E-Tree를 생성하는 역할을 수행하고, App는 eCFG와 E-Tree를 활용하는 역할을 수행한다.

CTOC에서는 바이트코드로부터 eCFG를 자동으로 생성하기 때문에 프로그램의 정적 제어 흐름 분석을 용이하게 한다. 또한 수행 과정 중 생성된 중간 결과물을 계속 eCFG와 E-Tree 형태로 유지하기 때문에 추후 분석과 코드에 대한 변환이 요구되는 경우라면 언제라도 필요한 기능을 추가하고 쉽게 확장할 수 있다는 특징을 가진다.

본 논문의 구성은 2장에서 관련 연구를 설명하고, 3장에서는 CTOC 프레임워크의 전체적인 구성과 각 구성 요소들에 대해 설명한다. 4장에서는 CTOC를 통해 생성되는 결과들에 대해 노드 수와 수행 시간 실험을 수행한다. 마지막으로 5장에서 결론을 맺는다.

II. 관련 연구

1. E-Tree

E-Tree는 CTOC에서 표현식과 문장을 나타내기 위해 사용하는 중간 코드이다. CTOC에서 E-Tree를 정의한 후 사용하는 이유는 다음과 같다. 첫째, E-Tree는 관련된 바이트코드를 묶어서 하나의 문장으로 표현하기 때문에 기존의 바이트코드를 읽는 것보다는 이해하기 쉽다. 둘째, E-Tree는 트리의 내부 정보를 확인할 수 있는 형태로 표현되기 때문에 필요한 정보를 쉽게 확인한다. 셋째, E-Tree는 바이트코드의 정보를 문장과 표현식으로 나누어 생성하기 때문에 코드 변환 시 특정 구문을 찾아 쉽게 처리한다. 넷째, 코드 변환 과정에서 E-Tree의 노드에 대해 트리 구조를 이용하여 해당 노드를 쉽고 간단하게 처리할 수 있기 때문이다.

2. eCFG

일반적인 제어 흐름 그래프에서는 시작(*entry*) 블록만을 사용하거나, 종료(*exit*) 블록을 추가한 후 사용한다[8][9]. 하지만 CTOC에서 사용하는 eCFG는 메소드에서 사용되는 매개 변수의 초기화와 타입 추론을 위해 초기화(*initial*) 블록이 추가된 형태를 사용한다. 초기화 블록에는 메소드의 형식 매개 변수들의 정보와 클래스 정보가 저장된다. 초기화 블록에 존재하는 형식 매개 변수들의 타입 정보는 상수 풀로부터 직접 얻어오기 때문에 항상 정확한 타입을 결정할 수 있다. 이를 바탕으로 나머지 노드들에 대한 타입 배정을 추후 수행한다.

3. SSA Form

CTOC에서는 존재하는 모든 변수를 정적으로 다루기 위해 정의(*def*)와 사용(*use*)에 따라 분리한다. 왜냐하면 동일한 변수라도 다른 위치에서 다른 값과 다른 타입을 가질 수 있기 때문이다. 일반적으로 정의와 사용에 대한 정보는 정의-사용 고리를 이용한다[8]. 하지만 CTOC에서는 정의-사용 고리 대신 SSA Form을 사용한다[5].

E-Tree는 노드 내부에 정의에 대한 위치 정보와 사용에 대한 리스트 정보를 유지하도록 하였기 때문에, 복잡하고 비용이 많이 발생하는 데이터흐름 분석 대신 SSA Form으로 변환하여 사용한다.

III. CTOC 프레임워크

[그림 1]과 같이 CTOC는 크게 바이트코드를 라벨 정보가 추가된 형태로 변환하는 MBC(*Modify ByteCode*), 확장된 제어 흐름 그래프인 eCFG를 작성하는 MeCFG(*Make extended CFG*), eCFG를 SSA Form으로 변환하는 과정을 수행하는 MSSA(*Make SSA*), 그리고 코드 변환 과정을 수행하는 App(*Applications*)로 구성된다.

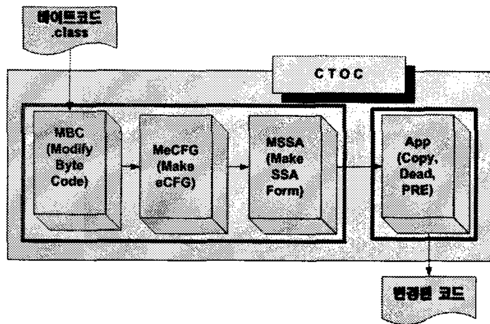


그림 1. CTOC의 overview

4가지 구성 요소에서 MBC, MeCFG, 그리고 MSSA는 eCFG와 E-Tree를 생성하는 역할을 수행하고, App는 eCFG와 E-Tree를 활용하는 역할을 수행한다.

1. MBC의 구조

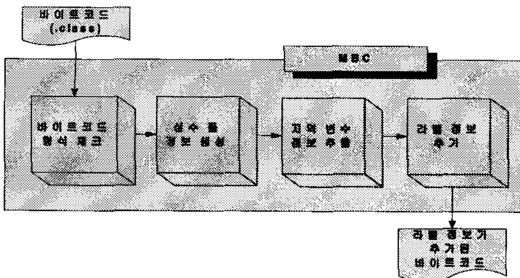


그림 2. MBC의 구조

CTOC의 첫 번째 구성 요소는 [그림 2]의 MBC이다. MBC는 바이트코드를 입력받아 우선 형식을 검사하고, 바이트코드로부터 상수 풀 정보를 추출하고, 바이트코드 내부에 존재하는 필드와 메소드 정보 그리고 생성된 상수 풀 정보를 이용하여 지역 변수에 대한 정보를 추출한다. 마지막으로 라인 번호와 관련된 정보들을 이용하여 기존의 바이트코드에 라벨 정보를 추가한 변형된 바이트코드를 생성한다.

MBC 과정을 통해 얻어진 정보는 [그림 3]과 같이 CTOC내에 존재하는 Bytecode_Viewer를 이용해 확인할 수 있다. Bytecode_Viewer는 이진 파일의 내용, 역어셈블된 바이트코드 내용, 상수 풀 내용 등을 직접 확인할 수 있는 도구이다. 왼쪽은 이진 파일의 내용을 핵사 코드로 변환하여 출력한 내용이고, 오른쪽은 바이트코드를 역어셈블한 형태로 출력한 결과이다.

[그림 4]는 추출된 상수 풀에 대한 정보로 Bytecode_Viewer 내에 있는 ConstantPool_Viewer를 통해 확인할 수 있다.

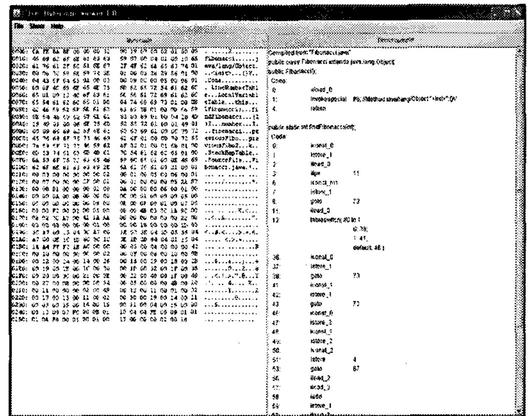


그림 3. CTOC의 바이트코드 뷰어

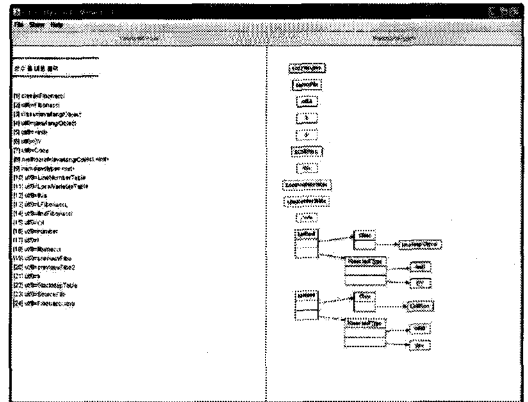


그림 4. 상수 풀에 대한 정보

ConstantPool_Viewer에서 왼쪽은 바이트코드로부터 추출한 상수 풀 정보이고, 오른쪽은 상수 풀 정보들의 연관 관계를 그래프로 나타낸 것이다. 이 결과를 통해 상수 풀 내에 존재하는 정보들 사이의 연관 관계를 한 눈에 쉽게 파악할 수 있다.

상수 풀 정보를 추출한 이후에는 변수와 관련된 정보를 추출하고, 기존의 바이트코드에 라벨 관련 정보를

추가하여 라벨 정보가 추가된 바이트코드를 생성하는 과정을 수행한다. 기존의 바이트코드가 들어있는 배열로 부터 정보를 추출하여 연결 리스트 형태로 명령어를 유지할 수 있도록 하고, 그 연결리스트에 라벨 정보를 삽입함으로써 라벨이 추가된 바이트코드가 생성된다.

```
foo.(ILA;LB;)I
L_0
        aload a$2
        getfield <Field LA;.f I>
        ldc 2

        ...
L_31
        aload b$3
        iload j$5
        putfield <Field LB;.f I>
L_37
        iload j$5
        ireturn
L_40
```

그림 5. 라벨 정보가 추가된 바이트코드

[그림 5]는 foo(int x, A a, B b) 메소드[11]에 대해 라벨 정보가 추가된 바이트코드 형태이다. [그림 5]와 [그림 3]의 바이트코드를 비교해 보면, [그림 5]에서는 MBC를 통해 코드의 시작 부분에 L_0과 같이 라벨 정보가 추가된 것을 확인할 수 있다. 이렇게 라벨 정보가 추가된 바이트코드와 MBC를 통해 수집된 여러 가지 관련 정보들은 다음 단계인 MeCFG의 입력이 된다.

2. MeCFG의 구조

MeCFG는 CTOC의 두 번째 구성 요소로, MBC가 생성한 라벨이 추가된 바이트코드로부터 확장된 제어 흐름 그래프인 eCFG를 생성하는 역할을 한다. MeCFG가 생성하는 eCFG와 E-Tree는 CTOC에서 분석과 코드 변환 수행을 위한 중간 표현으로 가장 중요한 정보들이다. MeCFG를 자세히 나타내면 [그림 6]과 같다.

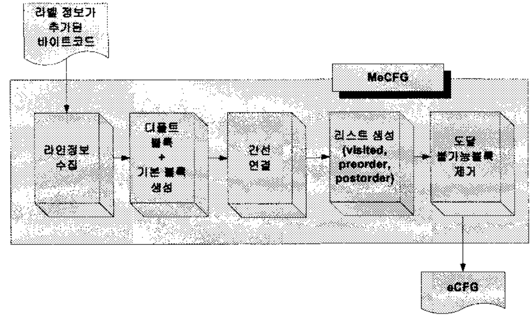


그림 6. MeCFG의 구조

[그림 6]과 같이 MeCFG는 변경된 바이트코드를 입력으로 사용하고, 최종적으로 eCFG를 생성하는 일련의 동작을 수행한다. 우선 MBC 과정에서 추가된 라인 관련 정보를 수집하고, 라인 정보를 바탕으로 디폴트 블록과 나머지 기본 블록들을 생성한다. 각 기본 블록들은 바이트코드 내에 있는 정보를 바탕으로 선행자와 후행자의 관계를 설정한 후 간선으로 연결하여 eCFG를 완성한다.

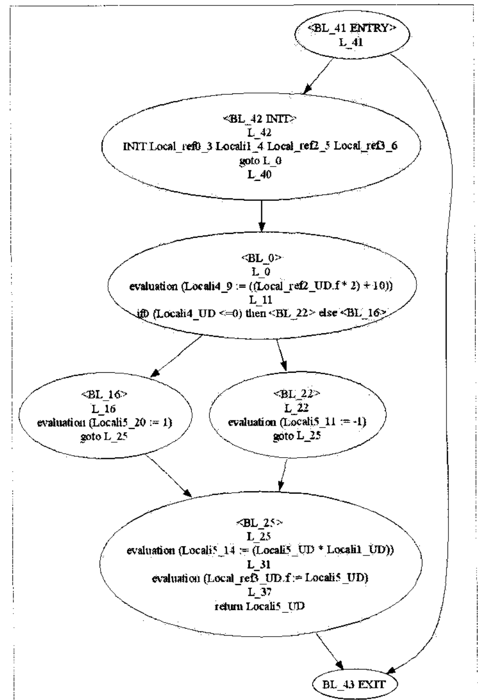


그림 7. 생성된 eCFG

[그림 7]은 MeCFG 과정에서 만들어진 eCFG이다. eCFG는 각 블록 내부에 문장을 표현하기 위해 E-Tree를 포함한다. CTOC에서 eCFG는 시작 블록 <ENTRY>, 종료 블록 <EXIT>, 그리고 초기화 블록 <INIT>를 갖도록 확장된 형태를 사용한다. 완성된 eCFG를 깊이 우선 순서(*depth first search*)로 방문하면서 이후 분석 과정과 코드 변환 과정에서 요구되는 방문(*visited*) 집합, 전위순서(*preoreder*) 리스트와 후위순서(*postorder*) 리스트 등 방문 순서에 관련된 각종 정보를 수집한다.

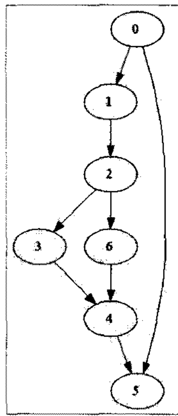


그림 8. DFS

[그림 8]은 깊이 우선 탐색 순서를 그래프로 출력한 것으로 전위순서 리스트와 동일한 결과를 보인다. MeCFG에서 생성된 [그림 7]의 eCFG와 [그림 8]의 깊이 우선 탐색 순서는 SSA From의 변환 과정에서 방문 순서에 대한 정보를 제공하기 위해 사용된다.

3. MSSA의 구조

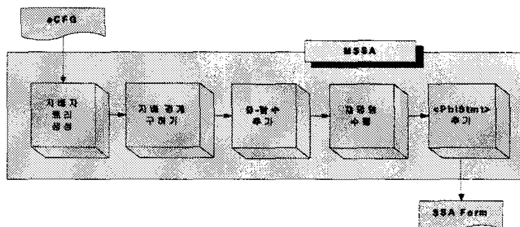


그림 9. MSSA의 구조

MSSA는 eCFG에서 SSA Form을 갖는 E-Tree를 생성하는 CTOC의 세 번째 구성 요소이다.

[그림 9]와 같이 MSSA의 입력으로는 이전 과정인 MeCFG의 결과물인 eCFG를 사용한다. 받아들인 eCFG로부터 SSA Form 형태를 갖는 E-Tree를 생성하기 위해서는 [그림 9]의 여러 단계를 거쳐야 한다. 우선 eCFG로부터 지배 관계 정보를 나타내는 지배자 트리를 생성한다. 지배자 트리는 ∅-함수를 삽입할 위치를 결정하기 위해 지배 경계를 계산할 때 eCFG와 함께 사용된다.

[그림 10]은 eCFG로부터 생성된 지배자 트리이다. SSA Form 변환 과정에서 ∅-함수를 삽입할 부분을 정하기 위해서는 eCFG와 지배자 트리를 이용하여 지배 경계를 계산한다. 지배 경계로 계산된 기본 블록은 ∅-함수가 삽입될 수 있는 위치이기 때문에, 이 블록에 ∅-함수가 추가된다. 이후 이름을 재명명하는 과정에서는 재명명 스택을 이용하여 새로운 정의가 발생하는 변수마다 새로운 이름을 배정한다. ∅-함수와 관련된 일련의 동작은, 기존의 eCFG의 기본 블록에 해당 E-Tree인 <PhiStmt>를 추가하여 변수들이 병합되는 지점에서 기존 변수에 대한 SSA Form을 완성한다.

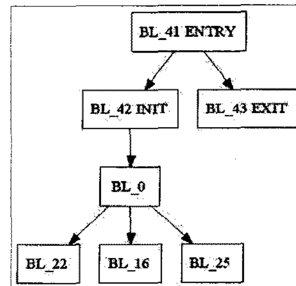


그림 10. 생성된 지배자 트리

[그림 11]은 완성된 SSA Form을 가진 eCFG이다. <BL_25> 블록을 보면 PHI로 표현된 <PhiStmt>가 추가된 것을 확인할 수 있다. <PhiStmt>가 추가된 위치인 <BL_25> 블록은 eCFG에서 병합되는 지점을 의미한다. 이러한 부분에서 정적으로 변수를 선택할 수 있도록 SSA Form으로 변경한 것이다.

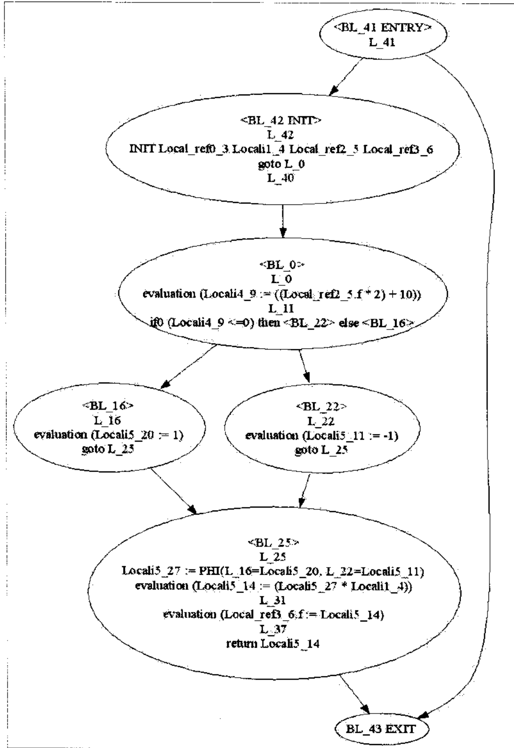


그림 11. 완성된 eCFG

4. App의 구조

SSA Form으로 변경된 eCFG를 생성한 후, 기본 블록 내에 존재하는 E-Tree를 활용하는 응용 분야로 코드 변환을 수행한다. 코드 변환 과정을 수행하는 CTOC의 네 번째 구성 요소인 App는 [그림 12]와 같이 구성된다.

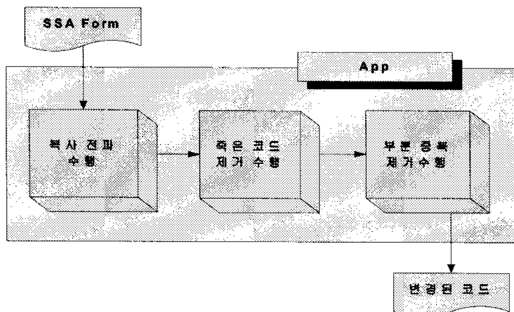


그림 12. App의 구조

[그림 12]의 App는 SSA Form으로 변환된 eCFG를 입력으로 사용한다. 그리고 App는 eCFG와 E-Tree를 이용하여 복사 전파, 죽은 코드 제거, 부분 중복 제거 등의 코드 변환을 필요에 따라 수행한다.

CTOC에서 eCFG와 E-Tree에 대한 코드 변환은 새로운 중간 표현의 적용 가능성을 보이기 위해 이 논문에서는 앞에서 설명한 세 가지 코드 변환만 수행하였지만, 필요에 따라 추후 더 많은 코드 변환 과정을 추가하여 기능을 확장할 수 있다. 왜냐하면 App에서 구현된 코드 변환들은 SSA Form을 가진 eCFG를 입력으로 받아들여 코드 변환을 수행하고, 내부적으로 E-Tree를 이용하기 때문이다. 물론 코드 변환의 결과들 역시 변경된 E-Tree와 eCFG로 생성된다. 따라서 추후에 추가되는 새로운 코드 변환 기법들도 eCFG와 관련된 메소드를 통해서 확장이 가능하다.

IV. 실험

실험은 Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz, 메모리 2GB를 가진 PC에서 수행하였으며, CTOC 작성과 테스트를 위해 eclipse 3.4를 사용하였고, 바이트코드 출력을 위해 editplus 3.10 버전을 사용하였다. 자바 컴파일러는 jdk1.6.0_09를 사용하였다.

예제 프로그램은 실험 결과의 비교를 위해 제어 흐름을 살펴볼 수 있는 6가지 경우를 분석하였다. 이 데이터들은 실험 결과의 비교를 위해 Don Lance의 논문에서 사용한 예제를 이용하였다[10]. [표 1]은 실험에 사용될 프로그램에 대한 간단한 설명이다.

1. 노드 수

[표 1]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 2]는 eCFG로 변환한 후 생성된 정보에 대한 결과이다.

[표 2]에서 바이트코드는 javap -c를 이용하여 생성된 바이트코드의 라인 수를 의미한다. 변경후는 CTOC

를 통해 기본 블록을 생성하기 위해 기존의 코드를 변경하는 과정에서 추가되거나 삭제된 후의 코드 라인 수를 의미한다. 결과를 보면 바이트코드에 라벨 정보가 추가되기 때문에 기존의 소스보다 길이가 늘어나고 역어셈블된 바이트코드보다는 길이가 줄어드는 것을 확인할 수 있다. 기본 블록은 변경된 코드와 기본 블록을 위한 리더를 통해 생성된 기본 블록의 수를 의미한다. 간선은 기본 블록과 다른 기본 블록 사이의 관계를 표현하기 위해 사용된 간선의 수를 의미한다. 마지막으로 노드는 기본 블록 내에 명령어와 문장을 인식하기 위해 사용된 노드의 개수를 의미한다.

[표 3]은 정적 단일 배정 형태로 변환 후에 제어 흐름 그래프와 비교한 결과이다. 정적 단일 배정 형태로 변환 된 이후 전반적인 라인수와 노드의 개수가 증가한 것을 볼 수 있다. 이는 기존의 제어 흐름 그래프에 존재하지 않았던 Ø-함수의 추가에 의해서 발생하는 것이다.

표 1. 사용 예제와 간단한 설명

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

표 2. eCFG 실험 결과

	바이트코드	변경후	기본 블록	간선	노드
SquareRoot	94	60	15	18	99
SumOfSquareRoot	103	63	18	19	108
Fibonacci	76	69	18	22	86
BubbleSort	79	68	16	21	101
LableExample	51	59	13	16	58
Exceptional	99	149	26	29	143

표 3. 정적 단일 배정 형태 변환 후 결과

	CFG lines	SSA lines	%	CFG node	SSA node	%
SquareRoot	60	63	4.76	99	117	15.38
SumOfSquareRoot	63	71	11.27	108	143	24.48
Fibonacci	69	77	10.39	86	126	31.75
BubbleSort	68	76	10.53	101	133	24.06
LableExample	59	63	6.35	58	74	21.62
Exceptional	149	177	15.82	143	304	52.96

표 4. 프로그램 수행 시간 측정 (단위 : ms)

프로그램	바이트코드	기본블록	노드	시간
SquareRoot	94	15	99	188
SumOfSquareRoot	103	18	108	203
Fibonacci	76	18	86	202
BubbleSort	79	16	101	281
LableExample	51	13	58	171
Exceptional	99	26	143	261

표 5. 클래스 별 수행 시간 측정 (단위 : ms)

클래스	메소드	최소	최대	전체수행
SquareRoot	2	31	63	188
Fibonacci	2	31	78	202
CFG	73	0	875	2344
Tree	180	0	218	2765
Block	23	0	78	406
SSA	10	0	62	437

2. 수행 시간

[표 4]는 [표 1]에서 설명한 각 프로그램에 대한 수행 시간을 측정한 결과이다. 시간 측정을 위해 자바의 System 클래스에서 제공되는 currentTimeMillis() 메소드를 사용한다. 이 메소드는 현재 시스템 시간에 대한 long형의 정수 값을 제공한다. 각 프로그램의 시작과 끝 부분에서 시간을 가져와 각 프로그램이 수행하는데 걸리는 시간을 측정한다.

결과를 보면, 바이트코드의 크기가 크고 기본 블록의 수가 많을수록 시간이 많이 걸리는 것을 확인할 수 있다. 하지만 [표 4]에서 측정한 프로그램들은 크기가 너

무 작아, CTOC를 개발하면서 사용한 클래스들에 대해 [표 5]와 같은 실험을 수행한다.

[표 5]의 실험은 각 클래스 내부에 존재하는 메소드 개수, 가장 시간이 많이 걸리는 메소드와 가장 시간이 적게 걸리는 메소드, 그리고 전체 수행 시간을 측정한 결과이다. 실험 결과, 대체적으로 파일 사이즈가 클수록, 내부에 있는 메소드가 클수록 시간이 많이 걸리는 것을 확인할 수 있다. 사용한 클래스 중 CFG와 Tree의 최대 부분을 보면, 파일 내에서 가장 시간이 많이 걸리는 메소드는 CFG에 존재하며 875(ms)의 수행 시간을 갖지만, 전체 수행 시간은 2765(ms)로 오히려 Tree가 큰 것을 확인할 수 있다. 이것은 Tree 내에 존재하는 메소드 처리에 평균 수행 시간이 더 길었기 때문에 나타난 결과이다. 최소에 해당하는 데이터 중 0(ms) 값이 측정되는 경우가 발생하는데, 이는 실제 수행은 발생하지만 윈도우 환경에서 수행하기 때문에 나타난 결과 값이다. 최소에 해당하는 메소드의 대부분은 아주 짧은 경우이기 때문에 측정 범위를 벗어난 결과가 출력된 것이다. 보통의 경우 CTOC에서 분석하는 클래스 파일 내에 있는 메소드들은 길이가 길지 않은 경우가 많아 실제 각 구성 요소 별 수행 시간을 정확히 측정하는 것은 어렵다. 따라서 CTOC 도구의 정확성과 각 구성 요소의 수행 시간을 측정하기 위해서는 메소드의 크기와 내용을 바꿔가면서 각 구성 요소들이 수행하는 데 걸리는 시간을 측정하여야 한다.

```
public class Temp {
    void test(){
        int sum, mul;
        sum = 0; mul = 0;
        for(int i=0; i<10; i++){
            sum = sum + i;
            mul = mul * i;
}}}
```

그림 13. for 문을 포함한 메소드

[그림 13]은 for 문을 포함한 메소드이다. for 문을 임의의 개수만큼 복사하여 실험을 수행하였다.

표 6. 바이트코드 크기에 따른 수행 시간 측정 (단위 : ms)

for 문장	1	5	10	25	50	100
파일 사이즈	425B	625B	875B	1.58KB	2.80KB	5.24KB
바이트코드	33	93	168	393	768	1518
메소드수행시간	31	78	141	344	1032	3279
전체수행시간	156	219	266	469	1157	3374

수행한 실험 방법과 내용은 [표 6]과 같다. [표 6]의 항목에 존재하는 1, 5, ... , 100은 메소드 안에 존재하는 for 문을 복사한 개수이다. 일반적으로 메소드 안에는 아주 많은 양의 코드가 존재하는 경우가 드물다. 따라서 for 문을 필요한 만큼 복사하여 복사 개수에 따라 변경된 파일 사이즈와 바이트코드 라인 수를 측정하고, for 문이 증가한 해당 메소드를 분석하고 코드 변환을 수행하는 데 걸리는 시간을 측정하기로 한다. 마지막으로 파일 전체를 수행 하는 데 걸리는 시간을 측정한다.

실험결과, for 문의 개수가 증가함에 따라 파일 사이즈, 바이트코드의 라인 수가 비례하여 증가하며, 내용이 증가된 메소드를 처리하는 데 대부분의 시간이 소비되는 것을 확인할 수 있다. 프로그램을 수행하는 데 걸리는 시간도 1518 개의 바이트코드 명령어를 처리하는 데 3279(ms) 정도의 시간인 것을 확인 할 수 있다. 일반적으로 한 메소드에 1500개 이상이 바이트코드가 사용되는 경우는 드물다. 따라서 일반적인 경우인 30~150개의 바이트코드 명령어의 경우를 살펴보면, 31 ~ 141(ms) 정도의 빠른 시간 내에 처리된다는 것이 확인된다.

V. 결론

자바 바이트코드는 스택 기반 코드이기 때문에 수행 속도가 느리고 프로그램 분석이나 최적화에 적절한 표현이 아니라는 단점이 존재한다. 또한 바이너리 형태이기 때문에 효과적으로 분석하고 변환하기 위해서는 적절한 형태로 변환이 요구되었다. 이를 위해 CTOC를 구현하였다.

CTOC는 크게 바이트코드를 라벨 정보가 추가된 형태로 변환하는 MBC, 확장된 제어 흐름 그래프인 eCFG를 작성하는 MeCFG, eCFG를 SSA Form으로 변환하는 과정을 수행하는 MSSA, 그리고 코드 변환 과정을 수행하는 App 이렇게 4가지 구성 요소로 구성된다.

CTOC에서 E-Tree는 표현식과 문장을 나타내기 위해 사용하는 중간 코드로 정적 분석을 위해 SSA Form으로 생성하였다. E-Tree는 기본 블록 내에서 관련된 바이트코드 명령어를 묶어 트리 형태로 표현하였다.

CTOC에서 생성된 SSA Form을 가진 eCFG를 이용해 프로그램의 정적 제어 흐름 분석을 용이하게 수행할 수 있었다. 또한 수행 과정 중 생성된 중간 결과물을 계속 eCFG와 E-Tree 형태로 유지하여 추후 분석과 코드 변환이 요구되는 경우 언제라도 필요한 기능을 추가하고 확장할 수 있도록 하였다. 또한 CTOC는 보통 한 메소드내에서 사용되는 30~150개의 명령어인 경우 31 ~ 141(ms) 정도의 빠른 시간 내에 처리되는 것을 실험을 통해 확인하여 도구로서의 가능성을 보였다.

참 고 문 헌

[1] T. Linholm and F. Yellin, *The Java Virtual Machine Specification*, Addison Wesley Pub., 1997.

[2] J. Gosling, B. Joy, and G. Steel, *The Java Language Specification*, Addison Wesley Pub., 1997.

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

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

[5] 김기태, 유원희, "CTOC에서 자바 바이트코드를 위한 정적 단일 배정 형태", 정보처리학회논문지 D, 제13-D권, 제7호, pp.939-946, 2006.

[6] 김기태, 김제민, 유원희, "CTOC에서 죽은 코드 제거 구현", 한국컴퓨터정보학회논문지, 제12권, 제2호, pp.1-8, 2007.

[7] 김기태, 김제민, 유원희, "CTOC에서 루프 벗기기 구현", 한국컴퓨터정보학회논문지, 제13권, 제5호, pp.27-35, 2008.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques and Tools(2nd Edition)*, Addison-Wesley Longman Publishing Co., 2006.

[9] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Pub., 1997.

[10] D. Lance, "Java Program Analysis: A New Approach Using Java Virtual Machine Bytecodes," Master's. Thesis, Middle Tennessee State University, 1997.

[11] G. Bian and K. Nakayama, "Java bytecode dependence analysis for secure information flow," International Journal of Network Security, Vol.4, No.1, pp.59-68, 2007.

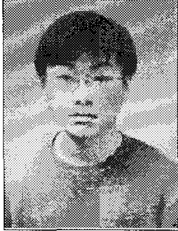
저 자 소 개

김 기 태(Ki-Tae Kim) **정회원**

- 1999년 2월 : 상지대학교 전산학과(이학사)
- 2001년 2월 : 인하대학교 전자계산공학과(공학석사)
- 2008년 2월 : 인하대학교 정보공학과(공학박사)
- 2008년 ~ 현재 : 인하대학교 컴퓨터정보공학부 강의전임강사

<관심분야> : 컴파일러, 프로그래밍 언어

김 제 민(Je-Min Kim) 정회원



- 2006년 2월 : 인하대학교 컴퓨터 공학부(공학사)
- 2008년 2월 : 인하대학교 정보공학과(공학석사)
- 2008년 3월 ~ 현재 : 인하대학교 정보공학과 박사과정

<관심분야> : 프로그래밍 언어, 컴파일러

유 원 희(Weon-Hee Yoo) 정회원

- 1975년 2월 : 서울대학교 응용수학과(이학사)
- 1978년 2월 : 서울대학교 대학원 계산학(이학석사)
- 1985년 2월 : 서울대학교 대학원 계산학(이학박사)
- 1979년 ~ 현재 : 인하대학교 컴퓨터 공학부 교수

<관심분야> : 컴파일러, 프로그래밍 언어, 실시간 시스템, 병렬시스템