제 27 회 한국정보처리학회 춘계학술발표대회 논문집 제 14 권 제 1 호 (2007. 5)

CTOC 에서 배열을 위한 SSA Form 의 구현

김제민*, 김기태*, 유원희*
*인하대학교 컴퓨터 정보공학과
e-mail : jeminya@gmail.com

Implementation of SSA Form for Array in CTOC

Je-Min Kim*, Ki-Tae Kim*, Weon-Hee Yoo*
*Dept. of Computer Science & Information Engineering, Inha University

요 약

자바 바이트코드는 인터넷에서 바로 내려 받아 사용할 수 있는 형태로서 많이 사용된다. 따라서 그것에 대한 최적화가 중요하다. 따라서 이를 효율적으로 수행하기 위한 CTOC 프레임워크를 구현하였다. CTOC 에서 자바 바이트코드에 대한 스칼라 변수에 대한 최적화는 이미 많이 존재하지만, 배열에 대한 최적화는 그렇지 않다.

이 논문에서는 기존의 CTOC 에서의 배열 최적화를 위한 SSA form 에 대한 적용과 구현을 위해 기존에 있는 Array SSA form 과 Region Array SSA 의 사용을 보다 간단하게 하는 방법을 제안한다.

1. 서론

자바의 클래스 파일은 인터넷에서 많이 다운로드 되어 실행되고 사용되기 때문에 클래스 파일의 최적화는 중요하다. 따라서 CTOC(Classes To Optimized Classes)를 구현하여 자바 클래스 파일, 즉 바이트코드에 대한 최적화와 분석을 수행하였다.[3]

바이트코드는 스택 머신 기반 코드이므로 분석과 최적화가 어려우므로 트리 구조 코드로 바꾸어야 한다. [5]는 이렇게 바꾼 트리 구조 코드를 SSA Form 으로 바꾸었다. SSA Form 은 변수의 정의와 사용간의 관계를 명백히 나타낼 수 있어서 데이터 흐름 분석과 최적화에 많이 이용된다. 하지만 [5]에서는 배열에 대한 SSA Form 은 다루지 않는다.

일반적으로 배열을 SSA Form 으로 다룰 때는 배열이 새로 정의될 때마다 그 전에 정의된 배열의 모든 요소들을 죽이고 새롭게 정의한다. 하지만 프로그램에서 배열은 최적화와 병렬성을 위해서 각각의 요소간의 정의와 사용을 명백하게 아는 것이 좋다.

하지만 기존의 방법들은 배열의 요소 단위의 정의와 사용에 대한 관계를 명확하고 효율적으로 알 수 있다 하지만 구현이 어렵다는 단점이 있다. 따라서 이 논문에서는 좀 더 쉬운 구현 방법을 제시한다.

이 논문의 2 장에서는 SSA Form 에서의 배열에 대한 기존 구현들에 대해 설명하고, 3 장에서는 바이트코드에서 배열을 위한 SSA Form 을 구현할 때 고려해야 할 사항을 설명하고 4 장에서는 이를 구현하는 예를 직접 보여준다. 마지막으로 5 장에서는 결론을 제시한다.

2. SSA Form 에서의 배열에 관한 구현들

SSA form 에서 배열을 다루기 위한 방법으로는 첫째, 배열이 정의될 때마다 배열의 모든 요소들을 재정의하는 방법과, 둘째, 배열의 각 요소가 정의될 때마다 정의에 대한 정보를 유지하는 방법이 있다.

첫 번째 방법은 배열 변수가 정의될 때든지 배열 변수의 특정 요소만 정의될 때든 지에 상관없이 배열의 모든 요소를 재정의 한다. 예를 들어 (그림 1)에서 자바 소스 코드는 3 주소 형태의 SSA Form 으로 변환될 수 있는데 이 같은 경우 배열의 특정 요소의 값이 어디서 정의되었는지에 대한 정보를 얻을 수 없다. 즉 B_1 에 배정될 때 A_1 에서 온 값인지 A_2 에서 온 값인지 알 수 없다.

자바 소스 코드	SSA Form
A[0] = 0; A[1] = 1; B = A[0];	A_1[0] = 0 A_2[1] = 1; B_1 = A_2[0];

(그림 1) 자바 소스 코드와 SSA Form

배열에서 요소 단위로 정의 사용을 알기 위한 SSA Form 의 구현을 위한 노력으로는 Array SSA Form[9]이 있다. Array SSA Form 에서는 배열이 정의될 때 최초 배열을 제외하고는 배열이 정의될 때마다 Φ 함수를 위치시켜야 한다는 Define Φ 규칙과, 구별되는 제어 경로에서 온 정의들을 병합해야 한다는 Merge Φ 규칙을 이용해 SSA Form 을 완성하고, @array 를 이용해 배열의 특정 요소가 정의된 최근에 정의된 시간을 알 수 있다. 따라서 배열의 요소 단위로 정의와 사용간의 관계를 명확히 파악할 수 있다.

다른 방법으로 Region Array SSA[10]가 있다. Region 은 정의되거나 사용되는 배열의 요소를 나타낸다. 인터벌의 표현식을 공식화할 수 있는 USR(Uniform Set Reference)을 이용해 배열의 특정 요소의 배열의 정의와 사용을 표현하고 그들의 관계를 쉽게 알아낼 수 있는 Region 을 표시할 수 있다.

3. CTOC 에서 배열 SSA Form 을 위한 고려사항

3.1 배열의 요소 표현

배열의 요소들을 표현하기 위해서 (그림 2)과 같은 [i:j]와 같은 표현을 사용한다. 또한 인터벌간의 교집합, 합집합을 나타내는 기호 ∪, ∩ 또한 사용한다.

```
A[1:10] = 3;
B[3:3]  = 4;
C[1:10 ∪3:3] = Φ(A,[1:10],B,[3:3]);
```

(그림 2) 배열의 요소 표현

(그림 2)에서 A[1:10]은 A 배열에서 인덱스가 1 인 요소부터 인덱스가 10 인 요소를 의미하고 [1:10∪3:3]은 인터벌[1:10]과 [3:3]의 합집합 인터벌을 의미한다.

3.2 Φ 함수의 위치

일반적인 SSA Form 에서는 제어 흐름이 병합되는 부분에 Φ 함수를 위치시켜야 하지만 배열을 위한 SSA Form 에서는 조금 더 복잡하다.

두 가지 경우로 Φ 함수를 구분할 수 있다. 서로 다른 정의들을 병합하는 경우와 루프에서 정의를 병합하는 경우이다. 루프에서 정의를 병합하는 경우는 루프가 시작되기 전의 정의와 루프를 수행하면서 병합되는 정의에 대한 구분이 필요하다.

서로 다른 정의를 병합하는 가장 간단한 경우는 우선 배열의 요소가 정의될 때 마다 두 배열의 정의를 병합하는 경우이다. (그림 3)에서 배열 $A_1[3]$이 정의되고 $A_2[4]$가 정의되었으므로 $A_3[3:4]$에서는 $A_1[3]$과 $A_2[4]$를 병합해 주어야 한다.

```
A1[3:3] = 3;
A2[4:4] = 4;
A3[3:4] = Φ(A1,[3:3],A2,[4:4])
```

(그림 3) 간단한 배열의 요소 정의

여기서 Φ 함수는 A_1 의 인터벌과 A_3 의 인터벌 중에 중복되는 인터벌에 해당하는 A_1 의 인터벌의 값들을 A_3 의 인터벌에 할당해 준 다음, A_2 의 인터벌과 A_3 의 인터벌 중 중복되는 A_2 의 인터벌에 해당하는 값들을 A_3 의 인터벌에 할당하라는 의미를 가진다.

또 다른 경우는 If 블록과 for 블록이 끝난 후 병합하는 경우가 있다. 예를 들어 (그림 4)는 if 블록이 끝난 후 병합하는 경우이다. 여기서는 Φ 함수의 의미가 다르기 때문에 다른 기호 γ 를 사용한다. 여기서는 if 절을 수행할 수도 있고 안 할 수도 있기 때문에 γ 함수는 If 문을 수행할 경우 A_3 와 A_4 를 병합하고 수행하지 않을 경우 A_3 의 값을 A_5 에 할당하라는 의미를 갖게 된다.

```
A3[3:4] = Φ(A1,[3:3],A2,[4:4])
  if( i>0)
    A4[i:i] = I;
A5[i∪3:4] = γ(A3,[3:4],A4,[i:i]);
```

(그림 4) if 문이 끝난 후 두 정의의 병합

루프에서 정의를 병합하는 경우는 (그림 5)와 같이 루프가 시작될 때 루프 밖에서 온 정의와 루프 내에서 반복이 수행되면서 바뀌는 정의를 병합해 주어야 하기 때문에 Φ 함수를 루프가 시작되는 지점에 위치시켜야 한다. 여기서의 Φ 함수도 일반적인 Φ 함수와 의미가 다르므로 다른 기호 μ 를 사용한다. μ 함수는 for 루프가 시작하기 전의 정의와 루프 안에서 정의된 것들 병합한다.

```
A 1[3:3] = 3;
for(int i=0;i<10;i++){
    A2[3:3∪i:i] = μ(A1,[3:3],A4,[ i:i∪3:3])
    A3[i:i] = 3;
    A4[i:i∪3:3] = Φ(A2,[3:3∪i:i],A3,[i:i]);
}
```

(그림 5) Loop 문에서의 정의들의 병합

4. 구현

배열을 위한 SSA Form 을 구현하기 위해서는 바이트코드로 이루어진 CFG(제어흐름그래프)를 작성한 후 BNF 를 이용해 트리 구조 코드로 바꾼 후 배열 정의를 병합하기 위한 적절한 Φ 함수를 위치 시켜야 한다.

4.1 CFG 작성.

CTOC 에서는 (그림 7)와 같은 바이트 코드를 입력으로 해서 변환 및 최적화를 수행한다. (그림 6)는 (그림 7)의 바이트코드로 컴파일 되기 전 소스코드이다.
(그림 7)와 같은 바이트코드를 가지고 CFG 를 작성한다. 바이트코드에서 리더를 알아내어 리더부터 다음 리더까지를 기본 블록으로 하여 잘라낸 후 기본 블록들을 제어 흐름에 따라 연결해 주면 된다. 여기서 리더란 (1) 맨 첫 문장, (2) 조건 또는 비조건 goto 의 목

표가 되는 문장 (3) goto 나 조건 goto 의 다음 문장이 리더가 된다[7]

```
public class Array {
        public static void main(String[] args) {
                int x[] = new int[10];
                int j;
                for(int i=0;i<x.length;i++){
                                x[i] = i;
                }
                j = x[0];
        }
}
```

(그림 6)그림 8 이 컴파일 되기 전 java 파일

```
Code:
    0:   bipush   10
    2:   newarray int
    4:   astore_1
    5:   iconst_0
    6:   istore_3
    7:   goto      17
   10:   aload_1
   11:   iload_3
   12:   iload_3
   13:   iastore
   14:   iinc      3, 1
   17:   iload_3
   18:   aload_1
   19:   arraylength
   20:   if_icmplt          10
   23:   aload_1
   24:   iconst_0
   25:   iaload
   26:   istore_2
   27:   return
```

(그림 7)그림 7 의 자바 바이트 코드

4.2 트리 구조 코드로 변환

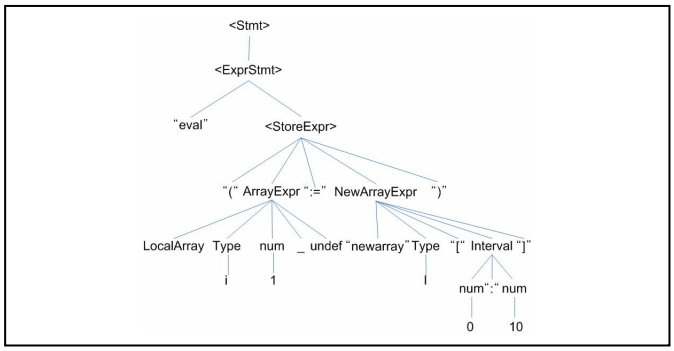
현재 CFG 는 바이트코드로 작성이 되어있으므로 기본 블록에 있는 문장들을 트리 구조로 바꾸기 위해 트리 형태로 만든다. 이를 위해 (그림 8)과 같은 BNF 를 정의하였다[3]. 여기서는 배열에 관한 부분을 좀더 자세히 살펴본다.

```
Stmt → ExprStmt | InitStmt | JumpStmt | LabelStmt | PHIStmt
LabelStmt → Label
ExprStmt → eval Expr
JumpStmt → GotoStmt | IfStmt | ReturnExprStmt
IfStmt → IfZeroStmt
GotoStmt → goto Block
IfZeroStmt → if0 ( Expr (== |!= |> |>= |< |<= ) (null |0 ) ) then
Block else Block
ReturnExprStmt → return Exp
Block → <block Label>
Label → label_ Num
Expression → ConstantExpression| DefExpr | StoreExpression
| ArithExpression | NewArrayExpr
DefExpr → MemExp
StoreExpression → ( MemExpr := Expression )
MemExpr → VarExpr
ConstantExpression → ' ID ' | Num
NewArrayExpr → newarray Type [Interval]
Interval → Num : Num (∪ Interval | ∩ Interval)
VarExpr → LocalExpr | StackExpr | ArrayExpr
ArrayExpr → LocalArray Type Num _Num[Interval]
```

(그림 8) BNF 의 일부

(그림 8)의 BNF 는 원래의 BNF 의 일부이다. BNF 에서 볼드체로 된 부분은 배열을 다루기 위해 확장된 부분이다. 배열 변수의 정의, 배열 요소별 정의 그리고 배열의 사용을 위해 BNF 를 확장하였다.

(그림 7)에서 각각의 바이트 코드 앞에 붙은 번호는 라벨을 의미한다. 라벨 0 부터 4 까지는 배열 변수를 정의하는 부분이다. (그림 8)의 BNF 를 이용해서 (그림 9)와 같은 트리를 만들 수가 있다.



(그림 9) 트리

라벨 10 에서 13 까지는 배열의 요소를 정의하는 부분이고 라벨 23 에서 26 까지가 배열의 특정 요소에서 값을 얻어 특정 변수에 배정하는 부분이다.

이렇게 확장된 BNF 를 가지고 트리 구조 코드의 CFG 를 만들게 되면 (그림 10)와 같다.

```
<BL_0>
L_0
      eval(LocalArrayi1_undef := newarray I[0:10])
L_5
      eval(Locali3_3 := 0)
      goto L_17

<BL_10>
L_10
   eval(LocalArrayi1_1[Locali3_19:Locali3_19] := Locali3_19)
L_14
      eval(Locali3_11 := (Locali3_19 + 1))
      goto L_17

<BL_17>
L_17
      Locali3_19 := PHI(L_0=Locali3_3, L_10=Locali3_11)
      if (Locali3_19 < LocalLoc) then <BL_10> else <BL_23>

<BL_23>
L_23
      eval(Locali2_14 := LocalArrayi1_1[0:0])
L_27
      return
```

(그림 10) BNF 를 이용한 CFG

4.3 Φ 노드 위치 시키기

3.2 절에서 논의된 데로 Φ-함수들을 위치시켜야 한다. 이를 위해서는 작성된 CFG 를 가지고 배열의 요소가 정의되는 두 부분이 나올 때 마다 Φ 함수를 위치시키고 루프의 헤더를 찾아서 헤더 전의 오는 정의와 루프내의 정의들을 루프의 헤더에서 μ 함수를 이용해 병합해 주면 된다. if 문의 다음 문에서 γ 를 위치시켜 주기 위해서는 CFG 의 기본 블록 중 들어 오는 간선이 순차 간선인 경우 γ 함수를 위치시켜주면 된다. 이러한 결과가 (그림 11)에 나와있다.

```
<BL_0>
L_0
      eval(LocalArrayi1_undef := newarray I[0:10])
L_5
      eval(Locali3_3 := 0)
      goto L_17

<BL_10>
L_10
      LocalArrayi1_1:=
μ(LocalArrayi1_undef,LocalArrayi1_2,[Locali3_19:Locali3_19])
  eval(LocalArrayi1_2[Locali3_19:Locali3_19] := Locali3_19)
L_14
      eval(Locali3_11 := (Locali3_19 + 1))
      goto L_17

<BL_17>
L_17
      Locali3_19 := PHI(L_0=Locali3_3, L_10=Locali3_11)
      if (Locali3_19 < LocalLoc) then <BL_10> else <BL_23>

<BL_23>
L_23
      eval(Locali2_14 := LocalArrayi1_1[0:0])
L_27
      return
```

(그림 11) 완성된 SSA Form 의 CFG

5. 결론

기존에 스칼라 변수에 대한 SSA Form 은 많이 존재하지만 배열 단위의 최적화와 병렬성을 적용하기 위해서는 배열의 요소 단위로 정의 사용을 명확히 아는 것이 중요하다. 기존의 배열의 요소 단위의 SSA Form 은 구현이 복잡하기 때문에 CTOC 에서 배열의 요소 단위를 용이하게 할 수 있도록 배열을 위한 SSA Form 을 구현하였다. 인터벌을 사용하여 구현이 편리해졌지만 정의 사용이 명확하게 되지 않는 부분이 발생하였다.

이러한 단점을 고치기 위해 단지 배열의 요소 인덱스만을 가리킬 수 있는 인터벌보다는 제어흐름에 대한 정보도 가지고 있는 자료구조를 고안해 내면 배열의 요소단위의 정의 사용의 관계를 명확하게 할 수 있을 것으로 보인다.

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, “CTOC 에서 자바 바이트코드를 이용한 제어 흐름 분석에 관한 연구”, 한국콘텐츠학회 논문지 제 6 권 제 1 호, pp. 160-169, 2006(1)

[4] 김기태, 유원희, “정적 단일 배정 형태를 위한 정적 타입 배정에 관한 연구”, 한국콘텐츠학회 논문지 제 6 권 제 2 호, pp. 117-126, 2006(2)

[5] 김경수, 유원희, “바이트코드 분석을 위한 중간코드에 관한 연구”,한국 컴퓨터 정보학회 논문지 제 11 권 제 1 호,pp 107-117,2006(3)

[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, pp. 437-477, 1998

[8] Don Lance, "Java Program Analysis: A New Approach Using Java Virtual Machine Bytecodes", http://www.mtsu.edu/~java

[9] Kathleen Knobe, Vivek Sarkar, “Array SSA form and its use in Parallelization”, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language, January 1998

[10] Silvius Rus, Guobin He, Christophe Alias, Lawrence Rauchwerger, “Region Array SSA”, In ACM PACT, September 2006