대소관계 그래프를 이용한 Just-in-Time 컴파일 환경에서의 배열 경계 검사 제거

(Array Bounds Check Elimination using Ineguality Graph in Java Just-in-Time Compiler)

최 선 일 † 문 수 묵 ††
(Sun-il Choi) (Soo-mook Moon)

요 약 Just-in-Time 컴파일러를 이용하여 자바의 성능을 향상시키려면 여러 문제들을 극복하여야 한다. 이 문제 중 중요한 부분을 차지하는 것이 배열경계 검사(Array bounds check) 명령어를 어떻게 최적화하느냐는 것이다. 정적인 컴파일 환경의 경우에는 이미 많은 연구가 진행되어 매우 강력한 성능을 가지는 알고리즘이 알려져 있으나 컴파일 시간이 수행시간의 일부인 Just-in-Time 컴파일 환경에 이를 적용하기에는 컴파일에 시간을 너무 많이들이는 결과를 낳아 적절하지 않다. 현재 Just-in-Time 컴파일러들은 가볍고 단순한 반면에 중복된 배열 경계 검사를 찾아내는 능력이 다소 부족한 알고리즘을 사용하거나 아니면 강력하지만 정적 단일 배정(Static Single Assignment) 형태로 명령어 표현 방식을 변환해 주어야만 하는 알고리즘을 사용하고 있다. 정적 단일 배정 형태로의 변환 및 되돌림은 가볍고 빠른 컴파일러를 지향하는 Just-in-Time 컴파일러에 부합되지 않는다. 본 논문은 변수 간의 대소관계를 표현하는 그래프를 배열 경계 검사 알고리즘에 적용하는 것을 통해 충분한 성능을 내면서도 정적 단일 배정 형태로의 변환을 필요로 하지 않는 알고리즘을 제안한다.

Java에서의 모든 최적화는 Java 언어 명세에서 강제하고 있는 정확한 예외 처리(precise exception) 규칙을 따라야 한다. 명령어의 위치를 바꿈으로써 성능 향상을 꾀하는 최적화의 경우 이것으로 인해 많은 제약을 받게 되는데, 배열 경계 검사 최적화(Array bounds check elimination optimization) 또한 이 규칙에 의해 많은 최적화 기회를 잃는다. 우리는 이 제약을 극복하여 배열 경계 검사최적화의 적용 범위를 넓힐 수 있는 새로운 방법도 아울러 제안하고자 한다.

키워드 : 자바, 자바 가상 머신, Just-in-Time 컴파일러, 배열 경계 검사 최적화

Abstract One of the problems in boosting Java performance using a Just-in-Time (JIT) compiler is removing redundant array bound checks. In conventional static compilers, many powerful algorithms have been developed, yet they are not directly applicable to JIT compilation where the compilation time is part of the whole running time. In the current JIT compilers, we can use either a naive algorithm that is not powerful enough or an aggressive algorithm which requires the transformation into a static single assignment (SSA) form of programs (and back to the original form after optimization), thus causing too much overhead not appropriate for JIT compilation. This paper proposes a new algorithm based on an inequality graph which can eliminate array bounds check codes aggressively without resorting to the SSA form. When we actually perform this type of optimization, there are many constraints in code motion caused by the precise exception rule in Java specification, which would cause the algorithm to miss many opportunities for eliminating array bound checks. We also propose a new method to overcome these constraints.

Key words : *Java, Just-in-Time Compiler, Java Virtual Machine, Array Bounds Check Elimination*

† 정 회 원 : 삼성전자 DM총괄DM연구소 연구원
sun12@altair.snu.ac.kr
†† 종신회원 : 서울대학교 전기공학부 교수
smoon@altair.snu.ac.kr
논문접수 : 2005년 2월 7일
심사완료 : 2005년 10월 11일

1. 서 론

Java 프로그램을 실행하는 데 있어 아직은 그 성능이 C 언어로 작성된 프로그램의 성능과 많은 차이를 보이고 있다. 그 중요한 이유 중 하나는 컴파일된 Java 프

로그램의 실행 코드에 많은 수의 검사 명령어들이 포함되어 있기 때문이다. Java 언어 specification에서는 객체 변수의 속성 값을 참조할 때마다, 배열 변수의 원소를 참조할 때마다 객체 변수 또는 배열 변수가 null인지를 검사하도록 강제하고 있으며(null 검사), 또한 배열의 원소를 참조할 때마다 해당 index가 0보다 크고 배열의 길이보다 작은지를 검사하도록 강제하고 있다(배열 경계 검사).

우리는 본 논문에서 위의 두 가지 검사 중 배열경계 검사를 제거하는 방법에 대해 논하고자 한다.

배열 경계 검사는 다음 두 가지 점에서 실행 코드의 성능을 저하시킨다. 첫째로 배열 경계를 검사하는 것 자체가 성능을 느리게 한다. 배열 경계를 검사하기 위해서는 서로 자료 의존적(data-dependent)인 add, load, compare 명령어가 순차적으로 실행되도록 해야 한다. 이는 무척 비싼 작업이며 또한 그 실행 빈도가 매우 높기 때문에 프로그램 성능 저하의 큰 요인이 된다. 둘째로 배열 경계 검사는 Java의 정확한 예외 처리 정책과 맞물려 명령어 재배치 최적화(instruction schedule)를 비롯한 다른 최적화 기법들의 제약으로 작용한다.

중복된 배열 경계검사들을 제거하기 위한 알고리즘은 지금까지 많은 연구가 진행되어왔다. 그러나 대부분의 기존 알고리즘들은 컴파일 시간을 고려해야 한다는 특수성을 가진 Just-in-Time 컴파일러에 적용하기에 너무 무겁다. 반면에 가벼운 변수 범위(value-range) 분석에 기반한 알고리즘들도 있으나 이들은 충분한 성능을 발휘하기에 부족하다. 그래서 현재 많은 Just-in-Time 컴파일러들은 반복 영역 개선 최적화(Loop versioning)를 통해 제한적인 코드 영역에서 배열 경계 검사들이 실행되는 횟수를 줄이는 방법을 쓰고 있다[1,2]. 그러나 이 방법은 최적화가 적용되는 범위가 제한적일 뿐만 아니라 실행 코드의 크기를 늘이는 단점이 있다. 근래에는 이러한 단점들을 극복하고자 정적인 컴파일 환경에서 제안된 Gupta의 알고리즘[3]을 변형하여 적용하거나 흐름도 독립적(flow-insensitive)인 표현 방식의 특수성을 이용해 배열 경계 문제를 그래프 문제로 바꾸어서 풀고자 하는 시도(ABCD-Array Bounds Checks Elimination on Demand [4])가 있어 왔다.

Gupta의 알고리즘은 문제를 비트 벡터(bit vector) 문제로 단순화하여 자료 흐름 분석(data flow analysis) 방법을 통해 중복된 배열 경계 검사 명령어를 찾는다. 비트 벡터를 이용하는 것은 변수들 간의 대소관계를 충분히 표현할 수 없기 때문에 최적화를 마친다고 하여도 여전히 중복된 배열 경계 검사 명령어가 남아있을 수 있는 단점이 있다.

반면에 ABCD는 정적 단일 배정 형태 상에서 적용되기 때문에 자료 흐름분석 방법을 쓰지 않아도 된다. 그리고 그래프를 통해 변수들 간의 대소관계를 표현하고 있다. 하지만 정적 단일 배정 형태로의 변환과 원래 형태로의 되돌림을 해야 하는 부담이 있고, 정적 단일 배정 형태로의 변환으로 인해 변수의 개수가 늘어나 이를 표현하기 위한 그래프의 크기가 커져서 발생하는 부담이 있기 때문에 가벼운 컴파일러를 지향하는 Just-in-Time 컴파일러의 성질에 안 맞는 측면이 있다.

본 논문에서는 ABCD와 같이 그래프를 사용하여 문제를 표현하되 정적 단일 배정 형태의 표현 방식을 사용하지 않고 문제를 해결하는 방법을 제시한다. 정적 단일 배정 형태의 표현 방식을 사용하지 않고 문제를 해결하기 위해서는 자료 흐름 분석 방법을 사용해야 하는데 이를 그래프와 동시에 사용하기 위해서는 여러 문제를 해결하여야 한다. 이 문제를 해결하는 것이 이 논문의 주요 요점 중의 하나이다.

자료 흐름 분석 방법을 통해 배열 경계 검사 최적화를 하는 것은 두 단계를 통해 이루어진다. 먼저 역방향으로 최적화(backward data flow analysis)를 한 후에 순방향으로 최적화(forward data flow analysis)를 하는 것이다. 그러나 역방향으로의 최적화는 java의 특성 때문에 적용할 수 있는 경우가 매우 제한적이다. 본 논문에서는 이를 극복할 새로운 역방향 최적화 방법도 제시할 것이다.

2. 그래프를 이용한 순방향 분석(forward data flow analysis)

명령어들 중에는 해당 인수(operand) 사이의 대소관계를 알 수 있는 정보를 포함하고 있는 것들이 있다. 이 정보들을 체계적으로 관리해 가능한 한 많은 제거 가능한 배열 경계 검사 명령어를 찾아내는 것이 배열 경계 검사 최적화의 목적이다. 본 알고리즘에서는 이 정보를 그래프의 정점(node)과 간선(edge)으로 표현한다. 그리고 이 그래프(이하 대소관계 그래프)를 사용해 배열 경계검사 최적화 문제를 해결한다.

2.1 Basic block 안에서의 분석

대소관계 그래프는 방향성 그래프(directed graph)이며 각각의 정점(node)은 분석할 변수들을 의미하고 간선(edge)은 그 변수들 간의 대소관계를 의미한다. 즉, 대소관계 그래프에서 a→b(가중치=c)는 변수 a와 변수 b 사이에 "a+c>=b"라는 관계가 있음을 의미한다. 자료 흐름 분석을 수행하면서 각 명령어는 대소관계 그래프에 표 1과 같이 정점과 간선을 생성한다.

분석 과정에서 만약 변수 a가 더 이상 live하지 않다는 것을 알게 되면 그래프에서 정점 a와 정점 a에 연결된 모든 간선을 제거한다. 이 때 정점 a를 목적지로 하

표 1

명령어	생성되는 간선	
v = w	v → w	가중치= 0
	v ← w	가중치= 0
v = c	v → 0	가중치= -c
	v ← 0	가중치= c
v = w + c	v → w	가중치= -c
	v ← w	가중치= c
v = new_array(c)	v → 0	가중치= -c
	v ← 0	가중치= c
v = array_length(a)	v → a	가중치= 0
	v ← a	가중치= 0
check v + c >= w	v → w	가중치= c
	v ← w	가중치= -c

※v, w는 변수, c는 상수, a는 배열 변수

는 간선의 출발지와 정점 a를 출발지로 하는 간선의 목적지 사이의 정보가 없어지지 않도록 두 정점 사이에 새로운 간선을 만든다. (ex. 대소관계 그래프에서 다음과 같은 간선 x→a→y가 존재하고 분석 과정에서 a가 더 이상 live하지 않으면 그래프에서 간선: x→y를 생성하고 정점 a와 간선: x→a, 간선: a→y를 제거한다.)

자료 흐름 분석 과정에서 배열 경계 검사명령어를 만나면 그 검사 명령어가 중복되는 것인지를 검사하게 된다. 이를 검사하기 위해서 먼저 대소관계 그래프를 통해 비교 대상인 두 변수 사이의 대소관계를 조사한다. 이 대소관계를 알아내는 것은 그래프에서 두 정점 사이의 경로를 찾는 문제와 동일하다. 배열 경계 검사 명령어 "check i<=a.length-1"의 경우 정점 a에서 정점 i로의 경로가 있고 그 경로 상의 간선들의 가중치의 합이 -1 이하인 경로가 있다면 "i<=a.length-1"은 항상 참이므로 이 명령어는 중복된 검사이다. 만약 그래프에 경로가 존재하지 않거나 경로들의 가중치의 합이 -1 보다 크면 그래프에 새로운 간선 "a→i (가중치=-1)"를 추가한다.

Gupta의 방법과는 달리 대소관계 그래프를 이용한 방법은 분석 과정에서 임의의 변수가 상수 값을 가지고 있는지 그리고 0보다 큰지 작은지를 검사할 수 있다. 따라서 두 변수의 덧셈 또는 뺄셈 명령어에서도 변수들 간의 대소관계를 파악할 여지가 있다.

2.2 확장 basic block(extended basic block)에서의 분석

Basic block 내에서 분석하는 경우와 비교해 조건 명령어만 더 처리해 주면 확장 basic block에서의 분석이 가능하다. 일례로 그림 1과 같은 예제에서 BB2의 초기값은 BB1의 결과 그래프에 a→b(가중치=0)을 추가한 그래프이고 BB3의 초기값은 BB1의 결과 그래프에 a←b(가중치=0)을 추가한 것이다.

2.3 전역 분석 방법(Global analysis)

```
BB1 :  a = ···
       b = ···
       if (a >= b) {
BB2 :      ···
       } else {
BB3 :      ···
       }
```

그림 1

본 배열 경계 검사 최적화 알고리즘의 전역 분석은 일반적인 자료 흐름 분석방법과 같이 각 basic block의 결과가 모두 수렴될 때까지 분석이 진행된다. 그러나 완벽한 분석을 하기 위해선 많은 시간과 자원을 소모하므로 우리는 보다 빠른 수렴을 위한 새로운 방법을 채택하였다. 전역 분석의 대략적인 방법은 그림 2와 같다.

전역 분석은 확장 basic block에서의 분석에 다음 두 가지 기능을 추가하면 된다.

1. 흐름도의 모든 join 지점에서 부모 basic block의 결과 그래프들이 공통으로 가지고 있는 경로들을 계산한다. 그 계산 결과가 join 지점에 해당하는 basic block의 초기 그래프 값이다(그림 2의 3.1).
2. 현재 반복 순서(iteration)에서의 결과 그래프와 이전 반복 순서에서의 결과 그래프가 같은지 비교한다(그림 2의 3.3 그리고 3.4, 3.5).

그림 2의 3.1과 3.3 작업 모두 두 그래프를 비교하는

1. 모든 basic block에 대해 "분석을 해야 함" 상태로 초기화한다.
2. 모든 basic block에 대해 그 결과를 NULL 그래프로 초기화한다.
3. "분석 해야 함" 상태의 모든 basic block에 대해
   - 3.1 부모 basic block의 결과들에 meet 연산을 한다. 이 결과가 basic block에 지역 분석(local analysis)을 시작하기 위한 초기값이다.
   - 3.2 basic block에 지역 분석을 실시하여 이 basic block의 결과값을 얻는다.
   - 3.3 결과값을 이전 결과값과 비교한다.
   - 3.4 basic block의 상태를 "분석 끝남"으로 바꾼다.
   - 3.5 3.3의 결과가 "다르다"라면 basic block의 모든 자식 basic block에 대해 그 상태 값을 "분석을 해야 함"으로 바꾼다.
4. 모든 basic block의 상태가 "분석 끝남"이 될 때까지 3을 계속한다. 만약, 분석이 오래 걸릴 것 같으면 포기한다.
5. 만약 분석을 포기했으면 분석 과정 중에 바꾼 basic block의 상태를 원래대로 복구한다.

그림 2

작업이 필요하다. 두 그래프의 비교는 매우 복잡한 작업이다. 그러나 basic block을 넘어서 live한 변수의 개수가 많지 않은데다가 그 변수 사이에 대소관계를 가지고 있는 경우는 다시 그 일부에만 해당된다. 따라서 그래프를 구성하는 정점과 간선의 개수가 충분히 작기 때문에 실제로 그 수행 시간은 그렇게 길지 않다. 그러나 이런 그래프 비교 작업의 회수가 빈번하기 때문에 가능한 한 그 회수를 줄이는 노력이 필요하다.

그림 2의 3.3 작업의 경우 매 반복 때마다 그래프를 비교하는 작업을 하는 것이 꼭 필요한 것은 아니다. 굳이 비교하지 않더라도 현재 반복 순서에서의 그래프가 이전 반복 순서에서의 그래프와 다를 가능성이 많다면 일단 "비교 결과가 다르다"라고 단정짓고 다음 반복 순서 때 자세히 비교해도 되는 것이다. 결과 그래프의 수정 여부를 예측하는 것은 basic block 마다 비트 벡터를 둔 후 그 것들을 OR 연산하는 것만으로도 구현이 가능하다.

$bv(B)$의 초기값 = Basic block B에 해당하는 bit을 1로 셋팅한 값

$$bv(B) = \bigsqcup_{\text{B의 모든 부모 basic block P}} bv(P)$$

수식 1

Basic block B의 비트 벡터 $bv(B)$에 대해 수식과 같이 초기화 시킨 후 그림 2의 3.3 작업을 수행하기 직전마다 수식에서와 같이 OR 연산을 하면 $bv(B)$는 현재까지의 분석 중 방문한 basic block을 의미하게 된다. 만약, 이전 반복 순서 때의 $bv(B)$가 현재 반복 순서때의 $bv(B)$와 다르다면 이전 반복 순서 때와 현재 반복 순서 때 사이에 새로 방문한 basic block이 있다는 뜻이다. 만약 그렇다면 새로 방문한 basic block이 끼친 영향에 의해 두 결과가 다를 가능성이 많은 것이다. 따라서 이 경우에는 "결과가 다르다"라고 예측하고 3.3 작업을 생략할 수 있다.

그림 3은 배열 변수 a에 있는 원소들의 합을 구하는 기능을 하는 함수를 예로 들어 이를 분석하는 과정을 나타낸 것이다(배열 경계 검사와 관련 있는 명령어만을 나타냄).

<iteration 2>에서 $BB2_{IN}$은 <iteration 1>의 $BB1_{OUT}$의 대소관계 그래프와 $BB3_{OUT}$의 대소관계 그래프를 join한 대소관계 그래프다. 그리고 $BB2_{OUT}$은 $BB2_{IN}$을 초기값으로 BB2를 분석한 후의 그래프이다.

일반적인 자료 흐름분석 방법으로 분석을 하게 되면 그림 3의 <iteration 2>와 <iteration 3>에서 볼 수 있듯이 반복 영역 안에 있는 add 명령어의 영향으로 매 반복마다 1씩 증가하는 간선이 생김을 알 수 있다. 이렇게 되면 반복 분석이 수렴하지 않고 무한히 계속되므로 예제의 add 명령어에서의 변수 i와 같은 반복 유도 변수(loop induction variable)의 영향을 제거하여야 한다.

우리 알고리즘은 이 문제를 해결하기 위하여 앞에서 설명한 비트 벡터를 다시 사용한다. 즉, basic block B의 이전 반복 순서 때와 현재 반복 순서 때의 비트 벡터 값 $bv(B)$이 같을 경우 두 시점의 그래프를 비교해 간선의 가중치 값이 변한 간선들을 제거하는 것이 그것이다. 의미 상 비트 벡터 값이 변하지 않았다는 것은 그

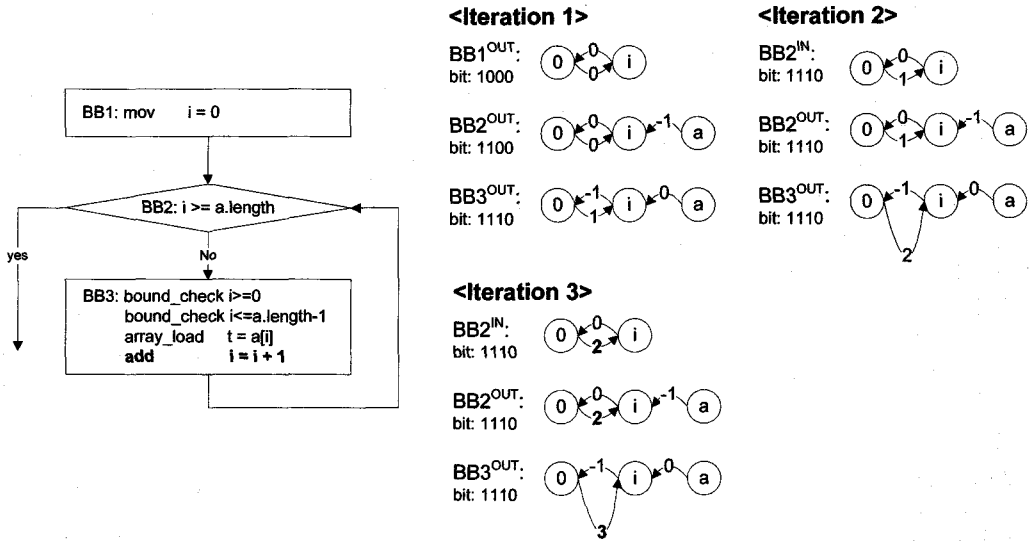


그림 3

동안 새로운 basic block을 방문하지 않았다는 것이고 따라서 그럼에도 불구하고 비트 벡터 값이 변했다는 것은 반복 영역안에 있는 어떤 명령어에 의해 그 값이 지속적으로 증가 또는 감소하는 변수가 있을 가능성이 많다는 것을 의미한다. 우리는 이것이 실험적으로 상당히 바른 예측임을 알 수 있었다. 비록 이 판단이 틀려 다른 이유에 의해 간선의 가중치 값이 변했다 하더라도 간선을 제거하는 것은 프로그램의 정확성(correctness)을 해치지는 않는다. 대소관계 그래프에서 간선을 제거하는 것은 분석의 정확도를 떨어뜨릴 뿐이지 잘못된 정보를 만드는 것이 아니기 때문이다.

그림 4는 <iteration 2>의 BB3에서 간선을 제거한 모습과 그 결과로<iteration 3>의 대소관계 그래프들이 변한 것을 나타낸 것이다. 결국, 다음 네 번째 반복 순서 때의 결과가 세 번째 반복 순서때의 결과와 같아져 수렴하게 된다. 결과적으로 이 최적화 과정을 통해 BB3의 두 배열 경계 검사 명령이 제거될 수 있다.

3. 그래프를 이용한 역방향 분석

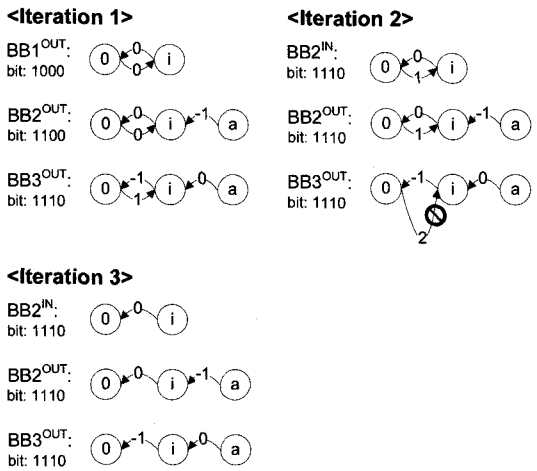


그림 4

그림 5는 배열 변수 a의 i번째 값을 변수 x에 i+1번째 값을 변수 y에 저장하는 간단한 가상코드(pseudo-code)이다. 그림 5의 A에서 (f)의 검사 결과가 참이면 (c)도 항상 참이다. 따라서 B와 같이 (f)에서 수행될 검사를 미리 (c)에서 수행해도 프로그램은 바르게 동작한다(A, B 모두 i+1 > a.length -1일 때 배열 경계 예외(bounds check exception)가 발생한다). 이렇게 나중에 실행될 검사 명령이 먼저 실행될 검사 명령을 의미상 포함하게 될 경우 후자를 전자의 위치로 옮길 수 있는데 이것은 역방향 자료 흐름 분석을 통해서 가능하다. 역방향 분석이 완료된 이 후엔 순방향으로 분석을 수행하여 중복된 배열 경계 검사 명령어들을 지우게 된다.

그러나, Java 프로그램에서 이런 역방향 최적화가 항상 가능한 것은 아니다. Java 언어는 정확한 예외 처리(precise exception)를 강제하고 있는데, 이것은 최적화가 일어나기 전의 원본 코드와 최적화를 적용한 코드가 있을 때, 원본 코드에서 예외가 발생을 했으면 최적화된 코드에서도 같은 종류의 예외가 발생해야 하며 예외가 발생했을 때 원본 코드의 프로그램 상태와 최적화된 코드의 프로그램 상태가 같아야 하는 것을 의미한다. 이것을 만족시키기 위해 경계 검사명령어가 움직이는 동선 상에 예외를 발생 시킬 수 있는 명령어가 없어야 하고 프로그램의 상태를 바꿀 수 있는 명령어가 없어야 한다. 그림 5에서의 예제의 경우 (c)와 (f)사이에 변수 x의 값을 수정하는 명령어 (d)가 있기 때문에 역방향 최적화를 할 수 없게 된다. 그러나, 만약 method에 배열 경계 예외를 처리할 처리기(handler)가 없다면 프로그램이 변수 x를 더 이상 사용하지 않으므로 값이 바뀌어도 프로그램의 동작에 영향을 미치지 않는다. 이런 예외적인 경우 최적화를 수행할 수 있다. 따라서 그림 5는 method에 배열 경계 예외 처리기가 없는 경우 사실상 최적화를 적용할 수 없는 경우의 예를 든 것이 된다.

본 장에서는 먼저 역방향 분석을 대소관계 그래프를 이용해 적용하는 방법을 얘기하고 다음으로 위와 같은 역방향 최적화의 문제점을 극복할 수 있는 방법을 소개

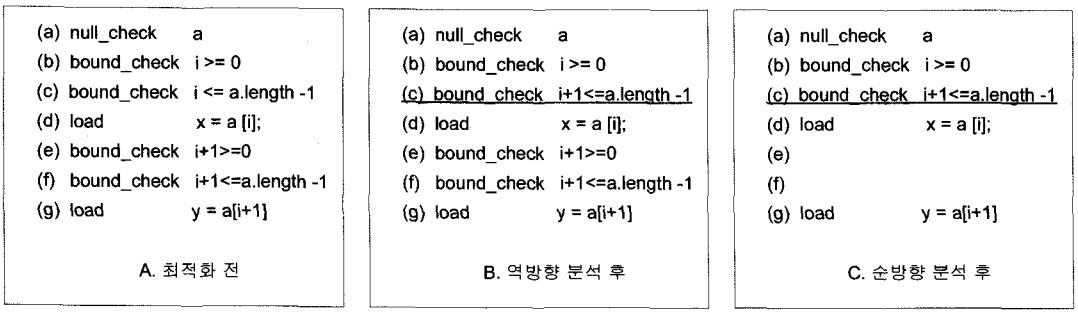


그림 5

하고자 한다.

3.1 대소관계 그래프를 이용한 역방향 분석 방법

순방향 분석 방법과 같이 그래프의 정점은 변수를, 간선은 그들 간의 대소관계를 의미한다.

역방향 분석 방법은 basic block의 마지막 명령어에서부터 분석이 진행된다. 분석 과정 중 새로 정의된 변수가 있으면 순방향 분석 때 live하지 않은 변수를 처리할 때처럼 그 변수의 해당 정점을 그래프에서 제거한다. 다음 그림 6은 그림 5에 있는 프로그램을 역방향 분석하는 과정을 나타낸 것이다.

그림 5의 (g)에서 (d)까지 분석하면 그림 6의 실선으로 그려진 것과 같은 대소관계 그래프를 얻어낼 수 있다. 역방향 분석 때는 순방향 분석 때와는 달리 그 간선이 배열 경계 검사 명령어에 의해 생성된 것일 경우 매번 그 배열 경계 검사명령어를 간선에 저장해둔다. 똑같은 방법으로 (c)를 분석하면 점선으로 그려진 간선을 새로 생성하게 되는데, 이 때 새로 생성될 간선의 가중치가 이미 존재하고 있는 간선의 가중치보다 크기 때문에 (c)가 (f)에 의미상 포함되는 검사임을 알 수 있다. 따라서 이 때 (f)를 (c)로 옮기는 작업을 수행하면 된다. 그림 (B)는 배열 경계 검사 명령어를 옮긴 후 (b)와 (a)의 분석이 끝난 후의 상태를 보여주는 것이다.

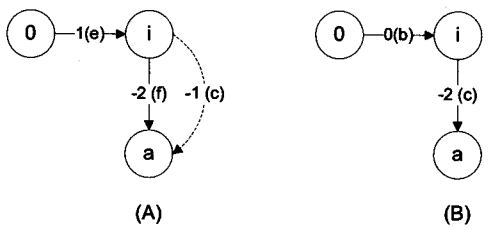


그림 6

3.2 역방향 최적화의 제약을 극복하는 방법

역방향 최적화는 배열 경계 검사 명령어의 이동 동선상에 예외를 발생시킬 수 있거나 프로그램의 상태를 바꿀 수 있는 명령어가 하나도 없을 경우에만 적용될 수 있다. 따라서 매우 제한적으로 적용되기 때문에 그 효과를 충분히 발휘할 수 없다. 우리는 이러한 문제점을 일단 공격적으로 최적화를 적용한 후 이것이 잘못된 결과를 야기할 수 있을 가능성이 있을 땐 최적화하기 전의 코드를 실행하도록 하는 방법으로 해결하였다. 그림 7은 그림 5의 코드에 위에서 설명한 기법을 적용한 것이다.

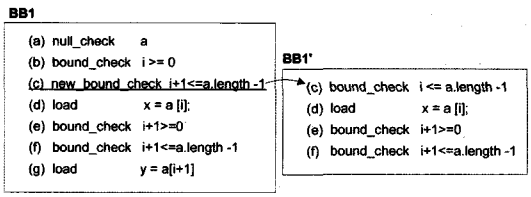
배열 경계 검사 명령어는 앞에서 언급한 것처럼 load, compare, branch로 구현된다. 즉, compare의 결과가 거짓이면 예외를 처리해주는 처리기로 도약(branch)하는 것으로 구현되는 것이다. 본 알고리즘에서는 그림 7의 BB1의 (c)와 같이 위로 이동할 배열 경계검사 명령어에 대해선 예외 처리기로 도약하는 것 대신 복사된 코드(최적화를 적용시키지 않은 코드)의 해당 명령어로 도약하도록 구현했다. 위 방법을 예제로 설명하면 다음과 같다.

그림 5 A의 (c) 시점에서 변수 i의 값이 (f)의 조건을 만족하는 경우와 만족하지 않을 경우로 나누어 보았을 때,

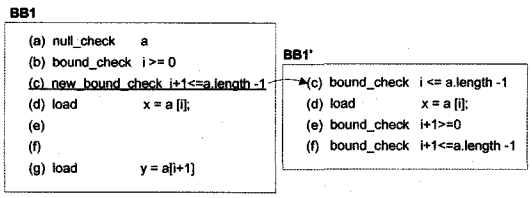
1. 만약 만족한다면 (c)와 (f) 모두에서 예외가 발생하지 않을 것이므로 그림 7의 BB1과 같이 (f)를 (c) 위치로 옮기는 것은 문제가 되지 않는다.

2. 만약 만족하지 않는다면 -즉, 그림 7의 (c)가 거짓이면- (c) 시점부터 최적화를 적용하지 않은 코드(BB1')를 실행시킨다.

일반적으로 예외는 말 그대로 매우 드물게 발생하므로 확률적으로 대부분의 경우 공격적인 최적화를 적용한 코드가 실행된다. 만약 2번의 경우라 하더라도 우리의 방법을 적용시키지 않은 경우와 비교해 branch 명령 하나만을 더 실행하게 되므로 가벼운 부담이라고 할 수 있다.



(a) 역방향 최적화를 적용한 후



(b) 순방향 최적화를 적용한 후

그림 7

4. 결 과

본 실험의 결과는 Intel Itanium 733MHz processor, 1GB RAM의 하드웨어와 Windows XP 64-bit Edition Version 2003 운영체제 위에서 동작하는 컴퓨터에서 얻었다. 본알고리즘은 Intel의 Open Runtime Platform JVM 위에서 구현된 VLaTTe 컴파일러[5] 안에 구현되었다. 사용한 벤치마크는 spec jvm98이다.

표 2는 각 벤치마크 프로그램의 총 method의 개수, 배열 경계 검사 최적화를 적용한 method의 개수, 배열

표 2

	M	적용 M	Iteration
_200_check	371	33	2.94
_201_compress	276	40	3.15
_202_jess	642	77	3.70
_209_db	313	41	3.10
_213_javac	883	67	3.12
_222_mpegaudio	398	100	3.41
_227_mtrt	402	51	3.27
_228_jack	530	40	2.85

M: method 개수 / 적용 M: 배열 경계검사 최적화를 적용한 method의 개수 / Iteration: 자료 흐름 분석이 수렴하기 위한 평균 반복 분석수

경계 검사 최적화를 적용했을 때 분석을 마치기 위해 수행한 반복 분석의 수를 나타낸 것이다.

Method의 basic block의 수가 많아서 최적화 수행 시간이 길 것이라 예측될 경우(basic block 개수 100개 이상) 배열 경계 검사최적화를 적용하지 않았고, method 내에 배열경계 검사 명령어가 1개 이하로 존재할 때도 적용하지 않았다. 배열 경계 검사 최적화를 적용하더라도 분석 시간이 많이 들 것으로 예상되는 것들은 확장 basic block 범위까지만 최적화를 적용하였다.

그림 8은 배열 경계 검사 최적화를 적용하지 않았을 때의 컴파일 시간 + 수행 시간을 기준으로 배열 경계 검사 최적화를 적용했을 때, 배열 경계 검사 최적화 + 새로운 역방향 최적화 방법을 적용했을 때의 컴파일 시간 + 수행 시간을 측정해 각각의 성능 향상을 나타낸 것이다. 배열 경계 검사 최적화를 적용했을 때 기하평균(geo-mean)으로 1% 정도의 성능 향상을 얻었고, 특히, _222_mpegaudio와 같이 배열 변수를 많이 사용하는 벤치마크 프로그램에서는 4%의 성능 향상을 얻었다. _202_jess와 _209_db에서 성능이 저하되었는데 이는 이 벤치마크 프로그램이 배열을 별로 사용하지 않아 배열 경계 검사 최적화의 대상이 별로 없어서 최적화의 이득보다는 최적화 수행 시간만큼의 부담이 작용했기 때문인 것으로 생각된다. 그러나, 성능 저하가 최적화 수행 시간(약 0.2%) 보다 훨씬 큰데 이는 다른 외적인 요인-캐쉬(cache) 행동의 변화-이 있는 것으로 생각된다. 특히, _202_jess의 경우에는 컴파일 시간과 수행 시간을 나누어서 측정할 경우에는 성능 저하를 보이지 않다가 이 두 시간을 동시에 측정하면 성능 저하를 보이는 경우이어서 원인 분석 후 수정하면 더 좋은 결과를 얻을 수 있을 것으로 보인다.

새로운 역방향 최적화 방법을 적용하면 기하평균 0.5% 정도의 추가적인 성능향상을 더 얻을 수 있었다. _227_mtrt의 경우 새로운 역방향 최적화가 적용된 method 중에 매우 빈번히 실행되는 method가 포함되어서 다른 벤치마크 프로그램들에 비해 더 많은 1.5%의 추가적인 성능 향상을 보였다. _222_mpegaudio의 경우엔 성능이 나빠졌는데, 이는 최적화 적용으로 인해 코드 크기가 늘어나(배열 경계 검사 최적화만을 적용했을 때의 코드 크기에 비해 9.4% 증가) 캐쉬 성능에 나쁜 영향을 주었기 때문이 아닌가 생각된다. 새로운 역방향 최적화 기법에 의해 복사되는 코드(그림 7의 BB1')에 대해 추가적인 최적화를 하지 않도록 하고-드물게 실행되는 코드 영역에 대해 최적화 포기-, 코드 영역 재배치(code reordering) 방법을 적용해 명령어 캐쉬(instruction cache)의 성능 저하를 막도록 구현을 보완하면 성능 저하를 줄일 수 있을 것이다.

그림 9와 그림 10은 배열 경계 검사 최적화로 인한

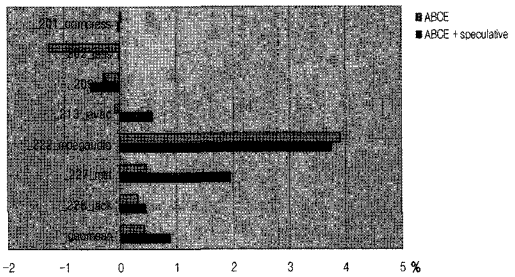


그림 8 성능 향상 (ABCE: 배열 경계 검사 최적화를 적용 시켰을 때, ABCE +speculative : 새로운 역방향 최적화 방법을 추가로 적용했을 때)

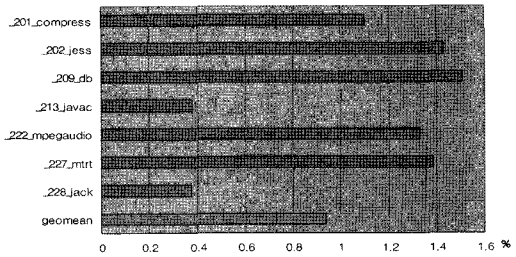


그림 9 전체 컴파일 시간에서 배열 경계 검사 최적화가 차지하는 시간(%)

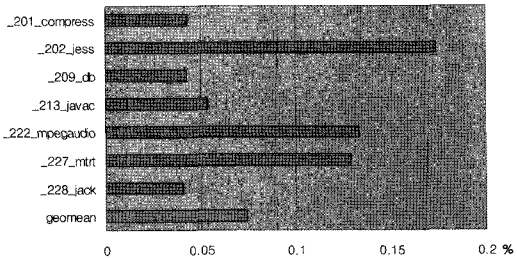


그림 10 전체 수행 시간에서 배열 경계 검사 최적화가 차지하는 시간 (%)

부하를 나타낸 것이다. 배열 경계 검사 최적화의 분석 시간은 전체 컴파일 시간의 1.5% 이내, 전체 수행 시간의 0.2% 이내로 크지 않음을 알 수 있다. 이는 본 알고리즘이 Just-in-time 컴파일 환경에 매우 적합함을 말해주는 것이다.

5. 관련 연구

대소관계 그래프의 개념은 ABCD[1]와 Fen Quian이 제안한 방법[6]에서도 사용되었다. Fen Quian의 경우에는 대소관계 그래프를 제약 조건 그래프(Constraints graph)라고 하였다. 기본적으로 그래프를 이용해 변수들 간의 대소관계를 표현하고 배열 경계 검사 명령어가 중복되는 지의 여부를 최단 경로 방법으로 푸는 것은 같으나 우리의 대소관계 그래프는 이들과 다음과 같은 점에서 다르다.

ABCD 방법에서는 명령어를 확장 정적단일 배정(e-SSA) 형태[1]로 변환한 후에 그래프를 생성한다. 확장 정적 단일 배정 형태의 표현 방식은 정적 단일 배정 형태의 표현 방식을 확장한 것으로 하나의 변수는 하나의 변수 범위 정의(value range constraints definition)를 갖게 된다. 이 같은 확정정적 단일 배정 형태의 표현 방식의 특성에 의해 ABCD에서 사용하는 대소관계 그래프는 우리의 대소관계 그래프와는 달리 자료 흐름 분석을 하지 않고도 얻어 질 수 있다. 하지만 분석을 시작하기 위해 명령어 표현 방식을 확정 정적 단일 배정 형태로 변환해야 한다는 점과 그 결과 만들어진 그래프의 크기가 커서 - 하나의 그래프가 method 내의 모든 변수와 그 변수들 간의 대소관계를 표현하고 있고 있기 때문에 - 나중에 최단 경로를 찾을 때 시간이 많이 걸린다는 점에서 부담이 된다.

Fen Quian이 제안한 배열 경계검사 최적화 방법은 Just-in-Time 컴파일 중에 분석을 하는 것이 아니라 컴파일 하기 전에 Java class 파일을 분석한 후 그 결과를 class 파일에 저장하는 방법이다. 이 방법은 분석 시간이 성능에 영향을 주지 않기 때문에 분석 시간을 줄이는 것에 큰 관심을 두지 않았다. 또한 이 방법은 저장된 분석 결과를 인식할 수있는 Just-in-Time 컴파일러를 통해 실행하여야만 효과를 볼 수 있다는 단점이 있다.

Java의 정확한 예외 처리규칙으로 인한 제약을 극복하기 위한 방법은 Manish Gupta[7]에 의해 제안된 바가 있다. 이 방법은 예외를 발생시킬 수 있는 명령어(이하 PEI)에 의해 제약 받는 명령어 재배치를 가능하게 해 주는 틀을 제공한다. 그러나 이것은 PEI 자체를 없앨 수는 없는 방법으로 PEI가 배열 경계 검사명령어인 경우 이를 제거하는 것을 통해명령어의 재배치를 가능하게 하는 우리의 방법과는 다르다. 우리의 방법은 배열 경계 검사 명령어의 경우에만 적용될 수 있지만 Manish Gupta의 방법보다 간단하게 구현될 수 있으며 분석 시간도 짧다.

6. 결론 및 향후 보완 연구

이 논문에서 우리는 Just-in-Time 컴파일러에 적합하면서도 성능이 좋은 배열 경계 검사 최적화 기법을 제안했다. 그리고 기존의 역방향 최적화 기법의 제약을 극복할 수 있는 새로운 최적화 기법을 제안했고 그 것이 성능 향상에 기여함을 보였다.

앞으로 우리가 제안한 최적화 기법이 더욱 높은 성능 향상을 보이기 위해서는 배열 경계 검사 최적화의 대상이 되는 method를 보다 지능적으로 예측해 최적화 효과가 작은 method에 대한 배열 경계 검사 최적화의 수행에 따른 부담을 피하는 노력이 필요하다. 이는 새로운 역방향 최적화 방법의 경우도 마찬가지이며 이와 더불어 역방향 최적화 방법이 코드 크기 증가를 일으키는 정도를 줄이는 노력도 더 필요하다 하겠다.

참 고 문 헌

[1] Michal Cierniak, Guei-Yuan Lueh and James M. Stichnoth. Practicing JUDO: Javaunder dynamic optimizations. ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. v.35 n.5. 2000.

[2] Rajiv Gupta. Optimizing array bound checks using flow analysis. ACM Letters on Programming Languages and Systems (LOPLAS). v.2 n.1-4, pp.135-150. 1993.

[3] Rastislav Bodík , Rajiv Gupta and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. ACM SIGPLAN Notices. v.35 n.5, pp.321-333, 2000.

[4] Rastislav Bodík , Rajiv Gupta and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. ACM SIGPLAN Notices. v.35 n.5, pp.321-333. 2000.

[5] S.Kim, S.-M. Moon and K. Ebcioglu. vLaTTe: A Java Just-in-Time Compiler for VLIW with Fast Scheduling and Register Allocation. July 2000.

[6] Feng Qian, Laurie J. Hendren and Clark Verbrugge. A Comprehensive Approach to Array Bounds Check Elimination for Java. Lecture Notes In Computer Science, Proceedings of the 11th International Conference on Compiler Construction. pp.325-342. 2002.

[7] Manish Gupta ,Jong-Deok Choi and Michael Hind. Optimizing Java Programs in the Presence of Exceptions. Lecture Notes In Computer Science,

Proceedings of the 14th European Conference on Object-Oriented Programming, pp.402-446. 2000.

최 선 일

1999년 서울대학교 전기공학사. 2005년 서울대학교 석사. 현재 삼성전자 DM연구소 선임 연구원. 관심분야는 Java virtual machine, Java Just-in-Time compiler, Java Ahead-of-Time compiler

문 수 묵

정보과학회논문지 : 소프트웨어 및 응용 제 32 권 제 7 호 참조