대소관계 그래프를 이용한 Just-in-time 컴파일 환경에서의 array bounds check elimination

최선일°, 문수묵
서울대학교 전기컴퓨터 공학부
{sun12°, smoon}@altair.snu.ac.kr

Array bounds check elimination using Inequality Graphs in Just-in-time compiler

Sun-il Choi°, Soo-Mook Moon
Seoul National University

요 약

Just-Just-in-Time 컴파일러를 이용하여 자바의 성능을 향상시키려면 여러 문제들을 극복하여야 한다. 이 문제 중 중요한 부분을 차지하는 것이 null check 코드와 array bound check 코드를 어떻게 최적화하느냐는 것이다. Static한 컴파일 환경에서는 이미 많은 연구가 진행되어 매우 강력한 성능을 가지는 알고리즘이 알려져 있으나 이를 컴파일 시간이 수행시간의 일부인 Just-in-Time 컴파일 환경에 적용하기에는 컴파일 오버헤드가 너무 커서 적절하지 않다. 현재 Just-in-Time 컴파일러들은 가볍고 단순한 알고리즘을 적용하지만 중복된 array bounds check를 충분히 제거하지 못하거나 아니면 강력하지만 flow-insensitive한 SSA form을 기반으로 해야 하는 알고리즘을 사용하고 있다. SSA form의 적용은 SSA form으로의 변환과 되돌림에 의한 overhead로 가볍고 빠른 컴파일러를 지향하는 Just-in-Time 컴파일러에 부합되지 않는다. 본 논문은 변수간의 대소관계를 표현하는 그래프를 array bounds check 알고리즘에 적용해 충분한 power를 내면서도 flow-sensitive한 환경에서 동작하는 알고리즘을 제안한다.

1. 서 론

Java 프로그램을 실행하는데 있어 아직은 그 성능이 C 언어로 작성한 프로그램의 성능과 많은 차이를 보이고 있다. 그 중요한 이유 중 하나는 컴파일 된 Java 프로그램의 실행 코드에 많은 수의 check 코드들이 포함되어 있기 때문이다. Java language specification에서는 object 변수의 property를 사용하기 전과 배열 변수를 사용하기 전에 항상 object 또는 배열이 null인지를 체크하도록 강제하고 있으며, 또한 배열의 element를 사용할 때 해당 index가 0보다 크고 배열의 길이보다 작은지를 체크하도록 강제하고 있다.

우리는 본 논문에서 위의 두 가지 중 상대적으로 해결이 어려운 중복된 array bounds check들을 제거하는 방법에 대해 논하고자 한다.

Array bounds check는 다음 두 가지 점에서 실행 코드의 성능을 저하시킨다. 첫째로 array bounds check를 수행하기 위한 코드 그 자체가 성능을 느리게 한다. array bounds check를 실행하기 위해서는 서로 data-dependent한 add, load, compare instruction이 생성 되어야 한다. 또한 그 빈도가 매우 높기 때문에 프로그램의 성능 저하의 큰 요인이 된다. 둘째로 Array bounds check는 Java의 precise exception 정책과 맞물려 instruction schedule의 장애로 작용한다. 따라서 다른 최적화 기법들의 효과를 많이 저하시킨다.

중복된 array bounds check들을 제거하기 위한 알고리즘은 지금까지 많은 연구가 진행되어왔다. 그러나 대부분의 기존의 알고리즘들은 컴파일 시간을 고려해야 하는 Just-in-Time compiler에 적용하기에 너무 무겁다. 반면에 가벼운 value-range 분석에 기반한 알고리즘들은 충분한 성능을 발휘하기에 부족하다. 그래서 현재 많은 Just-in-Time 컴파일러들은 Loop versioning을 통해 제한적인 코드 영역에서 bounds check들이 실행되는 횟수를 줄이는 방법을 쓰고 있다. [1][2]그러나 이 방법은 최적화가 적용되는 영역이 제한적일 뿐만 아니라 실행 코드 크기를 늘이는 단점이 있다. 따라서 static한 컴파일러 환경에서 제안된 Gupta의 알고리즘[3]을 변형하여 적용하거나 flow-insensitive한 환경에서 array bounds 문제를 그래프 문제로 바꾸어서 풀고자 하는 시도(ABCD-Array Bounds Checks Elimination on Demand [4])가 있어왔다. 하지만 Gupta의 알고리즘은 bit vector를 기반으로 한 알고리즘을 사용하기 때문에 중복된 array bounds check를 찾는데 한계가 있고 반면에 ABCD는 SSA Form으로의 변환과 되돌림 그리고 변수들 간의 대소관계를 표현하는 그래프의 크기 문제로 인한 부담으로 인하여 가벼운 컴파일러를 지향하는 Just-in-Time 컴파일러의 성질에 안 맞는 측면이 있다.

본 논문에서는 data flow analysis의 틀에 변수들 간의 대소관계를 그래프로 표현하는 문제 해결 방식을 도입하여 중복된 array bounds check들을 찾는 성능을 높이되 그로 인한 overhead를 어떻게 줄이는가에 대해 논하고자 한다.

2. 그래프를 이용한 analysis

Instruction들 중에는 해당 operand 사이의 대소관계를 알 수 있는 정보를 포함하고 있는 것들이 있다. 이 정보들을 체계적으로 관리해 가능한 한 많은 제거 가능한 bounds check instruction을 찾아내는 것이 array bounds check elimination

최적화의 목적이다. 본 알고리즘에서는 이 정보를 그래프의 node와 edge로 표현한다. 그리고 이 그래프 (이하 대소관계 그래프)를 사용해 array bounds checks elimination 문제를 해결한다.

2.1. Local analysis

Data flow analysis를 수행하면서 각 instruction은 대소관계 그래프에 node와 edge를 생성한다. (표1) 참조.

표 1

Instruction	그래프에 생성되는 edge
v = w	v→w weight= 0 v←w weight= 0
v = c	v→0 weight= -c v←0 weight= c
v = w + c	v→w weight= -c v←w weight= c
v = new_array(c)	v→0 weight= -c v←0 weight= c
v = array_length(a)	v→a weight= 0 v←a weight= 0
check v + c >= w	v→w weight= c v←w weight= -c

※ v, w, x, y는 변수, c는 상수, a는 array변수.

분석 과정에서 만약 변수 a가 더 이상 live하지 않다는 것을 알게 되면 그래프에서 node a와 node a에 연결된 모든 edge를 제거한다. 이 때 node a를 target으로 하는 node와 node a를 source로 하는 node 사이의 정보가 없어지지 않도록 두 node 사이에 edge를 생성한다. (ex. x→a→y 에서 a가 last use일 때 x→y로 edge를 만들고 a node와 그 edge들을 제거)

결국 그래프에서의 " edge: a→b (weight=c)" 는 " a+c>=b" 라는 변수 사이의 대소관계를 나타내게 된다.

Flow analysis 과정에서 bounds checks instruction을 만나면 그 check가 중복되는 것인지를 검사한다. 이를 검사하기 위해서 먼저 대소관계 그래프를 통해 비교 대상인 두 변수 사이의 대소관계를 조사한다. 이 대소관계를 알아내는 것은 그래프에서 두 edge사이의 path를 찾는 문제와 동일하다. " check i<=a.length-1" 의 경우 node a에서 node i로의 path가 있고 그 path를 구성하는 edge의 weight들의 합이 -1 이하인 path가 있다면 " i<=a.length-1" 은 항상 참이므로 중복된 체크이다. 만약 path가 존재하지 않아서 중복된 check라고 확신할 수 없다면 그래프에 새로운 edge " a→i (weight=-1)" 를 추가한다.

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

2.2. Extended basic block으로의 확장

Basic block 내에서의 경우와 비교해 조건문만 더 처리해 주면 extended basic block으로의 확장이 가능하다. 그림 1에서 BB2의 초기 그래프는 BB1의 결과 그래프에 a→b (weight=0)을 추가한 그래프이고 BB3의 초기 그래프는 BB1의 결과 그래프에 a←b (weight=0)을 추가한 것이다.

그림 1

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

2.3. Global analysis

Global analysis는 각 basic block의 결과 그래프가 모두 수렴되어야 끝이 난다. 그러나 수렴을 100% 보장하기 위해서는 많은 CPU power와 memory를 요구하는 비싼 계산들을 수행하여야 한다. 따라서 100%의 수렴을 포기하는 대신 보다 빠른 분석을 보장하는 (하지만 대부분의 경우 결과를 수렴시키는) 알고리즘을 사용한다. (그림 2)

그림 2

```
bool isContinue = true
모든 Basic Block의 BB.analysis=UNDONE로 초기화
while loop<ITERATION_LIMIT_NUM and isContinue==true
  foreach BB.analysis==UNDONE인 모든 Basicblock BB
    graph= meet_graph_of_parents(BB) …①
    graph= do_local_analysis(BB,graph)
    isDiffer= compare_graph(graph, BB.p_graph)…②
    BB.analysis= DONE
    if isDiffer==true
      isContinue= true
      BB의 모든 successor SucBB.analysis=UNDONE
      BB.p_graph= graph
    end if
end foreach
end while
if loop>=ITERATION_LIMIT_NUM
  모든 Basic Block에 대해 원래대로 복구
end if
```

Global analysis는 extended basic block에서의 analysis에 다음 두 가지 기능을 추가하면 된다.

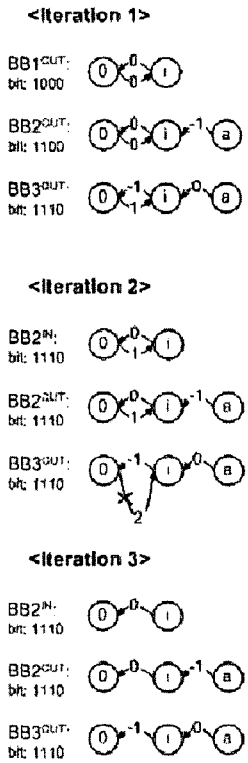
1. 모든 basic block의 join point에서 부모 그래프들이 공통으로 가지고 있는 path들을 계산한다. 그 계산 결과가 join된 basic block의 초기 graph값이다. (그림 2 ①)
2. 현재 iteration에서의 결과 그래프와 이전 iteration에서의 결과 그래프가 같은지 비교한다.(그림 2 ②)

1번, 2번 모두 두 그래프를 비교하는 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 node의 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 3

```
BB1: load_const i = 0
BB2: compare c1= (i>=a.length)
     (c1) jump BB4
BB3: check i>=0
     check i<=a.length-1
     load  t = a[i]
     add   i = i + 1
     jump BB2
BB4: ……
```

그림 4



2번의 경우 매 iteration마다 그래프 비교 작업을 하는 것이 꼭 필요한 것은 아니다. 굳이 비교하지 않더라도 현재 iteration에서 그래프가 변했을 가능성이 많다면 일단 " 비교 결과가 다르다" 라고 단정짓고 다음 iteration에서 자세히 비교해도 되는 것이다. 이것은 각각의 basic block 마다 bit vector를 필드를 추가하여 (각 bit는 basic block을 가리킴) 그들간의 OR 연산을 함으로써 구현할 수 있다.

그림 4는 그림 3을 분석하는 과정을 나타낸 것이다. 각 basic block에 할당된 bit은 모든 부모 basic block의 bit을 OR 연산한 것이다. <iteration 2>에서 $BB2^{OUT}$의 bit은 이전 iteration과 다르므로 <iteration 1>의 $BB2^{OUT}$의 그래프와 직접 비교하지 않고 아직 수렴되지 않았다고 판단한다. 왜냐하면 <iteration 1>의 BB3에서 수정된 그래프 내용이 <iteration 2>의 BB2에서 새로 감안되어야 하므로 그래프가 수정될 가능성이 많은 것이다.

<iteration 2>에서 $BB2^{IN}$은 <iteration 1>의 $BB1^{OUT}$와 $BB3^{OUT}$가 의미하는 대소관계 내용의 교집합이다. 그리고 $BB2^{OUT}$은 $BB2^{IN}$을 초기값으로 local analysis를 한 것이다.

<iteration 2>의 BB3는 <iteration 1>과 BB3와 비교해 basic block에 할당된 bit이 변하지 않았다. 일단 bit의 값이 바뀌지 않았으면 이전 iteration의 그래프와 비교하게 되는데 이 과정에서 0→i edge와 같이 weight 값이 바뀐 edge들은 제거한다. 이런 edge들은 거의 변수 i와 같은 loop의 induction variable에 의해 생성된다. Edge의 제거로 그래프가 가진 정보의 일부를 잃게 되기는 하지만 이 것이 최적화된 코드의 정확성을 손상시키지는 않는다.

<iteration 3>에서 $BB1^{OUT}$은 수렴하게 된다. 그리고 다음 네번째 iteration에서 $BB2^{OUT}$와 $BB3^{OUT}$도 수렴하게 되어 분석이 끝나게 된다. 결과적으로 BB3의 check 명령이 실행될 때의 대소관계 그래프가 <iteration 3>의 $BB2^{OUT}$와 같은 모양이므로, 두 check 명령어는 제거될 수 있다.

3. 결과 및 결론

본 실험의 결과는 Intel Itanium 733MHz processor, 1GB RAM의 Hardware와 Windows XP 64-bit Edition Version 2003의 OS 위에서 운영되는 컴퓨터에서 얻었다. 본 알고리즘은 Intel의 *Open Runtime Platform JVM 위에서 구현된 VLaTTe* 컴파일러[6] 안에 구현되었다. 사용한 벤치마크는 spec jvm98이다.

함수의 basic block의 수가 많을 경우 (50개 이상) extended basic block범위에서 최적화를 했으며 array bounds check들의 개수가 작을 경우 최적화를 적용하지 않았다. 실험 결과 최적화를 적용한 모든 함수가 수렴하였으며 수렴하는데 필요한 평균 iteration 수는 표 2와 같다.

표 2

	함수 개수	최적화 적용	평균 iteration 수	성능향상
_200_check	371	33	2.94	
_201_compress	276	40	3.15	-0.2%
_202_jess	642	77	3.70	-4.1%
_209_db	313	41	3.10	-1.5%
_213_javac	883	67	3.12	-0.4%
_222_mpegaudio	398	100	3.41	5.2%
_227_mtrt	402	51	3.27	1.1%
_228_jack	530	40	2.85	0.6%

실험 결과는 기존의 data flow analysis에서 처리할 수 없었던 대소관계 그래프 문제를 우리의 알고리즘이 잘 수렴시키고 있음을 말해주고 있다. 또한 이 우리의 최적화를 통해 hot한 함수에서 bounds check 명령을 자주 수행하는 _222_mpegaudio의 경우 5.2%의 성능향상을 얻었고 평균(geo-mean) 0.1%의 성능향상을 얻었다.

_202_jess와 _209_db의 경우 성능저하가 있었다. 이것은 array bounds checks elimination으로 컴파일 시간이 는 것에 반해 실제로 최적화된 함수가 많이 실행되지 않은 것, array bounds check 명령어의 제거로 instruction을 schedule하는 시간이 늘어난 것 등이 영향을 주었을 것으로 보인다. 성능 저하가 큰 _202_jess의 경우 이 밖에도 더 많은 이유가 있을 것으로 추측되는데 앞으로 이에 대한 좀 더 깊이 있는 분석이 필요하다.

4. 참고문헌

[1] Pedro V. Artigas, Manish Gupta, Jose E. Moreira. *Automatic loop transformations and parallelization for Java.* International Conference on Supercomputing, Proceedings of the 14th international conference on Supercomputing. pp1-10. May 2000.

[2] Michał Cierniak, Guei-Yuan Lueh, James M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, Volume 35 Issue 5. May 2000.

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

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

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