Optimizing Array Bound Checks Using Flow Analysis

Rajiv Gupta, University of Pittsburgh

Presented by
Cheng Jiang, Daniel Jin, and Eric Hao
EECS 583 Advanced Compilers
University of Michigan

Agenda

- Introduction & Background
- Bound Check Optimizations
  - Local optimizations
  - Global optimizations
  - Loops optimizations
- Results & Benchmarks
- Conclusion

Introduction & Background

Array out-of-bounds errors

```
int arr[3] = {1, 2, 3};
for (int i = 0; i < 4; ++i) {
    cout << arr[i] << endl;
}

cout << arr[7] << endl;
```

Output:

```
1
2
3
9173820
81937218
```

Introduction

- *Bound check:* a boolean expression that checks lower and upper bounds of a subscript expression
- Compilers generate run-time checks for array bound violations
    - Overhead of checks is high
- Traditional optimizations ineffective in reducing overhead
- Production software usually doesn't include bound checks due to performance
    - Less reliability
- **Goal:** maintain security of correct execution at an acceptable run-time cost

Intuition and Approach

- *Eliminate* and *propagate* bound checks
  - Elimination analogous to constant folding and common subexpression elimination
  - Propagation is analogous to loop invariant code motion

- Ensure *reliability* of program is not affected
  - Errors detected before optimization still detected after optimization
  - Errors can be detected in different places

Bound Check Optimizations

Local Elimination

- Very simple local analysis to eliminate *identical* and *subsumed* checks
- *Identical checks* - (C = C') if check C precedes check C' and the variables used in the checks between C and C', then C' is eliminated
- *Subsumed checks with identical bounds* - one of two checks eliminated based on value of subscript expression

$$\begin{array}{lllll} MIN \le f(v) & \text{and} & MIN \le g(v) \equiv MIN \le f(v) & \text{if} & f(v) < g(v), \\ f(v) \le MAX & \text{and} & g(v) \le MAX \equiv g(v) \le MAX & \text{if} & f(v) < g(v). \end{array}$$

- *Subsumed checks with identical subscript expressions* - one of two checks eliminated based on value of bounds

$$\begin{array}{lll} MIN_1 \le f & \text{and} & MIN_2 \le f \equiv maximum(MIN_1, MIN_2) \le f, \\ f \le MAX_1 & \text{and} & f \le MAX_2 \equiv f \le minimum(MAX_1, MAX_2). \end{array}$$

Global Elimination

Redundancy

```
if ( ) then
      -- 10 ≤ i ≤ 50
      ....
else
      -- 20 ≤ i ≤ 100
      ....
fi
-- 5 ≤ i ≤ 200
 ....
```

Before Optimization

```
if ( ) then
      -- 10 ≤ i ≤ 50
      ....
else
      -- 20 ≤ i ≤ 100
      ....
fi
....
```

After Optimization

Modification

```
-- 5 ≤ i ≤ 200
if ( ) then
      -- 10 ≤ i ≤ 50
      ....
else
      -- 20 ≤ i ≤ 100
      ....
fi
```

Before Optimization

```
-- 10 ≤ i ≤ 100
if ( ) then
      -- 10 ≤ i ≤ 50
      ....
else
      -- 20 ≤ i ≤ 100
      ....
fi
```

After Modification

```
-- 10 ≤ i ≤ 100
if ( ) then
      -- i ≤ 50
      ....
else
      -- 20 ≤ i
      ....
fi
```

After Elimination

Notation

- Availability
  - Check *C* and point in program *P*
  - All paths leading to *P*, either *C* or a stronger check is performed
  - Forwards dataflow analysis

- Very Busy
  - Check *C* and point in program *P*
  - All paths from *P*, either *C* or a stronger check is performed
  - Backwards dataflow analysis

Modification Algorithm: Very Busy Checks

$$C\_IN[B] = C\_GEN[B] \vee backward(C\_OUT[B], B),$$

$$C\_OUT[B] = \bigwedge_{S \in Succ(B)} C\_IN[S], \quad \text{where } B \text{ is not the terminating block},$$

$$C\_OUT[B] = \emptyset, \text{ where } B \text{ is the terminating block;}$$

Modification Algorithm: Very Busy Checks

$$C\_IN[B] = C\_GEN[B] \vee backward(C\_OUT[B], B),$$

$$C\_OUT[B] = \bigwedge_{S \in Succ(B)} C\_IN[S], \quad \text{where } B \text{ is not the terminating block,}$$

$$C\_OUT[B] = \emptyset, \text{ where } B \text{ is the terminating block;}$$

Modification Algorithm: Very Busy Checks

$$C\_IN[B] = \boxed{C\_GEN[B]} \vee backward(C\_OUT[B], B),$$

$$C\_OUT[B] = \bigwedge_{S \in Succ(B)} C\_IN[S], \quad \text{where } B \text{ is not the terminating block},$$

$$C\_OUT[B] = \emptyset, \text{ where } B \text{ is the terminating block};$$

Modification Algorithm: Very Busy Checks

$$C\_IN[B] = C\_GEN[B] \vee \boxed{backward(C\_OUT[B], B),}$$

$$C\_OUT[B] = \bigwedge_{S \in Succ(B)} C\_IN[S], \quad \text{where } B \text{ is not the terminating block,}$$

$$C\_OUT[B] = \emptyset, \text{ where } B \text{ is the terminating block;}$$

Modification Algorithm: Very Busy Checks

$$C\_IN[B] = C\_GEN[B] \vee backward(C\_OUT[B], B),$$

$$C\_OUT[B] = \bigwedge_{S \in Succ(B)} C\_IN[S], \quad \text{where } B \text{ is not the terminating block,}$$

$$C\_OUT[B] = \emptyset, \text{ where } B \text{ is the terminating block;}$$

Modification Algorithm: Modifying Checks

- A check *C* is modified if
  - Another check *C'* that is very busy at the point immediately following *C* and *C'* subsumes *C*
  - Replace *C* with *C'*

```
-- 5 ≤ i ≤ 200
if ( ) then
      -- 10 ≤ i ≤ 50
      ....
else
      -- 20 ≤ i ≤ 100
      ....
fi
```

Before Optimization

```
-- 10 ≤ i ≤ 100
if ( ) then
      -- 10 ≤ i ≤ 50
      ....
else
      -- 20 ≤ i ≤ 100
      ....
fi
```

After Modification

Notation

- Availability
  - Check *C* and point in program *P*
  - All paths leading to *P*, either *C* or a stronger check is performed

- Very Busy
  - Check *C* and point in program *P*
  - All paths from *P*, either *C* or a stronger check is performed

Redundancy Algorithm: Available Checks

$$C\_OUT[B] = C\_GEN[B] \vee forward(C\_IN[B], B),$$

$$C\_IN[B] = \bigwedge_{P \in Pred(B)} C\_OUT[P], \text{ where } B \text{ is not the initial block},$$

$$C\_IN[B] = \varnothing, \text{ where } B \text{ is the initial block.}$$

Redundancy Algorithm: Available Checks

$$C\_OUT[B] = C\_GEN[B] \vee forward(C\_IN[B], B),$$

$$C\_IN[B] = \bigwedge_{P \in Pred(B)} C\_OUT[P], \text{ where } B \text{ is not the initial block,}$$

$$C\_IN[B] = \varnothing, \text{ where } B \text{ is the initial block.}$$

Redundancy Algorithm: Available Checks

$$C\_OUT[B] = \boxed{C\_GEN[B]} \vee forward(C\_IN[B], B),$$

$$C\_IN[B] = \bigwedge_{P \in Pred(B)} C\_OUT[P], \text{ where } B \text{ is not the initial block,}$$

$$C\_IN[B] = \varnothing, \text{ where } B \text{ is the initial block.}$$

Redundancy Algorithm: Available Checks

$$C\_OUT[B] = C\_GEN[B] \vee \boxed{forward(C\_IN[B], B)},$$

$$C\_IN[B] = \bigwedge_{P \in Pred(B)} C\_OUT[P], \text{ where } B \text{ is not the initial block,}$$

$$C\_IN[B] = \varnothing, \text{ where } B \text{ is the initial block.}$$

Redundancy Algorithm: Available Checks

$$C\_OUT[B] = C\_GEN[B] \vee forward(C\_IN[B], B),$$

$$C\_IN[B] = \bigwedge_{P \in Pred(B)} C\_OUT[P], \text{ where } B \text{ is not the initial block,}$$

$$C\_IN[B] = \emptyset, \text{ where } B \text{ is the initial block.}$$

Modification Algorithm: Eliminating Checks

- A check *C* is eliminated if
  - Another check *C'* that is available at the point immediately preceding *C* and *C'* subsumes *C* or *C'* is identical to *C*

```
-- 5 ≤ i ≤ 200
if ( ) then
      -- 10 ≤ i ≤ 50
      ....
else
      -- 20 ≤ i ≤ 100
      ....
fi
```

Before Optimization

```
-- 10 ≤ i ≤ 100
if ( ) then
      -- 10 ≤ i ≤ 50
      ....
else
      -- 20 ≤ i ≤ 100
      ....
fi
```

After Modification

```
-- 10 ≤ i ≤ 100
if ( ) then
      -- i ≤ 50
      ....
else
      -- 20 ≤ i
      ....
fi
```

After Elimination

Loop Optimizations

Observation: propagation moves the checks to an earlier point in the code

Errors detected will be at a point different from the errors in the original code

We can propagate *loop invariant* bound checks out of loops!

Loop Optimizations

Observation: propagation moves the checks to an earlier point in the code

Errors detected will be at a point different from the errors in the original code

We can propagate *loop invariant* bound checks out of loops!

Can we do better?

Consider an Example

```
repeat:
    if (...):
        -- 10 <= i <= 100
        -- 1 <= j <= 10
    else:
        -- 5 <= i <= 50
        -- 1 <= j <= 10
    endif
```

Consider an Example

```
repeat:
    if (...):
        -- 10 <= i <= 100
        -- 1 <= j <= 10
    else:
        -- 5 <= i <= 50
        -- 1 <= j <= 10
    endif
```

Consider an Example

```
-- 1 <= j <= 10
repeat:
    if (...):
        -- 10 <= i <= 100

    else:
        -- 5 <= i <= 50

    endif
```

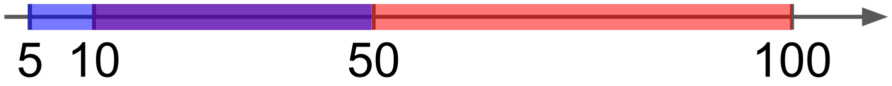
Consider an Example

2 bound checks
per iteration

```
-- 1 <= j <= 10
repeat:
    if (...):
        -- 10 <= i <= 100

    else:
        -- 5 <= i <= 50

    endif
```

Consider an Example

We can hoist a (combined) weaker condition

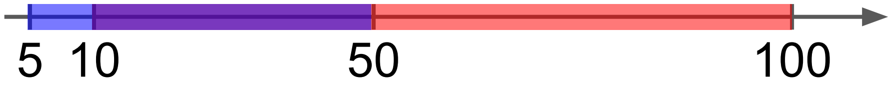



```
-- 1 <= j <= 10
repeat:
    if (...):
        -- 10 <= i <= 100

    else:
        -- 5 <= i <= 50

    endif
```

Consider an Example

The stronger conditions must be left in the loop

1 bound check per iteration




```
-- 5 <= i <= 100
-- 1 <= j <= 10
repeat:
    if (...):
        -- 10 <= i

    else:
        -- i <= 50

    endif
```

Consider an Example

```
-- 5 <= i <= 100
-- 1 <= j <= 10
repeat:
    if (...):
        -- 10 <= i
    else:
        -- i <= 50
    endif
```

How to Propagate?

Recall: we want to propagate bound checks that are invariant or can be hoisted with minor modifications

Intuitions:

- If a basic block (BB) dominates all loop exits, then its bound checks may be hoisted
- It is also possible to hoist some other bound checks as well

Analogous to instruction hoisting

The innermost loops are processed first, the outermost loops are processed last

Propagation Algorithm

Assumption: We have identified the loop, and have computed UD chains and dominator sets

1. Identify propagation candidates
   - Invariants
   - Increasing values (comparing to a lower bound)
   - Decreasing values (comparing to an upper bound)
   - Loops with increment / decrement of one
2. Hoist checks: from conditionally executed BB to unconditionally executed BB
3. Propagate checks out of the loop

Experimental Results

Experimental Results

Table I. Effects of Bound Check Optimization

Program	UNOPT	LELIM +	GELIM +	PROP =	Total deleted
BUBBLE	59,400	39,600 +	9,900 +	9,900 =	59,400 ≈ 100%
QUICK	271,184	72,784 +	10,014 +	54,347 =	137,145 ≈ 51%
QUEEN	13,784	2,288 +	1,748 +	1,778 =	5,814 ≈ 42%
TOWERS	556,262	261,944 +	97,844 +	0 =	359,788 ≈ 65%
LLOOP6	20,160	8,064 +	0 +	12,096 =	20,160 ≈ 100%
FFT	37,414	24,568 +	0 +	5,930 =	30,498 ≈ 82%
MATMUL	1,043,200	640,000 +	256,000 +	147,200 =	1,043,200 ≈ 100%
PERM	80,624	10,078 +	0 +	7,240 =	73,384 ≈ 91%

UNOPT = total number of bound checks before optimization; LELIM = number of checks eliminated by local elimination; GELIM = number of checks eliminated by global elimination; PROP = number of checks eliminated by propagation.

Thank You!
Questions?

Extra

Modification Algorithm: *backward* Function

```
backward( C_OUT[B], B ) {
    S = ∅
    for each check C ∈ C_OUT[B] do
        case C of
        lb ≤ v :
            case AFFECT(B ,v) of
                unchanged:  S = S ∪ {lb ≤ v }
                increment:  /* the check is killed */
                decrement:  S = S ∪ {lb ≤ v }
                multiply:   /* the check is killed */
                div>1:      S = S ∪ {lb ≤ v }
                div<1:      /* the check is killed */
                changed:    /* the check is killed */
            end case
```

Modification Algorithm: *backward* Function

```
backward( C_OUT[B], B ) {
    S = Ø
    for each check C ∈ C_OUT[B] do
        case C of
        lb ≤ v:
            case AFFECT(B, v) of
                unchanged:  S = S ∪ {lb ≤ v}
                increment:  /* the check is killed */
                decrement:  S = S ∪ {lb ≤ v}
                multiply:   /* the check is killed */
                div>1:      S = S ∪ {lb ≤ v}
                div<1:      /* the check is killed */
                changed:    /* the check is killed */
            end case
```

Modification Algorithm: *backward* Function

```
backward( C_OUT[B], B ) {
    S = ∅
    for each check C ∈ C_OUT[B] do
        case C of
        lb ≤ v:
            case AFFECT(B,v) of
                unchanged:  S = S ∪ {lb ≤ v}
                increment:  /* the check is killed */
                decrement:  S = S ∪ {lb ≤ v}
                multiply:   /* the check is killed */
                div>1:      S = S ∪ {lb ≤ v}
                div<1:      /* the check is killed */
                changed:    /* the check is killed */
            end case
```

20 <= i

lb   v

Modification Algorithm: *backward* Function

```
backward( C_OUT[B], B ) {
    S = ∅
    for each check C ∈ C_OUT[B] do
        case C of
        lb ≤ v:
            case AFFECT(B,v) of
                unchanged:  S = S ∪ {lb ≤ v}
                increment:  /* the check is killed */
                decrement:  S = S ∪ {lb ≤ v}
                multiply:   /* the check is killed */
                div>1:      S = S ∪ {lb ≤ v}
                div<1:      /* the check is killed */
                changed:    /* the check is killed */
            end case
```

Modification Algorithm: *backward* Function

```
backward( C_OUT[B], B ) {
    S = Ø
    for each check C ∈ C_OUT[B] do
        case C of
        lb ≤ v:
            case AFFECT(B ,v) of
                unchanged:  S = S ∪ {lb ≤ v}
                increment:  /* the check is killed */
                decrement:  S = S ∪ {lb ≤ v}
                multiply:   /* the check is killed */
                div>1:      S = S ∪ {lb ≤ v}
                div<1:      /* the check is killed */
                changed:    /* the check is killed */
            end case
```

Redundancy Algorithm: *forward* Function

```
forward( C_IN[B], B ) {
    S = ∅
    for each check C ∈ C_IN[B] do
        case C of
        lb ≤ v :
            case AFFECT(B ,v) of
                unchanged:  S = S ∪ {lb ≤ v }
                increment:  S = S ∪ {lb ≤ v }
                decrement:  /* the check is killed */
                multiply:   S = S ∪ {lb ≤ v }
                div>1:      /* the check is killed */
                div<1:      S = S ∪ {lb ≤ v }
                changed:    /* the check is killed */
            end case
```

Redundancy Algorithm: *forward* Function

```
forward( C_IN[B], B ) {
    S = ∅
    for each check C ∈ C_IN[B] do
        case C of
        lb ≤ v :
            case AFFECT(B ,v) of
                unchanged:  S = S ∪ {lb ≤ v }
                increment:  S = S ∪ {lb ≤ v }
                decrement:  /* the check is killed */
                multiply:   S = S ∪ {lb ≤ v }
                div>1:      /* the check is killed */
                div<1:      S = S ∪ {lb ≤ v }
                changed:    /* the check is killed */
            end case
```

Redundancy Algorithm: *forward* Function

```
forward( C_IN[B], B ) {
    S = ∅
    for each check C ∈ C_IN[B] do
        case C of
        lb ≤ v:
            case AFFECT(B,v) of
                unchanged:  S = S ∪ {lb ≤ v}
                increment:  S = S ∪ {lb ≤ v}
                decrement:  /* the check is killed */
                multiply:   S = S ∪ {lb ≤ v}
                div>1:      /* the check is killed */
                div<1:      S = S ∪ {lb ≤ v}
                changed:    /* the check is killed */
            end case
```

Redundancy Algorithm: *forward* Function

```
forward( C_IN[B], B ) {
    S = ∅
    for each check C ∈ C_IN[B] do
        case C of
        lb ≤ v:
            case AFFECT(B,v) of
                unchanged:  S = S ∪ {lb ≤ v}
                increment:  S = S ∪ {lb ≤ v}
                decrement:  /* the check is killed */
                multiply:   S = S ∪ {lb ≤ v}
                div>1:      /* the check is killed */
                div<1:      S = S ∪ {lb ≤ v}
                changed:    /* the check is killed */
            end case
```

Algorithm to Hoist Checks Out of Loops

```
hoist {
    ND = {n : block n does not dominate all loop exits}
    for each block n do
        C(n) = {c : at the entry to n we can assert that candidate check c will be executed in n }
    od
    change = true
    while change do
        change = false
        for each block n ∋ Succ(n) ∩ ND ≠ ∅ ∧ n is the unique predecessor of nodes in Succ(n) do
            prop = ⋀_{S ∈ Succ(n)} C(S)
            if prop ≠ ∅ then
                change = true
                hoist checks in prop to n
                for each check c ∈ prop do
                    if c ε S , S ∈ Succ(n) then eliminate c from S fi
                od
            fi
        od
    od
}
```

Propagation Out of Loops: Unknown Bounds

```
while
    -- MIN(a) ≤ i ≤ MAX(a)
    -- MIN(a) ≤ j ≤ MAX(a)
    a[i] ≠ a[j]
do
    i = i + 1
    j = j - 1
od
```

Before Propagation

```
-- MIN(a) ≤ i, j ≤ MAX(a)
while
        -- i ≤ MAX(a)
        -- MIN(a) ≤ j
        a[i] ≠ a[j]
do
        i = i + 1
        j = j - 1
od
```

After Propagation

Propagation Out of Loops: Known Bounds

```
for i ← min to max do
    if (inc) then -- MIN(a) ≤ i ≤ MAX(a)
        sum ← sum + a[i]
    else -- MIN(a) ≤ i ≤ MAX(a)
        sum ← sum - a[i]
    fi
od
```

Before Propagation

```
-- MIN(a) ≤ min, max ≤ MAX(a)
for i ← min to max do
        if (inc) then
                sum ← sum + a[i]
        else sum ← sum - a[i]
        fi
od
```

After Propagation