Optimizing Array Bound Checks Using Flow Analysis

Rajiv Gupta

Presented by Lauren Biernacki, Colton Holoday, Yirui Liu, Andrew McCrabb

Bounds Checking

Python

```python
array = [0, 1, 2]
for i in range(5):
    print (array[i]),

```

```
0 1 2
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print (array[i]),
IndexError: list index out of range
```

C++

```cpp
#include <iostream>
using namespace std;

int array [] = {0, 1, 2};
int main(){
    for (int i=0; i<10; i++)
        cout << array[i] << “ ”;
    return 0;
}
```

```
0 1 2 0 0 0 0 0 852851984 32534

...Program finished with exit code 0
```

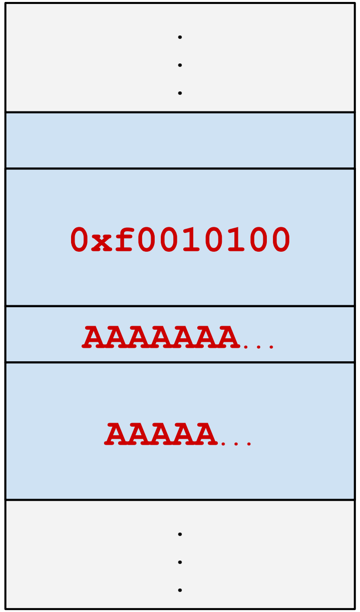
Stack Buffer Overflow Vulnerability

C++

```
void target() {
  printf("You overflowed successfully, gg");
  exit(0);
}

void vulnerable(char* str1) {
  char buf[5];
  strcpy(buf, str1);
}

int main() {
  vulnerable("AAAAAAAAAAAA\xf0\x01\x01\x00");
  printf("This only prints in normal control
  flow");
}
```

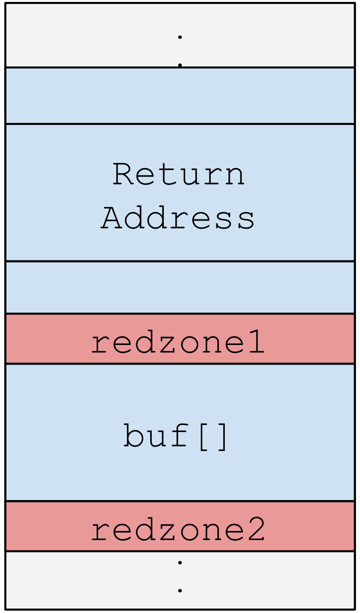


Address Sanitizer (ASan)

- An open source tool created by Google, included in LLVM
- Used to identify memory errors, including buffer overflows

Instruments code to:

  - Create poisoned *redzones* around stack objects
  - Check *shadow memory* before each memory access



Address Sanitizer (ASan)

- An open source tool created by Google, included in LLVM
- Used to identify memory errors, including buffer

“AddressSanitizer achieves efficiency without sacrificing comprehensiveness.”

73% slowdown, **337%** increased memory usage

In

  - Create poisoned *redzones* around stack objects
  - Check *shadow memory* before each memory access

Compile Time Optimizations for ASan

- Using dataflow techniques, such as the work done by Gupta, it should be possible to optimize ASan's checks
- This could be applied to other memory safety protections, or simply bounds checking in general

```
if (f) {
  a[i] = ...;
}
else {
  a[i] = ...;
}
a[i] = ...; //Redundant
```

Fully redundant checks

```
//Enough to check a[i] here
if (f) {
  a[i] = ...;
}
else {
  a[i] = ...;
}
```

Hoisting bounds checks

Optimizing Array Bounds Checks

1. **Local elimination**
2. **Global elimination**
   a. Elimination algorithm
   b. Further optimization
3. **How to deal with loops**
4. **Evaluation**

Local Elimination

-- *MIN(a)* ≤ i+1 ≤ MAX(a)
temp ← a[i+1]
-- *MIN(a)* ≤ i+1 ≤ MAX(a)
-- *MIN(a)* ≤ i ≤ MAX(a)
a[i+1] ← a[i]
-- *MIN(a)* ≤ i ≤ MAX(a)
a[i] ← temp

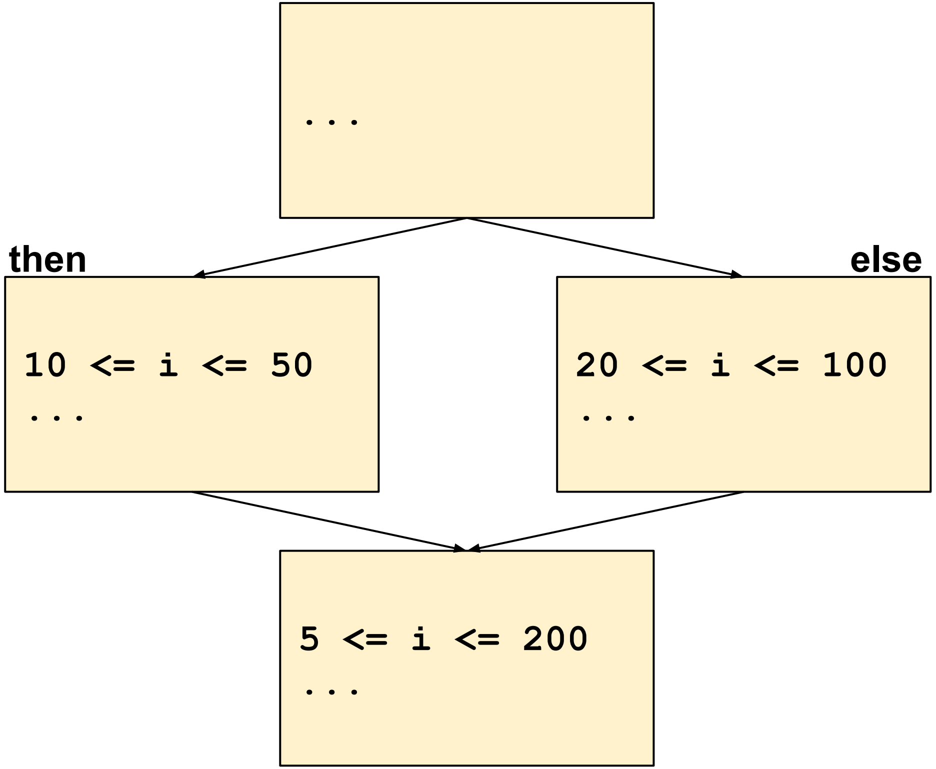
Before Optimization

-- *MIN(a)* ≤ i, i+1 ≤ MAX(a)
temp ← a[i+1]
a[i+1] ← a[i]
a[i] ← temp

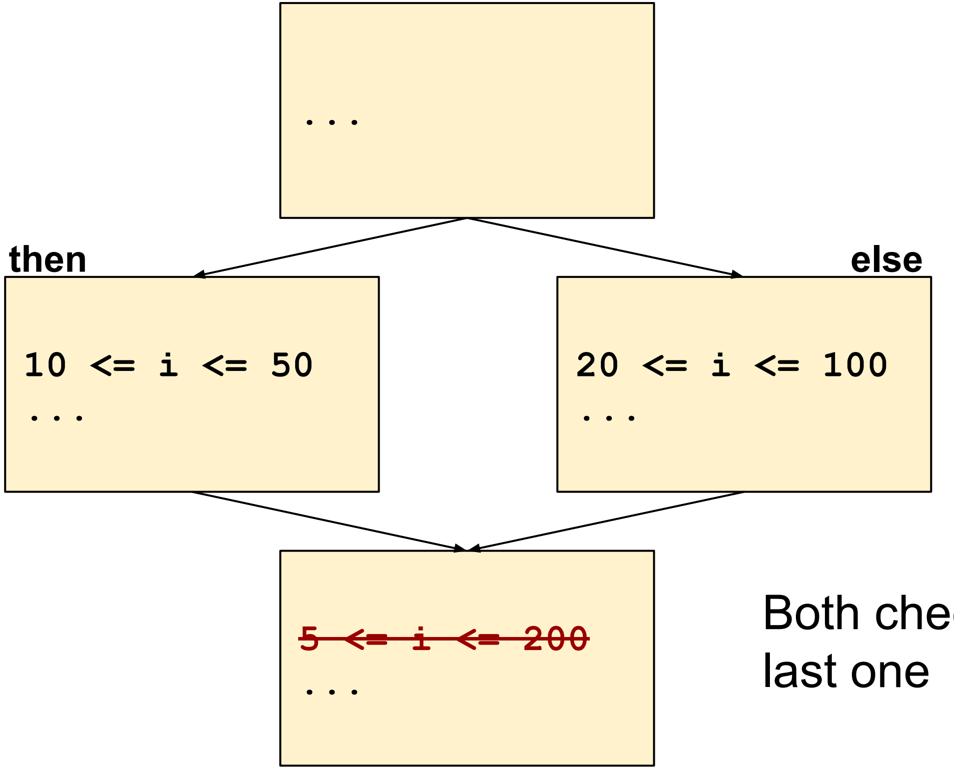
After Optimization

Fig. 1. Local elimination of bound checks.

Global Elimination

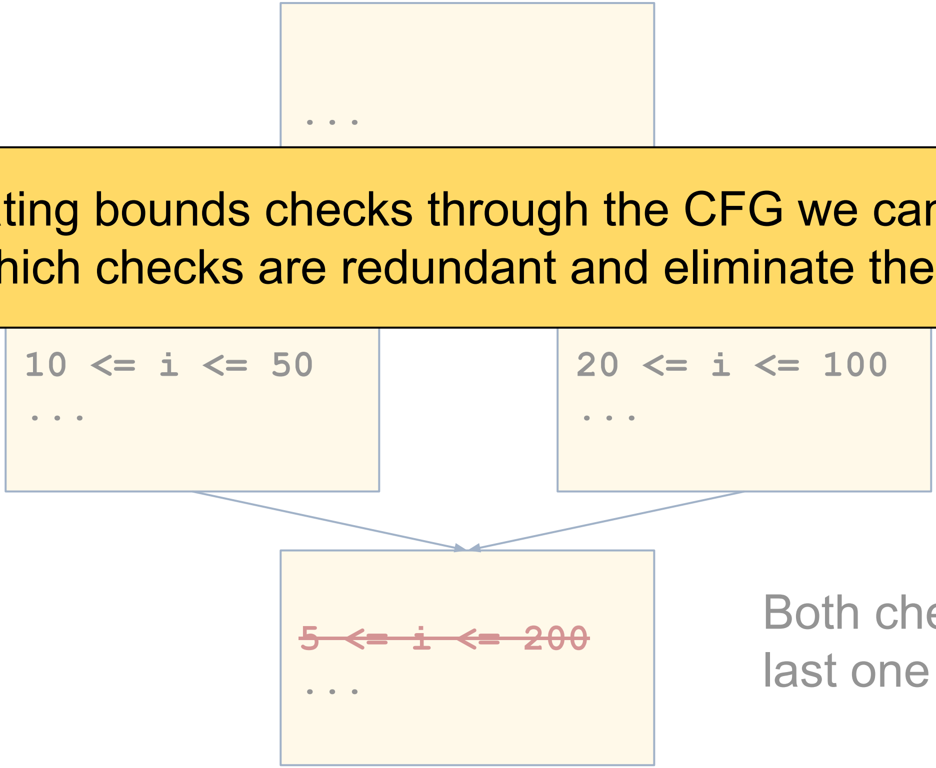


Global Elimination



Both checks ***subsume*** the last one

Global Elimination



By propagating bounds checks through the CFG we can determine which checks are redundant and eliminate them

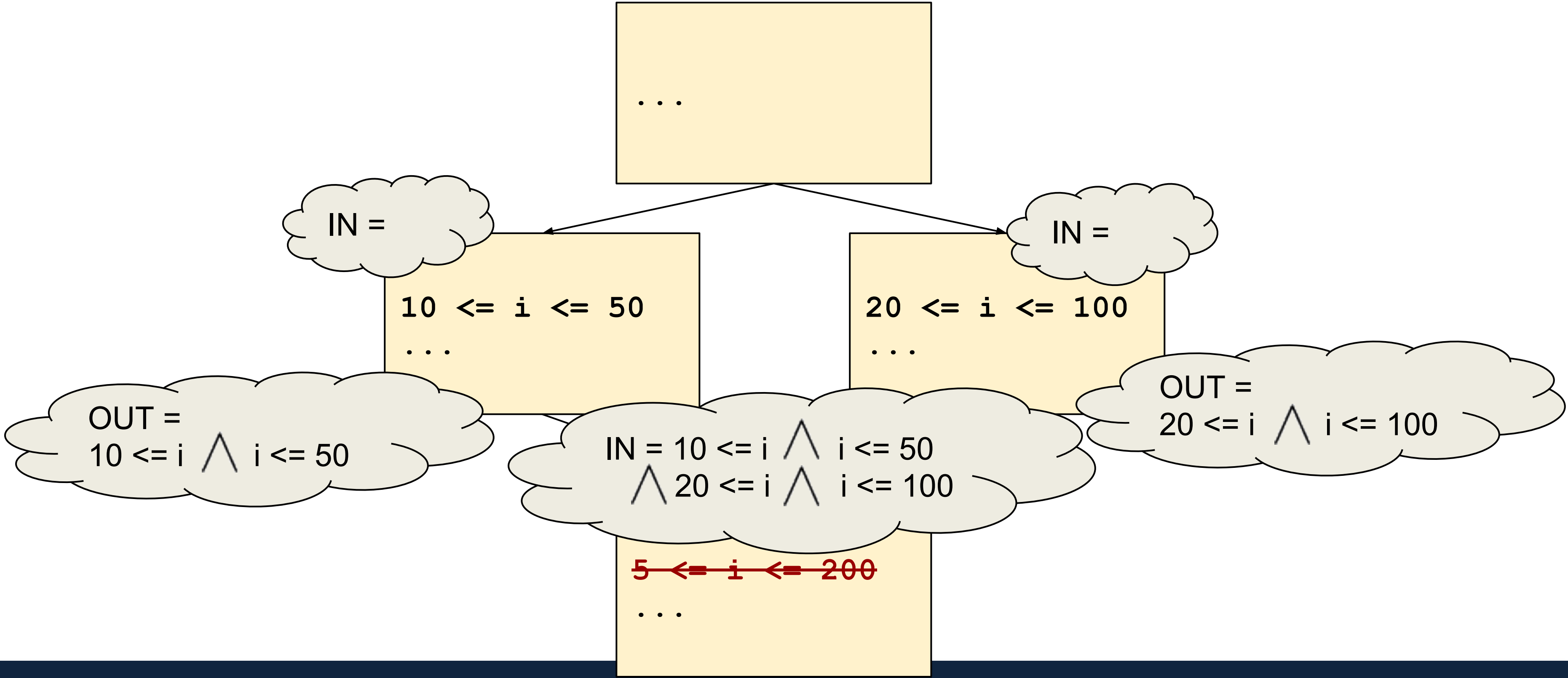
Both checks ***subsume*** the last one

Formulating a Dataflow Analysis

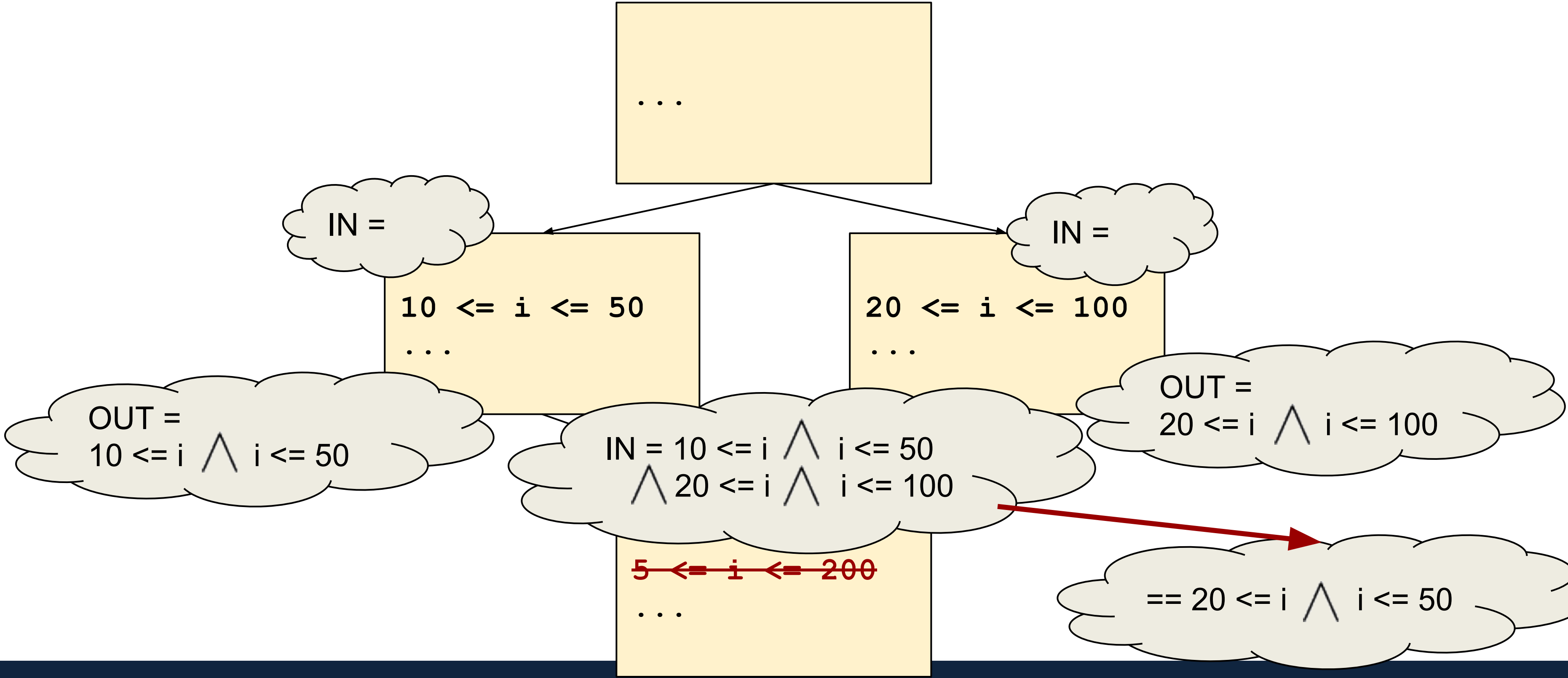
Available Checks ~ *"A bound check C is available at a program point p if it is guaranteed that, **along each path** **leading to point p**, either C is performed or a check that **subsumes** C is performed."*

Key Characteristics		
All Paths	**Forward**	$\bigwedge$ instead of $\bigcap$

Eliminating Redundant Checks



Eliminating Redundant Checks



Let's Formalize the Analysis

How kill is handled

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

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

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

Handling KILL Set

```
20 <= i <= 100
i = i + 1
...
```

OUT =
20 <= i $\wedge$ ~~i <= 100~~

- Monotonic operations can retain checks through kill filter

```
forward( C_IN[B], B ) {
    S = Ø
    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
        v ≤ ub:
            case AFFECT(B,v) of
                unchanged:  S = S ∪ {v ≤ ub}
                increment:  /* the check is killed */
                decrement:  S = S ∪ {v ≤ ub}
                multiply:   /* the check is killed */
                div>1:      S = S ∪ {v ≤ ub}
                div<1:      /* the check is killed */
                changed:    /* the check is killed */
            end case
                    .
                    .
                    .
```

Optimizing Array Bounds Checks

1. **Local elimination**
2. **Global elimination**
   a. Elimination algorithm
   b. Further optimization
3. **How to deal with loops**
4. **Evaluation**

Eliminating Redundant Checks

-- $5 \leq i \leq 200$
if ( ) then
    -- $10 \leq i \leq 50$
    ....
else
    -- $20 \leq i \leq 100$
    ....
fi
Before Optimization

-- $10 \leq i \leq 100$
if ( ) then
    -- $10 \leq i \leq 50$
    ....
else
    -- $20 \leq i \leq 100$
    ....
fi
After Modification

-- $10 \leq i \leq 100$
if ( ) then
    -- $i \leq 50$
    ....
else
    -- $20 \leq i$
    ....
fi
After Elimination

Formulating a Dataflow Analysis

Very-busy Checks ~ *"A bound check C is very busy at a program point p if it is guaranteed that, **along each path** **starting at point p**, either C is performed or a check that **subsumes** C is performed."*

Key Characteristics		
All Paths	**Backward**	$\bigwedge$ instead of $\bigcap$

Let's Formalize the Analysis

Different to available checks here...

Compute the set of very-busy checks at all points in the program

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

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

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

$$S_1 \land S_2 \land \cdots \land S_n \qquad \bigwedge$$

$$= \{C : \forall S_i, 1 \le i \le n, (C \in S_i \lor \exists C' \in S_i \land C' \text{ subsumes } C)\},$$

$$S_1 \lor S_2 \lor \cdots \lor S_n$$

$$= \{C : (\exists S_i, 1 \le i \le n, C \in S_i) \land (\nexists\, C' \in S_i, 1 \le i \le n, C' \text{ subsumes } C)\}.$$

Let's Formalize the Analysis

Compute the set of very-busy checks at all points in the program

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

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

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

$$S_1 \wedge S_2 \wedge \cdots \wedge S_n$$
$$= \{C : \forall S_i, 1 \le i \le n, (C \in S_i \vee \exists C' \in S_i \wedge C' \text{ sub}$$

$$S_1 \vee S_2 \vee \cdots \vee S_n$$
$$= \{C : (\exists S_i, 1 \le i \le n, C \in S_i) \wedge (\nexists C' \in S_i, 1 \le i \le n, C$$

Different to available checks here...

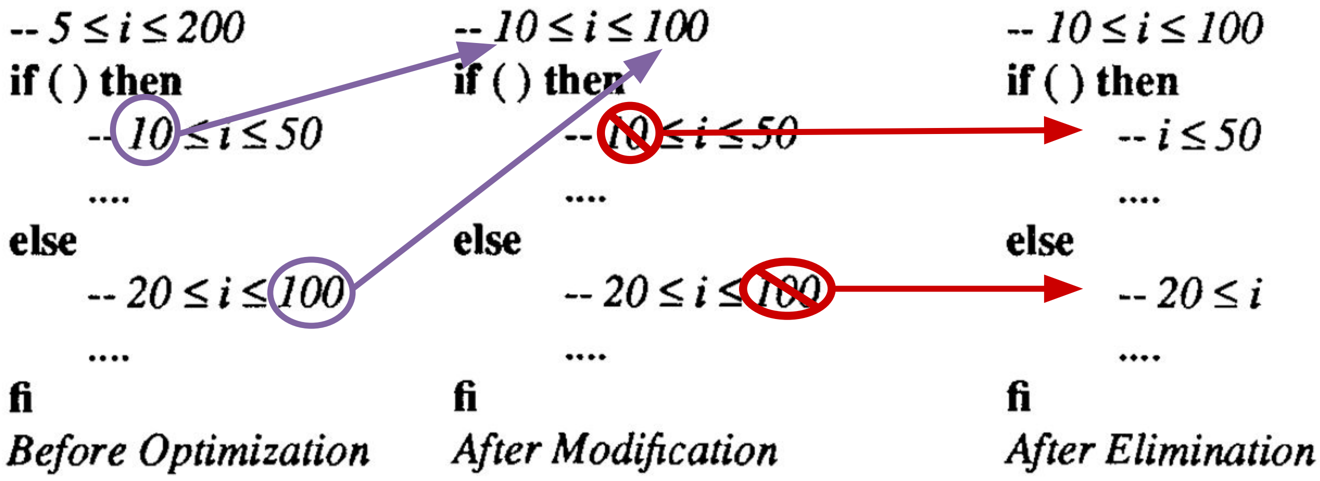
IN =
20 <= i $\bigwedge$ ~~i <= 100~~

```
i = i - 1
...
```

OUT =
20 <= i $\bigwedge$ i <=100

Modifying Checks

If a check C’ is very busy at the point immediately following the check C, and C’ subsumes C, then C can be replaced by C’.



Optimizing Array Bounds Checks

1. **Local elimination**
2. **Global elimination**
   a. Elimination algorithm
   b. Further optimization
3. **How to deal with loops**
4. **Evaluation**

Propagating the Checks out of the Loops

Example:

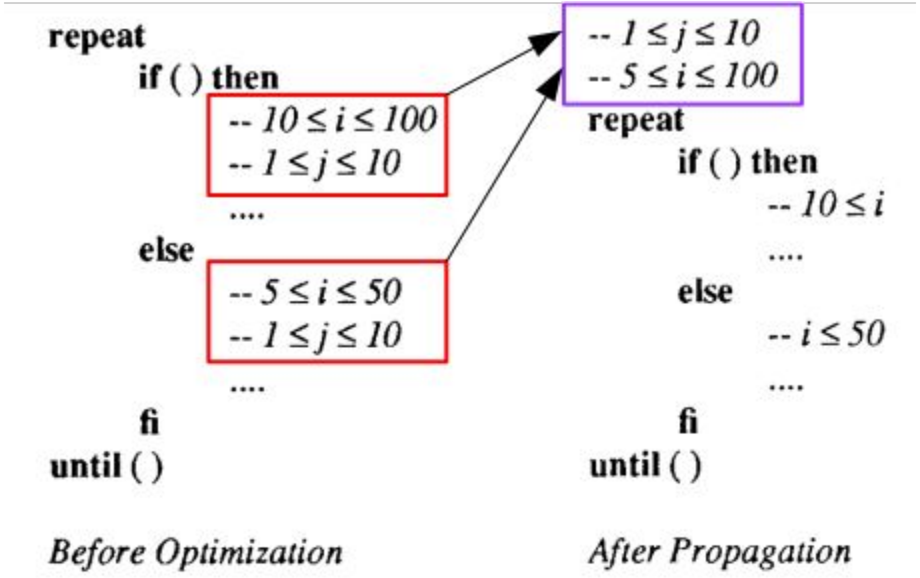


Fig. 7. Propagation of bound checks.

Propagating the Checks out of the Loops

Goal:

- Reduce the number of times the checks are executed

Algorithm:

- Identify the candidates (e.g. loop invariants) for propagation
  - Use-def Chain
  - Dominator Sets
- Check hoisting
- Propagate the checks out of the loop

Propagating the Checks out of the Loops

Another Example:

```
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

Fig. 10. Propagation out of loops with known bounds for subscript variables.

Optimizing Array Bounds Checks

1. **Local elimination**
2. **Global elimination**
   a. Elimination algorithm
   b. Further optimization
3. How to deal with loops
4. **Evaluation**

Experimental Evaluation

>80% of Bounds Checks Eliminated on Average

Effect of Bounds Check Optimization						
	UNOPT	L-elim	G-elim	Prop	**Total Deleted**	**% 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	**17,318**	**21%**

Implications of This Work

1993

- Compilers came with “array bound check” flag
- Too much performance and memory overhead
- Gupta publishes this paper

Today

- Address Sanitizer used to provide comprehensive memory checks
- Still comes with high overheads
- We can apply these three optimizations from Gupta to reduce overheads

Conclusion

Comprehensive Bounds Checking

- Useful for Testing & Debugging
- ***73% slowdown; 337% memory overhead***

Pre-process bounds checks to eliminate many runtime checks

- Local & Global Elimination; Loop Propagation
- ***>80% Runtime bounds checks eliminated***

Questions

“Optimizing Array Bound Checks Using Flow Analysis”

Backup Slides

Address Sanitizer (ASan)

- An open source tool created by Google, included in LLVM
- Used to identify memory errors, including buffer overflows
- Consists of two parts:
  - ***Code Instrumentation*** — Creates poisoned redzones around stack and global objects, instruments code to check shadow memory before each memory access
  - ***Run-time Library*** — Augments `malloc()` and `free()` to apply the above protections to the heap

Address Sanitizer (ASan)

Before:

```
void foo() {
  char a[32];
  ...
  *address = ...;
  ...
  return;
}
```

After:

```
void foo() {
  char redzone1[32];
  char a[32];
  char redzone3[32];
  int  *shadow = MemToShadow(redzone1);
  // poison redzones
  shadow[0] = 0xffffffff;
  shadow[1] = 0x00000000;
  shadow[2] = 0xffffffff;
  ...
  if (IsPoisoned(address)) {
    ReportError(address);
  }
  *address = ...;
  ...
  // unpoison all
```

Address Sanitizer (ASan)

Before:

```
void foo() {
  char a[32];
  ...
```

After:

```
void foo() {
  char redzone1[32];
  char a[32];
```

```
  ...
  if (IsPoisoned(address)) {
    ReportError(address);
  }
  *address = ...;
  ...
  // unpoison all
```

"AddressSanitizer achieves efficiency without sacrificing comprehensiveness."

73% slowdown, **337%** increased memory usage

Main Insights

Elimination:

- Eliminate redundant checks at compile time
- Analogous to constant folding and common subexpression elimination

Propagation:

- Propagate bound checks out of loops to reduce the number of run-time checks
- Analogous to loop invariant code motion optimization

Algorithm for Eliminating Redundant Checks

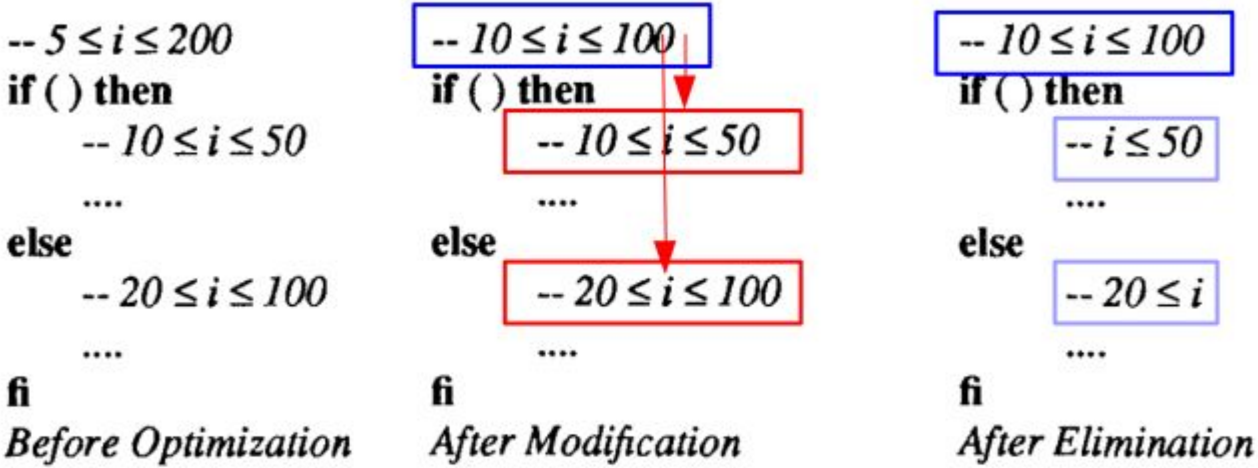


Fig. 3. Global elimination by modification of bound checks.