

ABCD: Eliminating Array-Bounds Checks on Demand

Rastislav Bodík

U of Wisconsin

Rajiv Gupta

U of Arizona

Vivek Sarkar

IBM TJ Watson

recent experiments by **Denis Gopan**, U of Wisconsin

Motivation: type safety

2

Pro: type-safe programs don't "crash"

Con: some violations checked at run time

Direct cost: *executing* the checks

checks are frequent, expensive

Indirect cost: *preventing* optimization

checks block code motion of side-effect instructions

Our goal: safety without performance penalty

How? remove redundant checks

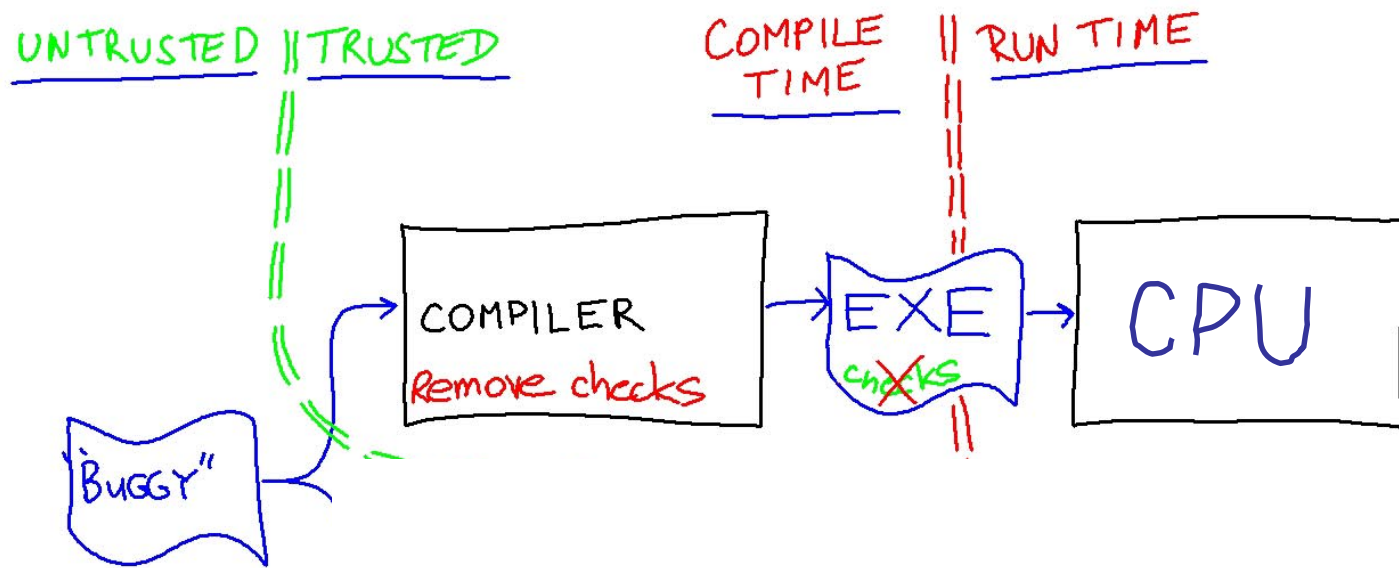
Talk outline

3

- Why remove bounds checks?
safety without performance penalty
- The need for dynamic optimization
 - existing optimizers not suitable
 - an ideal dynamic optimizer
- ABCD
- Experiments
- Summary

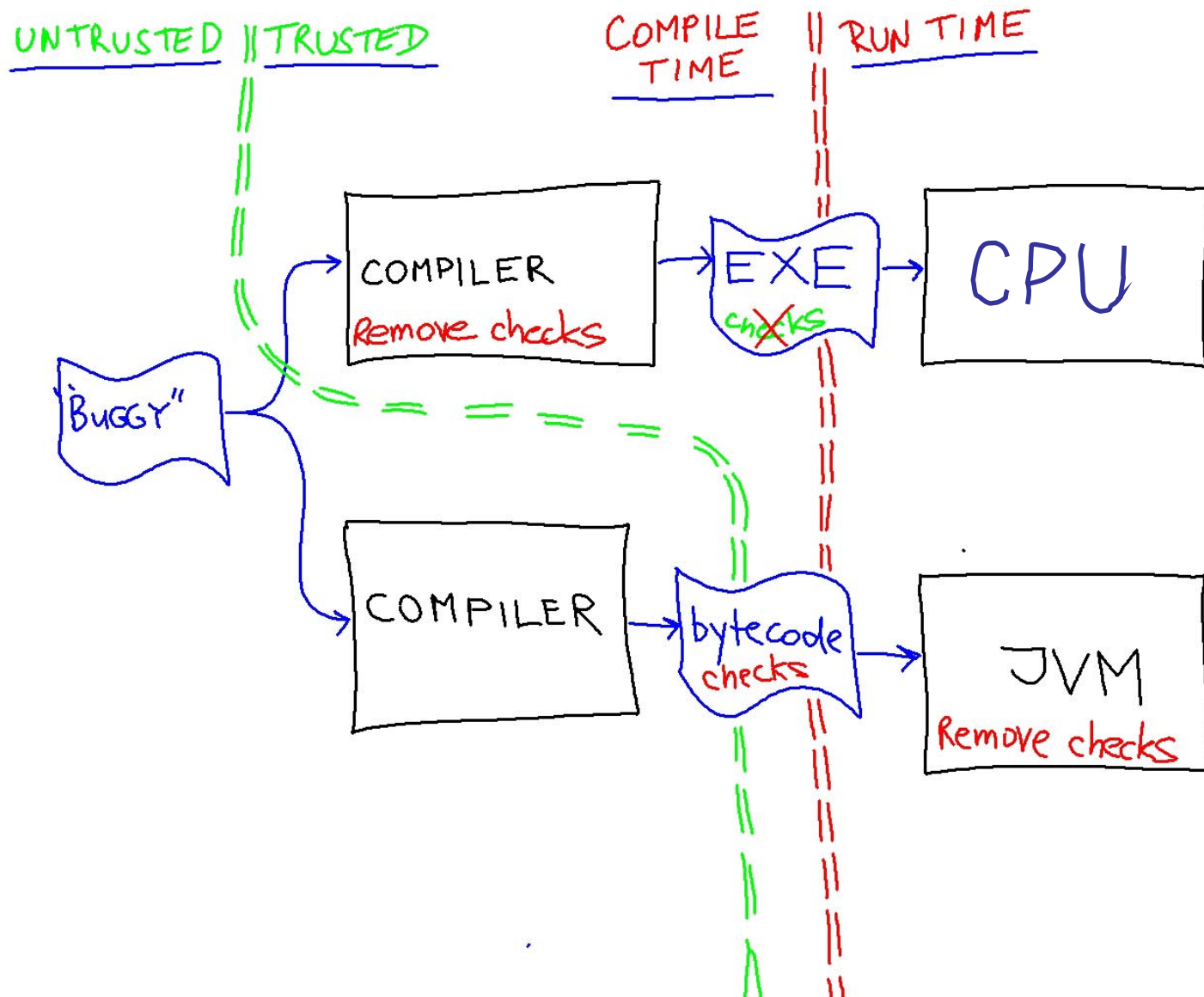
When to optimize the checks?

4



When to optimize the checks?

5



Existing check optimizers

6

- Emphasis: precision
 - **goal:** *all* checks removed = *statically* type-safe
 - **theorem prover:** [Necula, Lee], [Xu, Miller, Reps]
 - **range propagation:** [Harrison, Patterson]
 - **types:** [Xi, Phенning]
- Properties
 - too heavy-weight
 - limited notion of control flow:
 - how to add profile feedback?

An ideal dynamic optimizer?

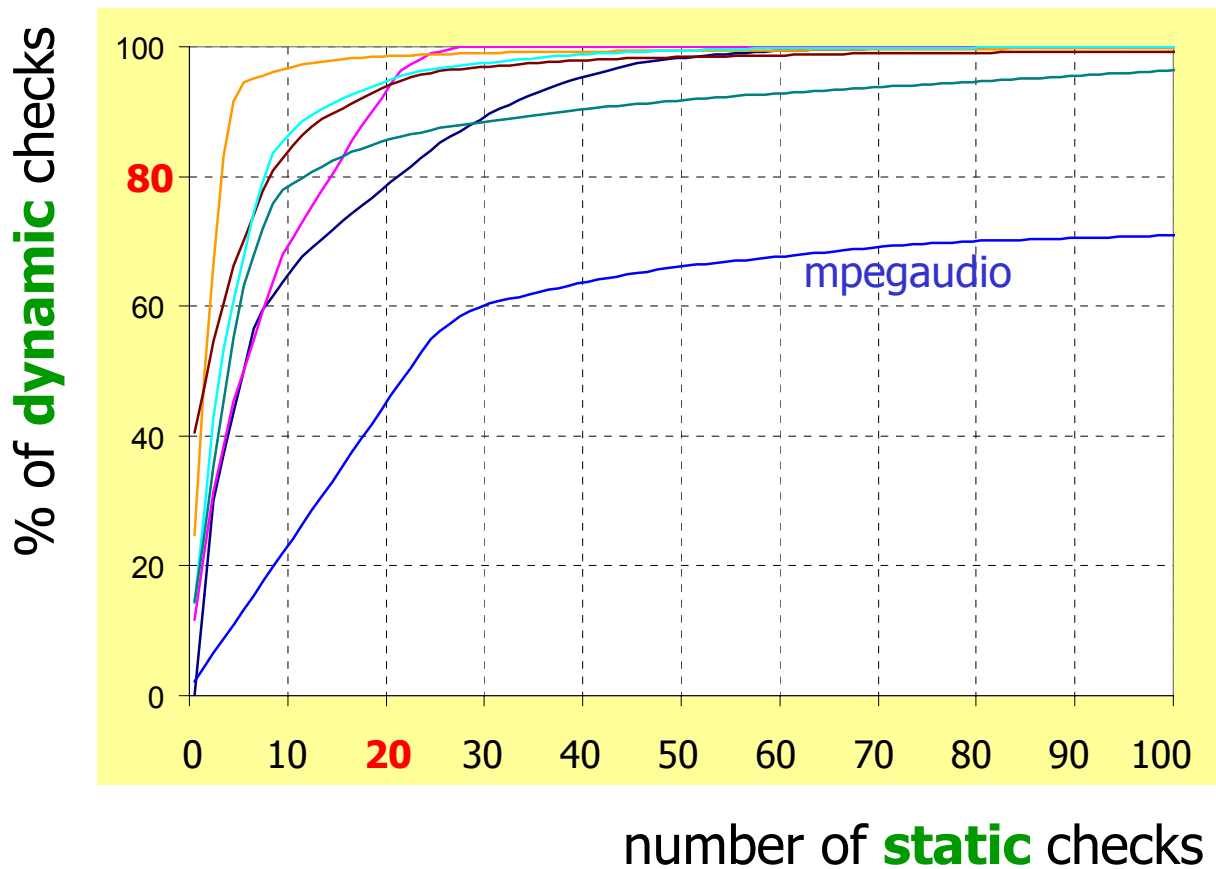
7

- **A balance between power and economy**
 - powerful just enough
 - minimize analysis work
 - reduce IR overhead
 - **Scalable**
 - optimize only **hot** checks
 - **no** whole-program analysis
- ☞ only common cases
 - ☞ efficient IR
 - ☞ reuse the IR
 - ☞ profile-directed
 - + demand-driven
 - ☞ use “local” info
 - + insert (cold) checks

Why optimize on demand?

👉 optimize only the **few hot** checks

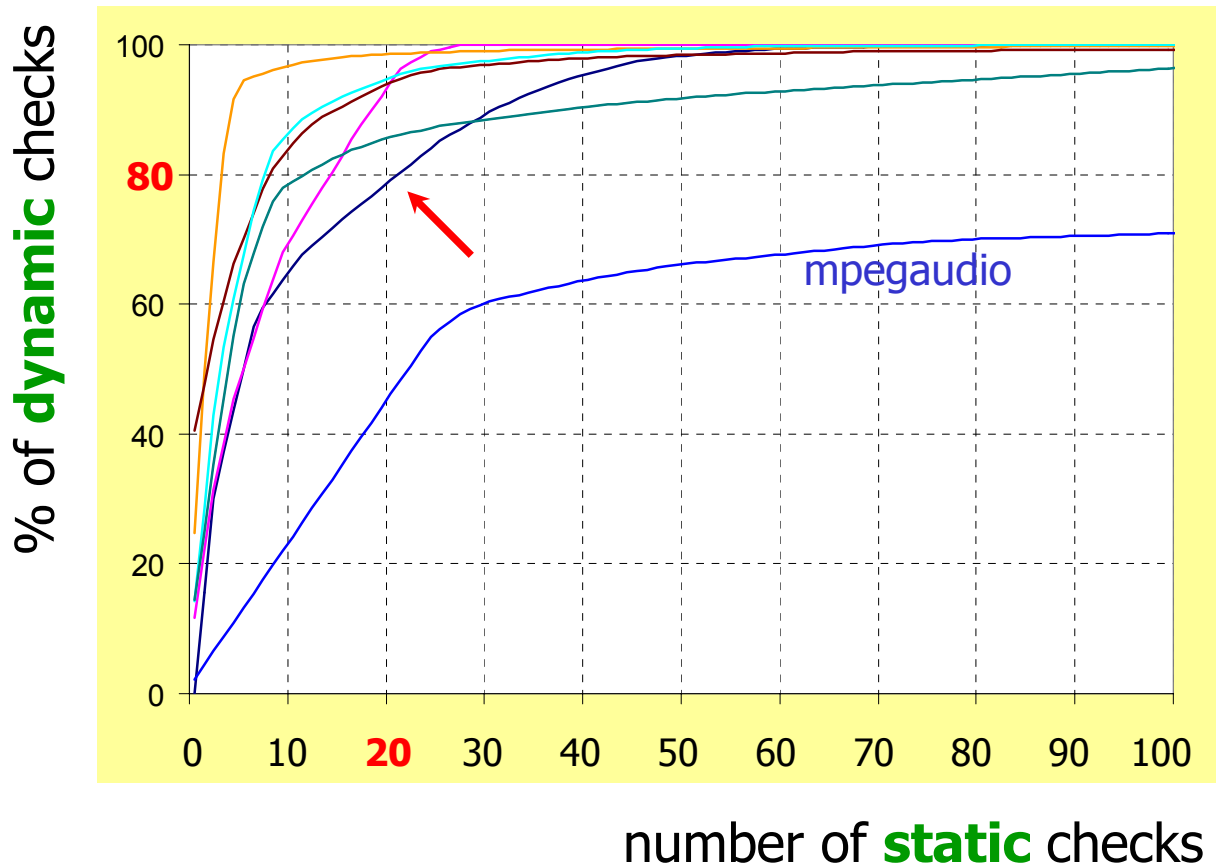
checks analyzed
(not necessarily removed)



Why optimize on demand?

👉 optimize only the **few hot** checks

checks analyzed
(not necessarily removed)



Talk outline

9

- Motivation
- An ideal dynamic optimizer
- ABCD “tutorial”
 - 1. Simple** ... standard SSA
 - 2. Full** ... extended SSA
 - 3. PRE** ... profile-directed
 - 4. ABCDE** ... work in progress
- Experiments
- Summary

High-level algorithm

10

```
for each hot array access  $A[i]$  do  
  -- optimize upper-bound check  
  ABCD(  $i < A.length$  )  
  
  -- optimize lower-bound check  
  ABCD(  $0 \leq i$  )  
  
end for
```

High-level algorithm

10

for each hot array access $A[i]$ **do**

this
talk

```
-- optimize upper-bound check  
ABCD(  $i < A.length$  )
```

```
-- optimize lower-bound check  
ABCD(  $0 \leq i$  )
```

end for

1. Simple ABCD

11

```
i ← A.length
while ( ) {
  --i
  ..A[i]..
}
```

1. Simple ABCD

11

```
i ← A.length
while ( ) {
  --i
  ..A[i]..
}
```

Simple ABCD = SSA + shortest path

1. Simple ABCD

12

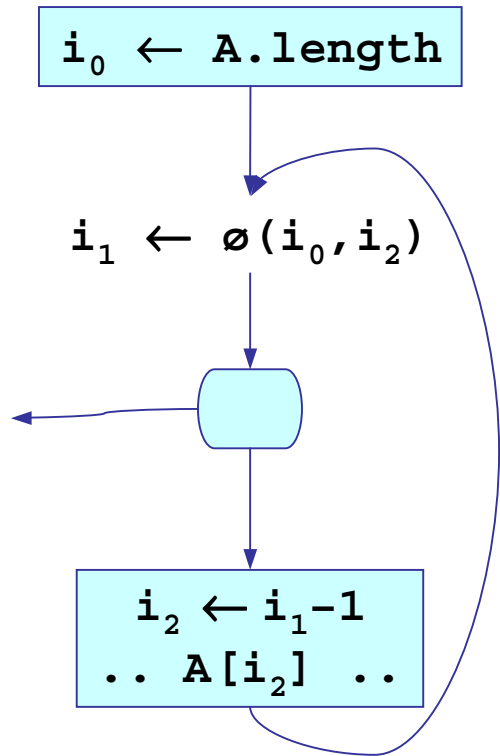
1. build SSA

2. label edges with constraints

3. analyze $A[i_k]$:

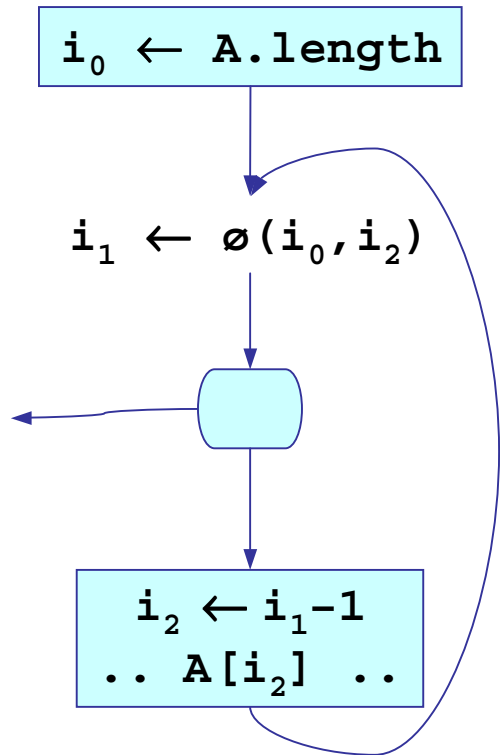
is $i_k < A.length$ always true?

Simple ABCD



Simple ABCD

13



- **A.length**

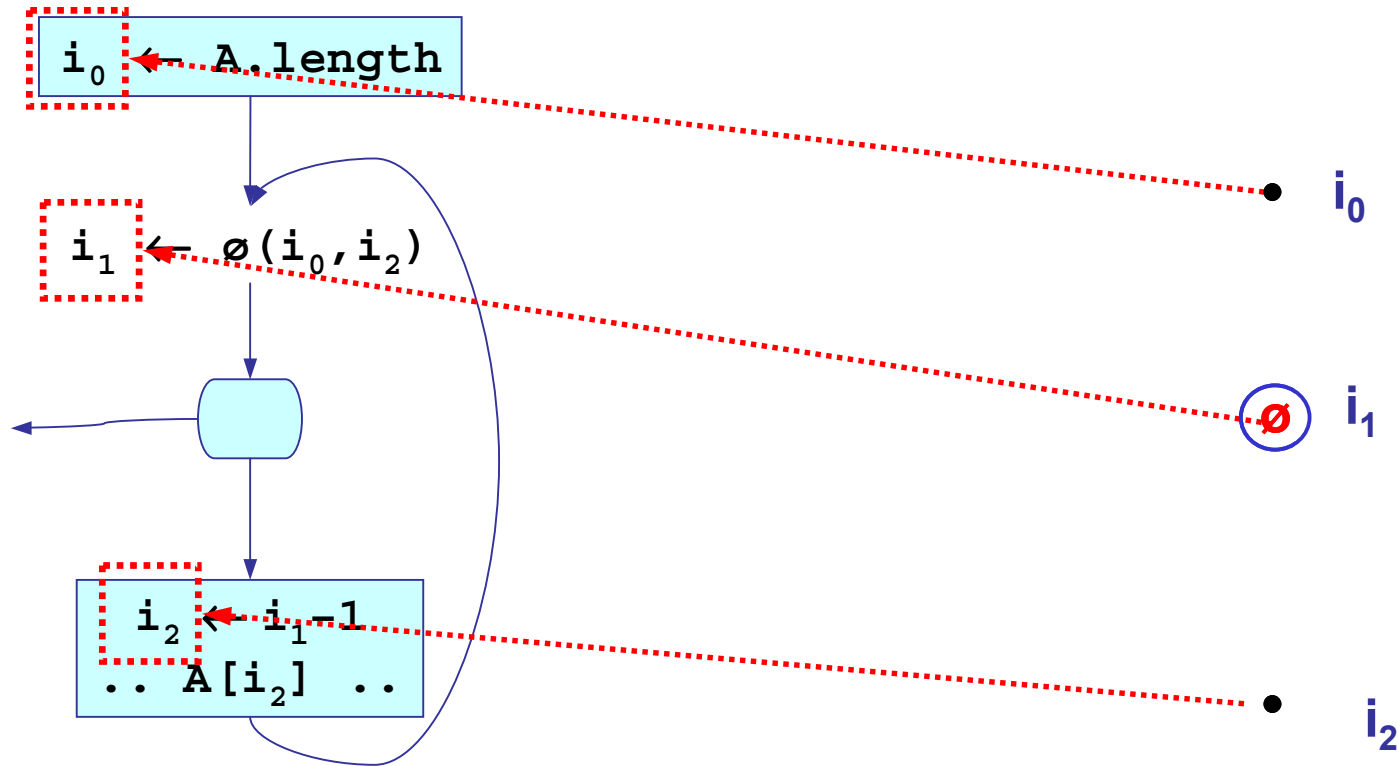
- i_0

- \emptyset i_1

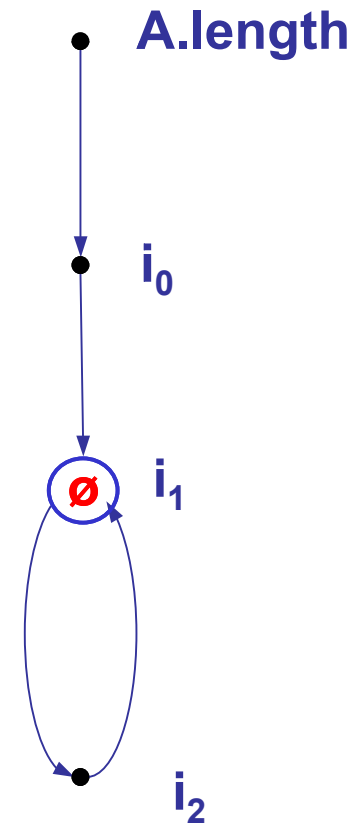
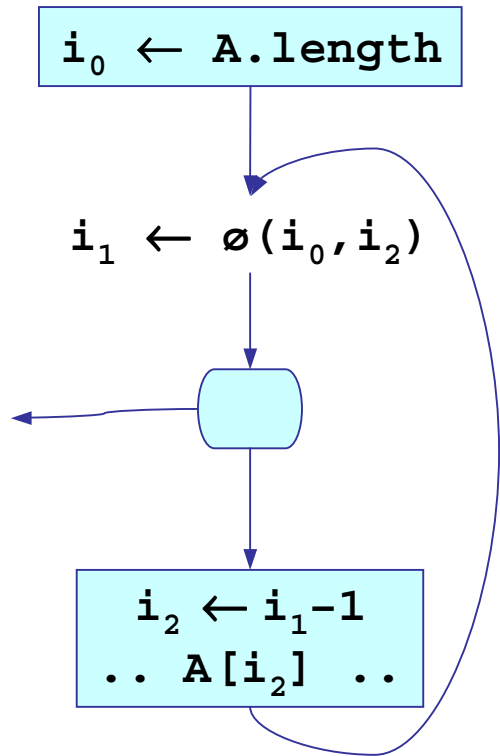
- i_2

Simple ABCD

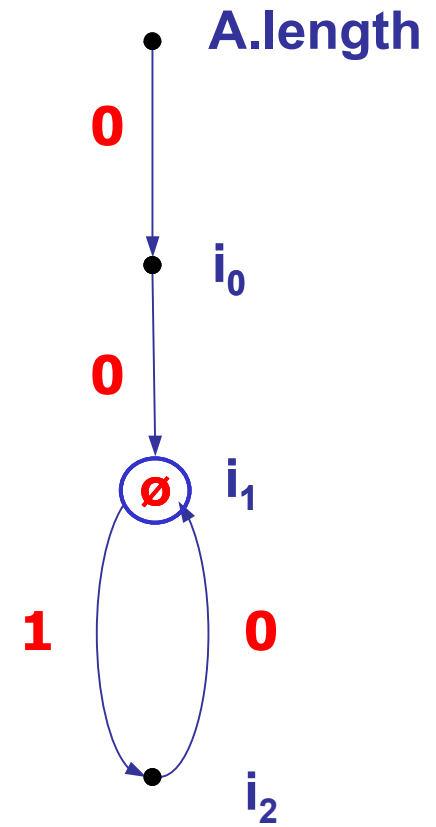
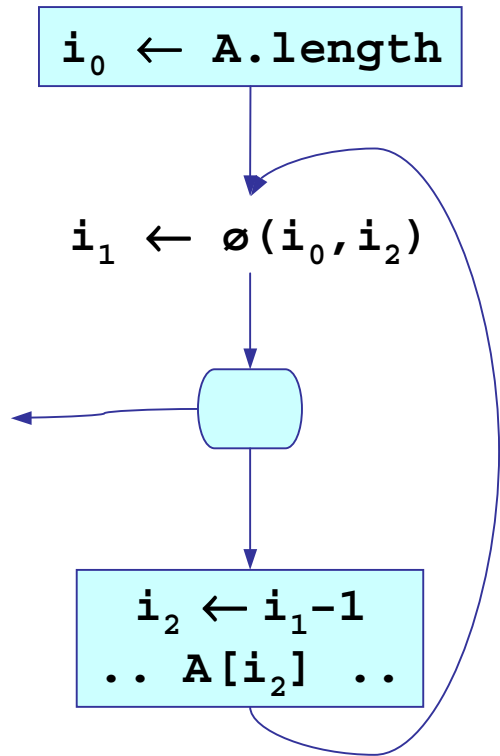
- A.length



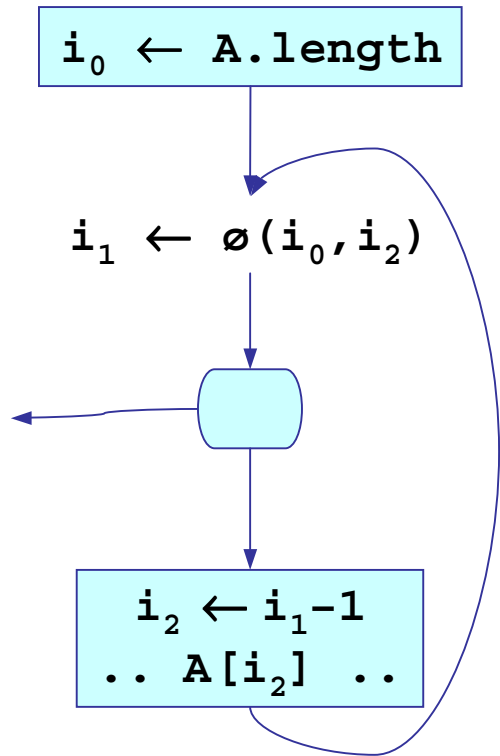
Simple ABCD



Simple ABCD



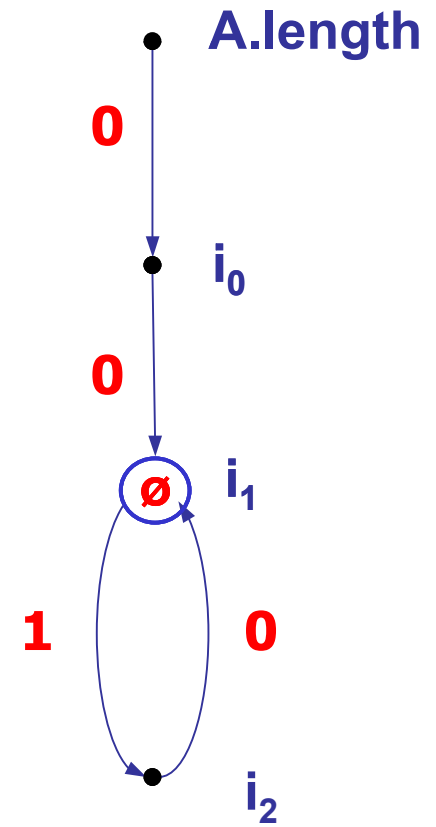
Simple ABCD



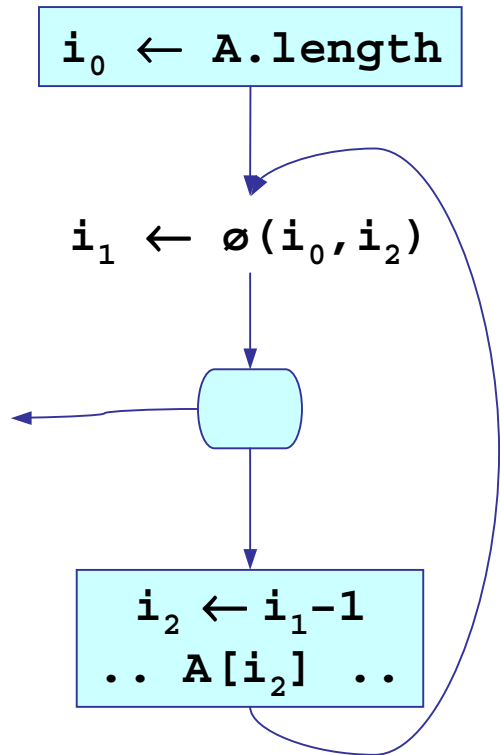
$$i_0 \leq A.length - 0$$

$$i_1 \leq i_0 - 0$$

$$i_2 \leq i_1 - 1$$



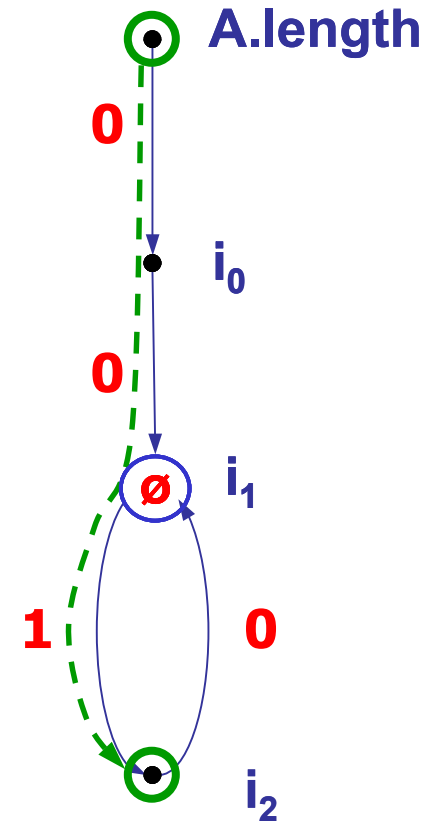
Simple ABCD



$$i_0 \leq A.length - 0$$

$$i_1 \leq i_0 - 0$$

$$i_2 \leq i_1 - 1$$



$$\text{weight}(A.length \rightarrow i_2) = 1 \Rightarrow i_2 \leq A.length - 1$$

1. Simple ABCD

14

1. build SSA

2. label edges with constraints

3. analyze $A[i_k]$:

input: a bounds check $i_k < A.length$

algorithm: find shortest path p from $A.length$ to i_k

output: check is redundant if $weight(p) > 0$

2. Full ABCD

15

```
for (i=0; i < A.length; i++)  
    ..A[i]..
```

2. Full ABCD

15

```
for (i=0; i < A.length; i++)  
    ..A[i]..
```

Full ABCD = SSA⁺⁺ + “shortest” path

2. Full ABCD

16

1. build **extended SSA**:

naming of standard SSA not fine-grain enough

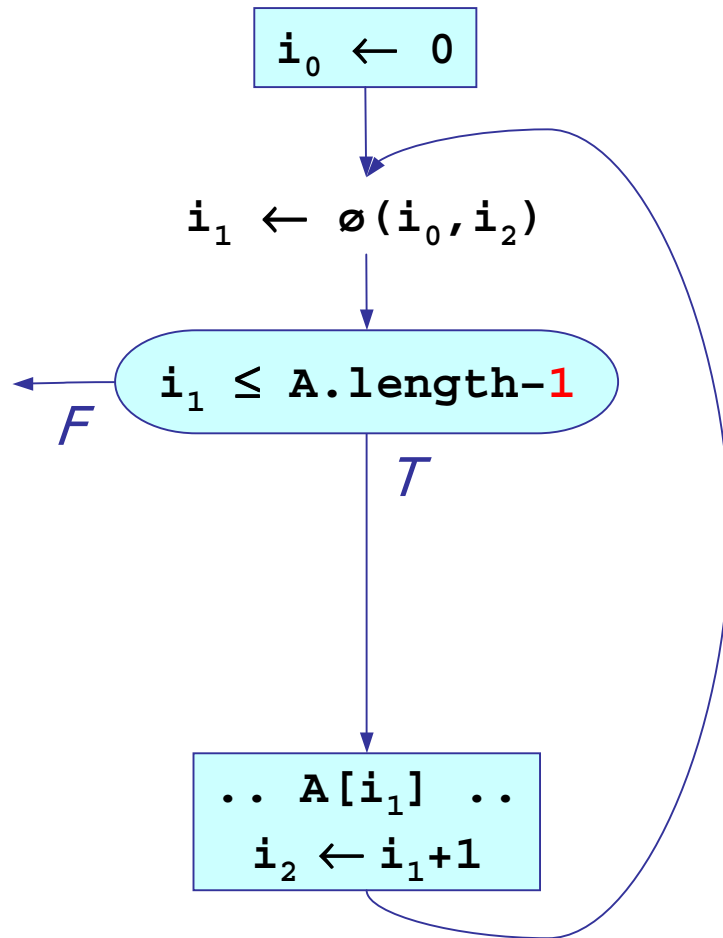
⇒ add dummy π -assignments

2. label edges with constraints

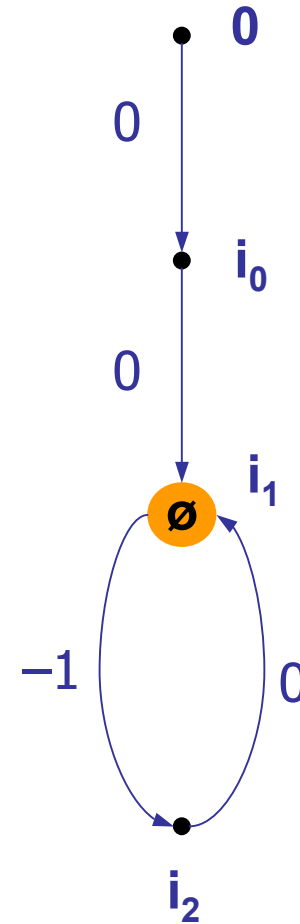
3. analyze:

shortest path → optimal path in a hyper-graph

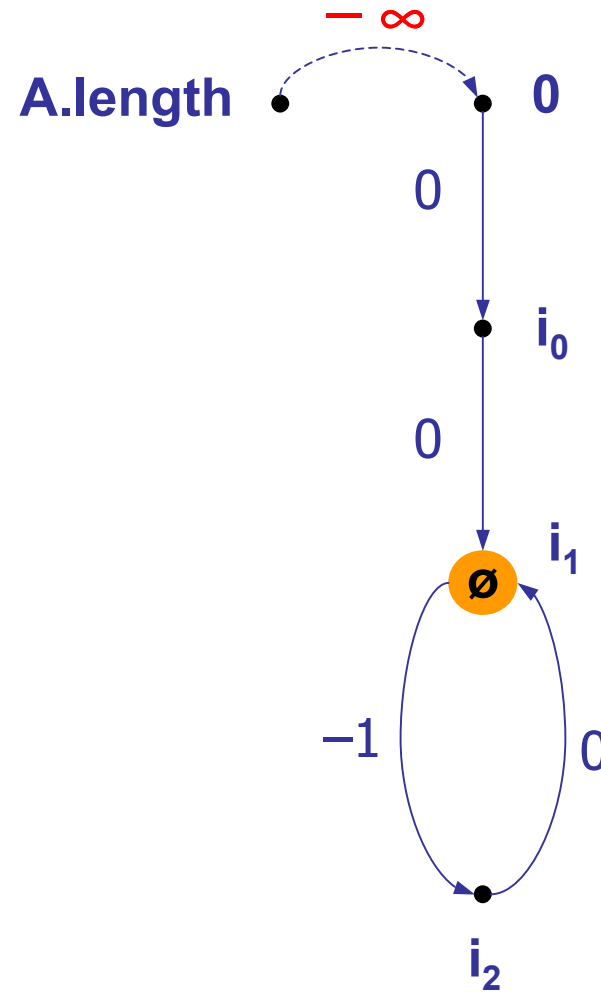
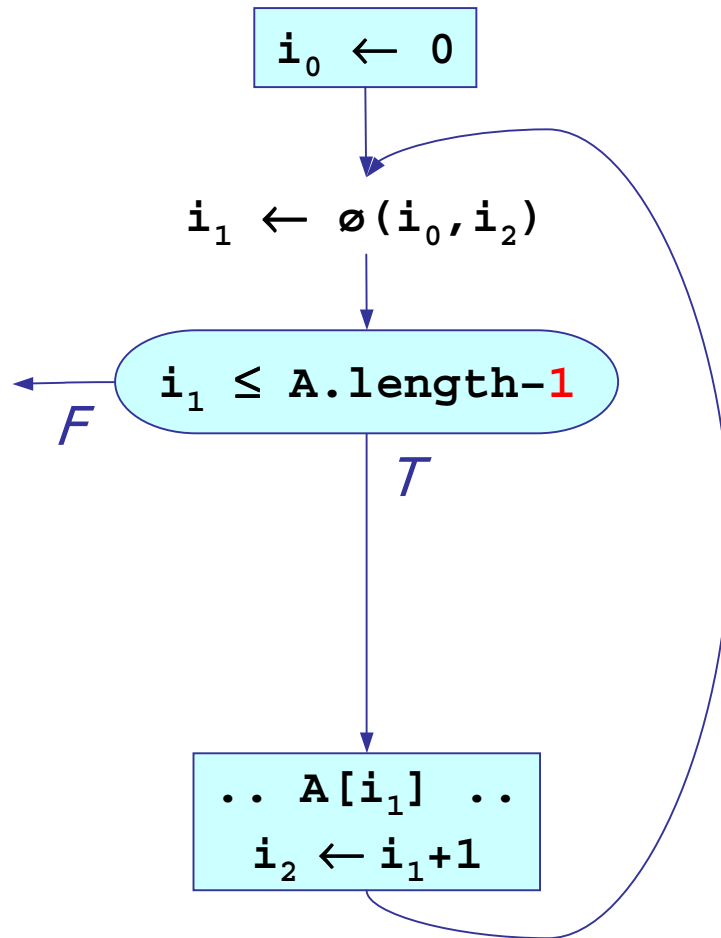
Extending the SSA form



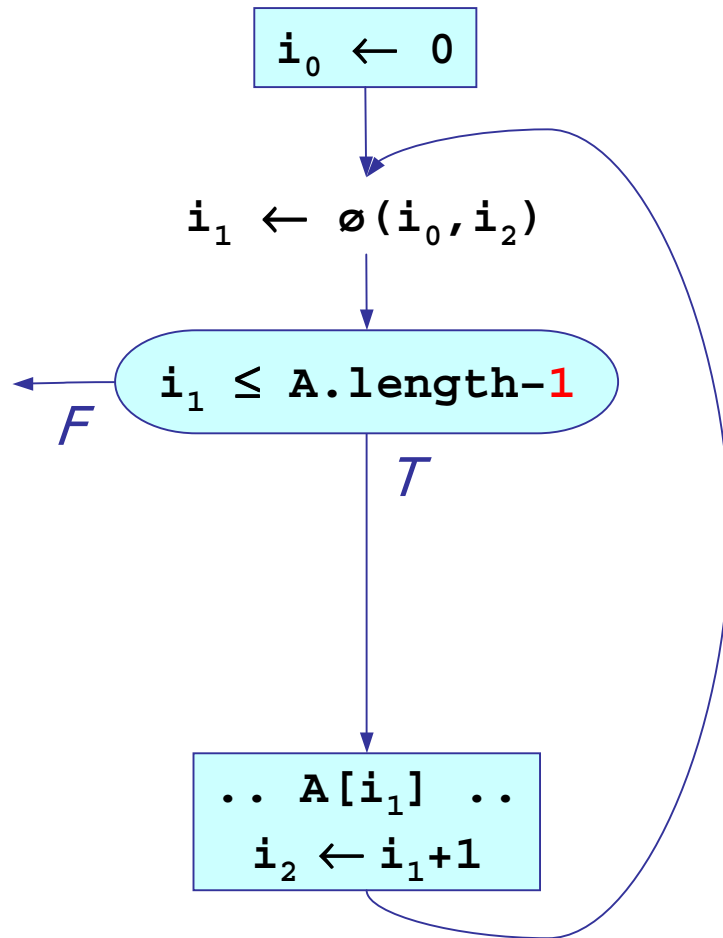
A.length •



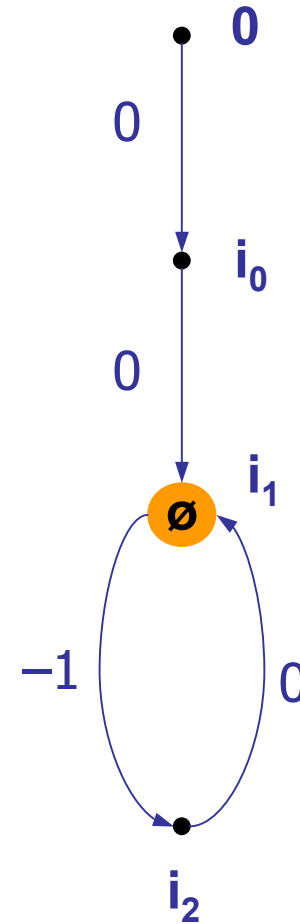
Extending the SSA form



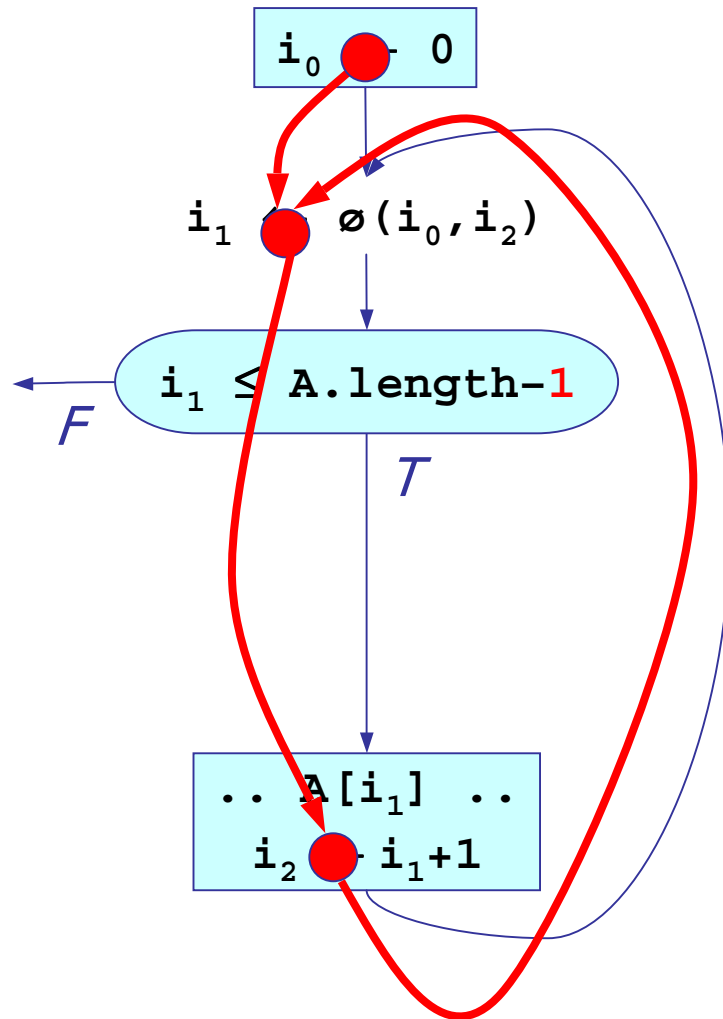
Extending the SSA form



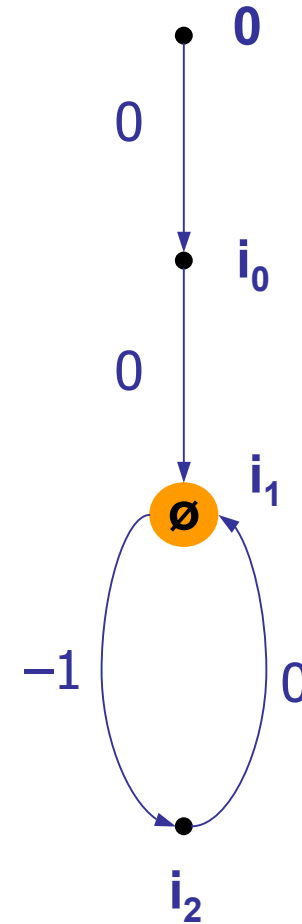
A.length •



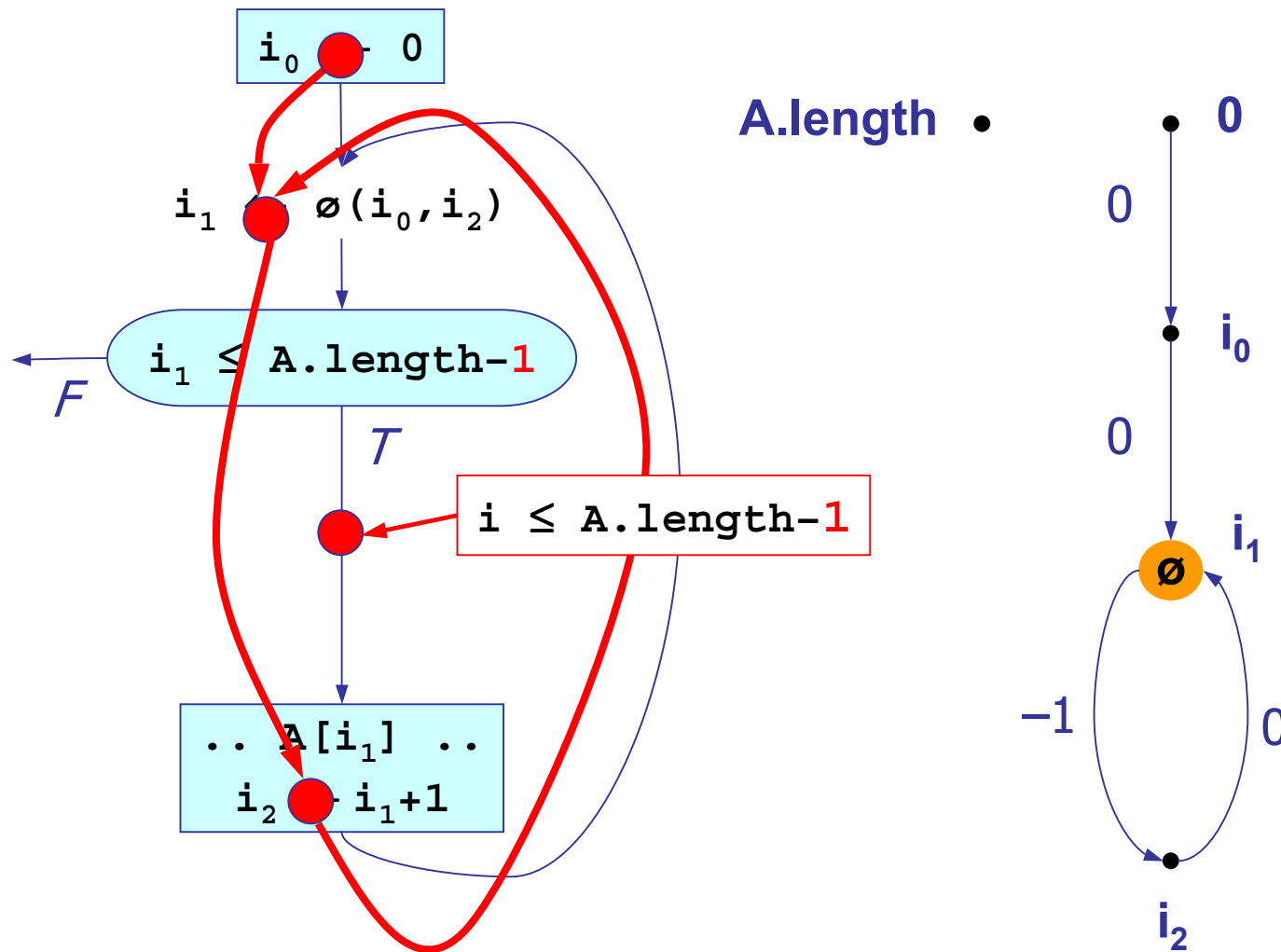
Extending the SSA form



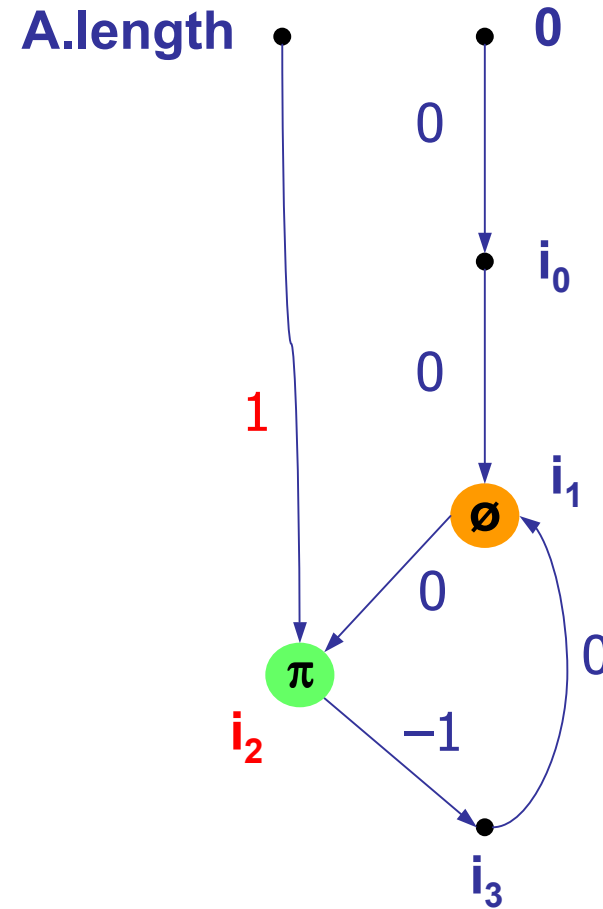
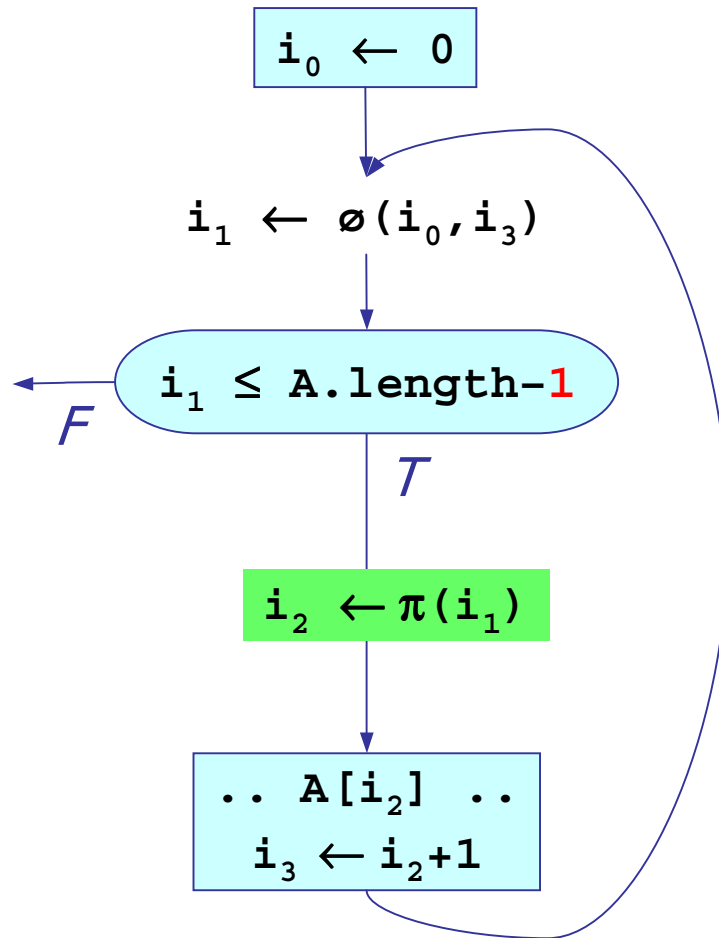
A.length •



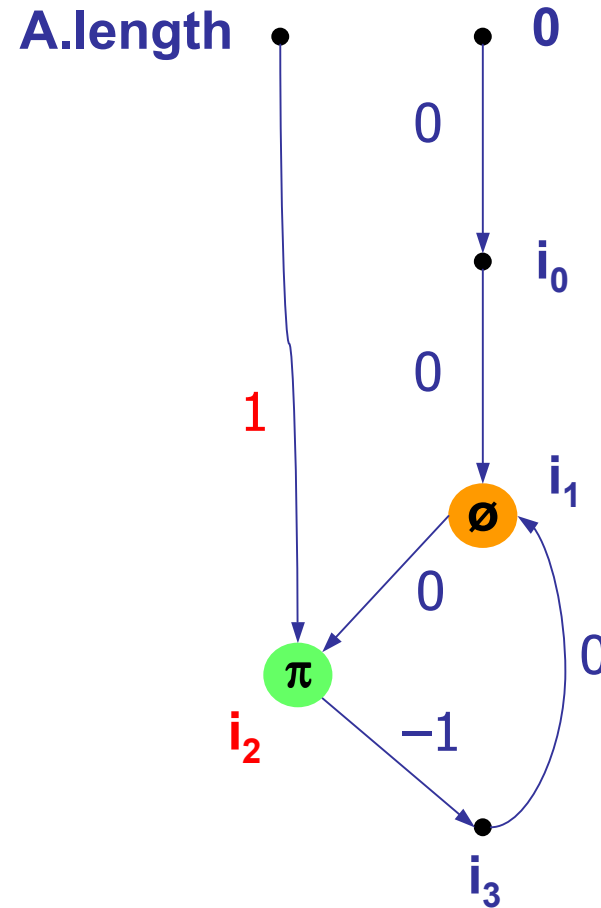
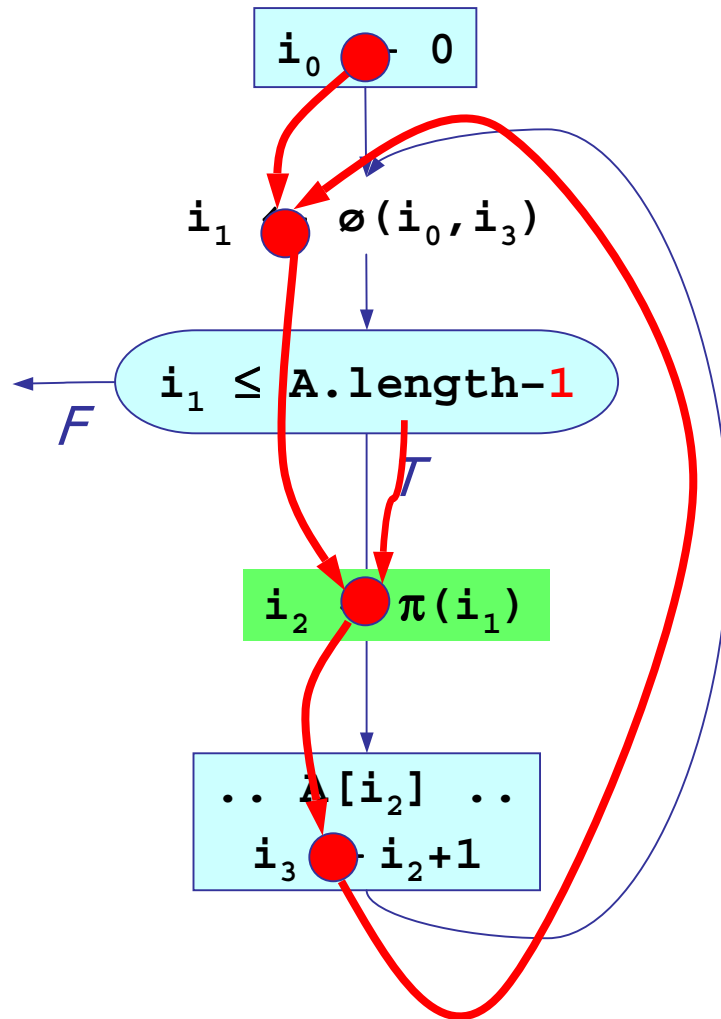
Extending the SSA form



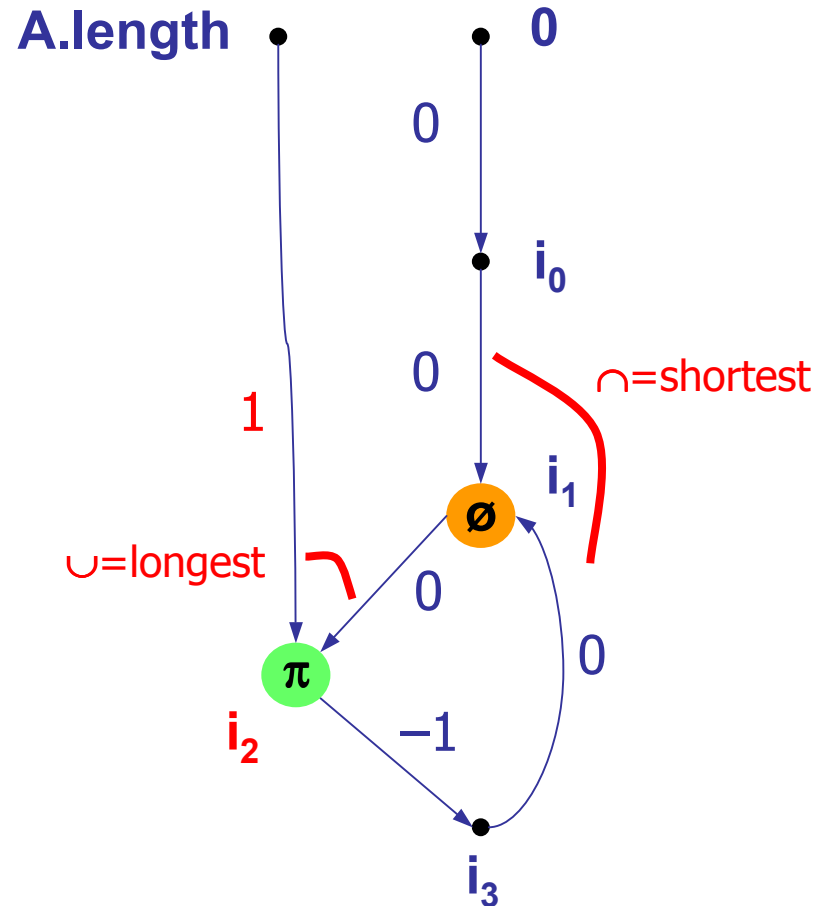
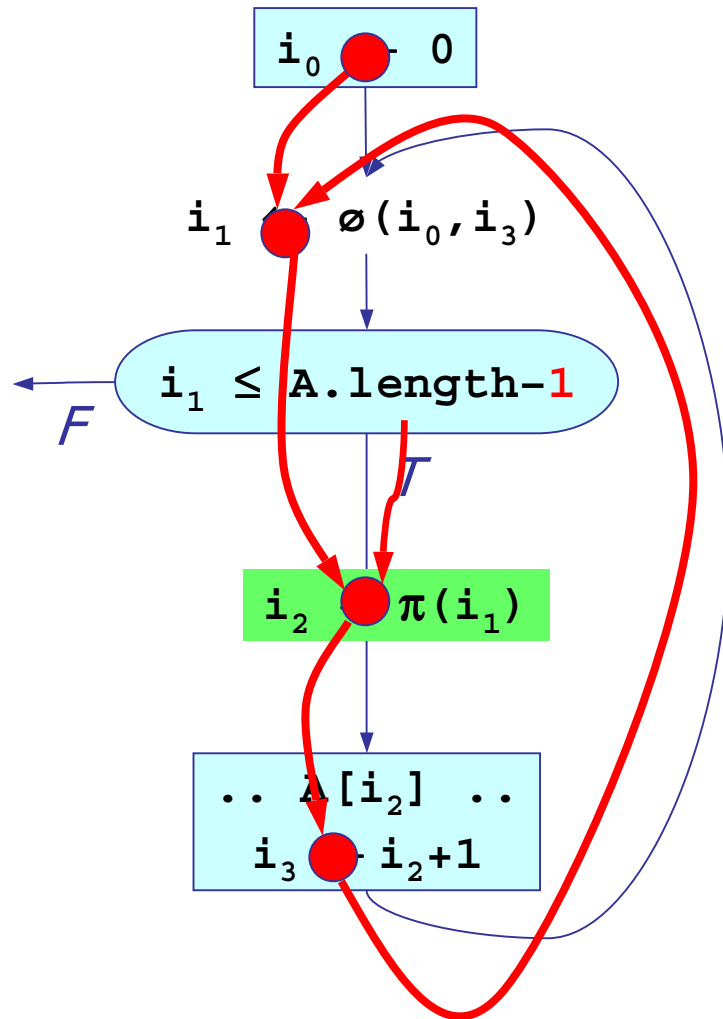
Extending the SSA form



Extending the SSA form



Extending the SSA form



hyper-graph: has two kinds of nodes

3. ABCD with PRE

19

PRE = partial redundancy elimination

```
f(int A[], int n)
{

    for (i=0; i < n; i++)
        ..A[i]..

}
```

3. ABCD with PRE

PRE = partial redundancy elimination

```
f(int A[], int n)
{
  if (n <= A.length)
  for (i=0; i < n; i++)
    ..A[i]..
}
```

false

unoptimized
loop

3. ABCD with PRE

PRE = partial redundancy elimination

```
f(int A[], int n)
{
  if (n <= A.length)
  for (i=0; i < n; i++)
    ..A[i]..
}
```

false

unoptimized
loop

ABCD with PRE = Full ABCD + profile feedback

3. ABCD with PRE

20

- 1. build extended SSA**
- 2. label edges with constraints**
- 3. analyze $A[i_k]$:**

Algorithm:

- find paths with “bad” length
- fix their length by inserting run-time checks

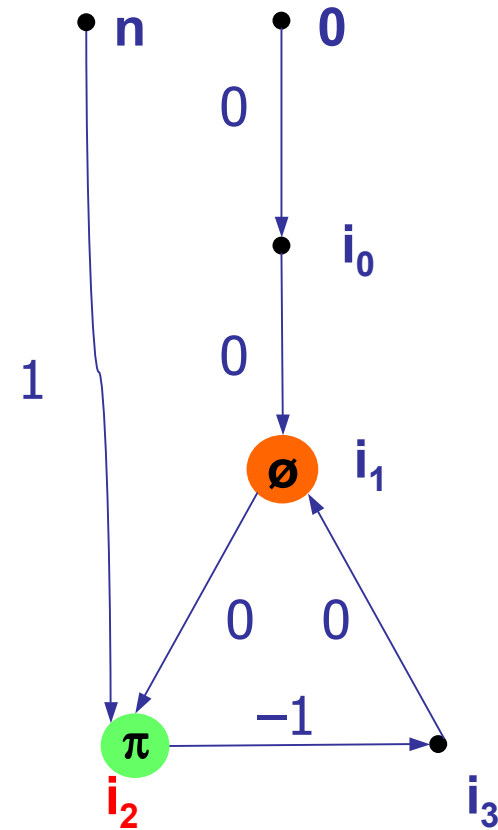
Issues:

- *What* check to insert?
- *Where* to insert the check?
- *When* is insertion profitable?

ABCD with PRE

A.length •

```
f(int A[], int n)
{
    for (i=0; i<n; i++)
        .. A[i] ..
}
```

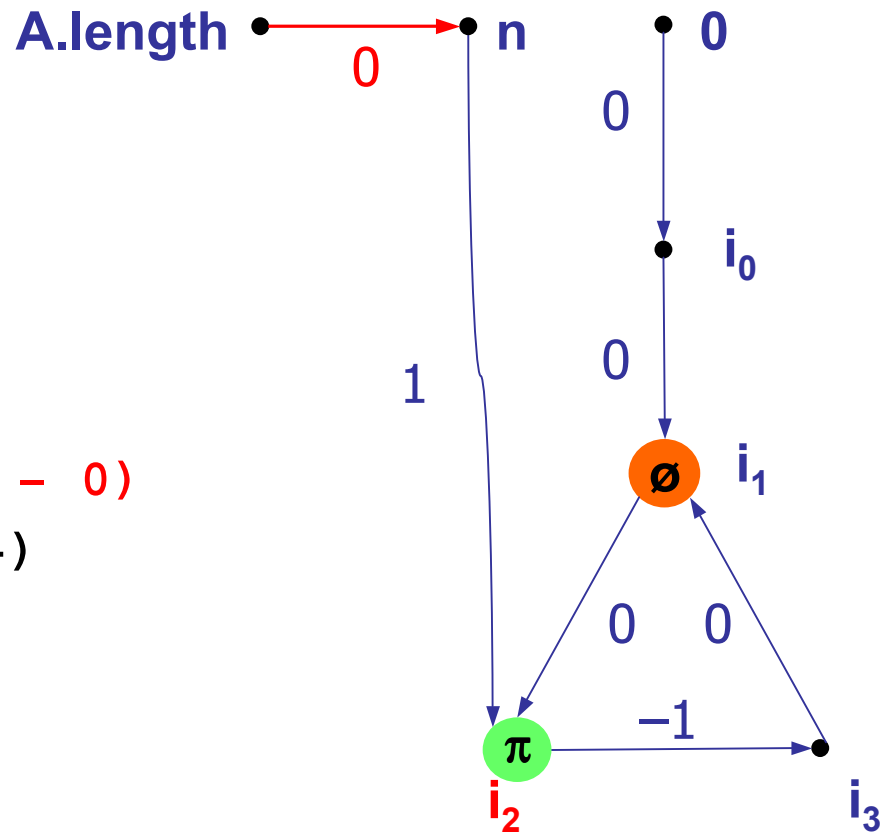


ABCD with PRE

```
f(int A[], int n)
{
  if (n <= A.length - 0)
  for (i=0; i<n; i++)
    .. A[i] ..
}
```

DONE!

constraint edges can be added
by inserting run-time checks



4. ABCDE

22

```
f(int A[], int n, i, j)
{


    for ( ; i < n; i++, j++)
        .. A[j] ..
}
```

When does ABCD fail?

4. ABCDE

22

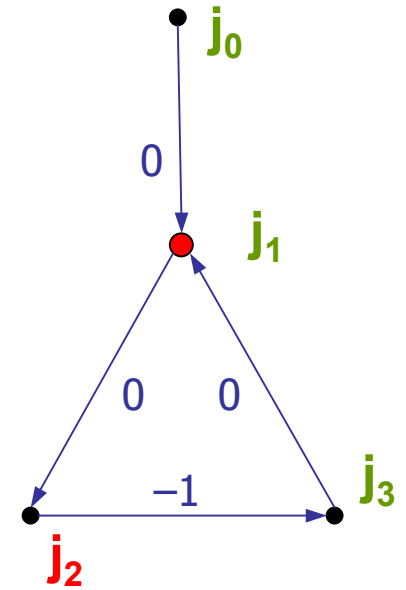
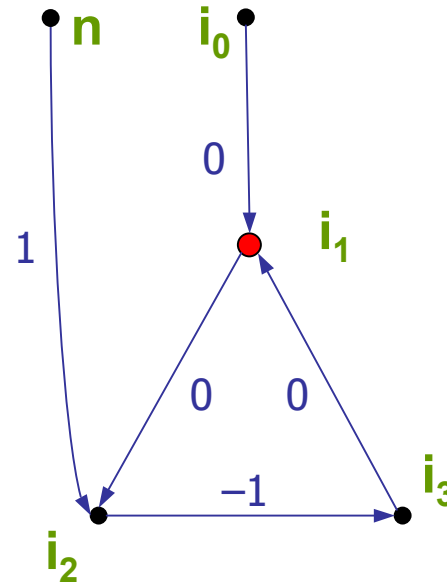
```
f(int A[], int n, i, j)
{
    if ( n <= A.length - (j - i) )
        for ( ; i < n; i++, j++)
            .. A[j] ..
}
```



unoptimized
loop

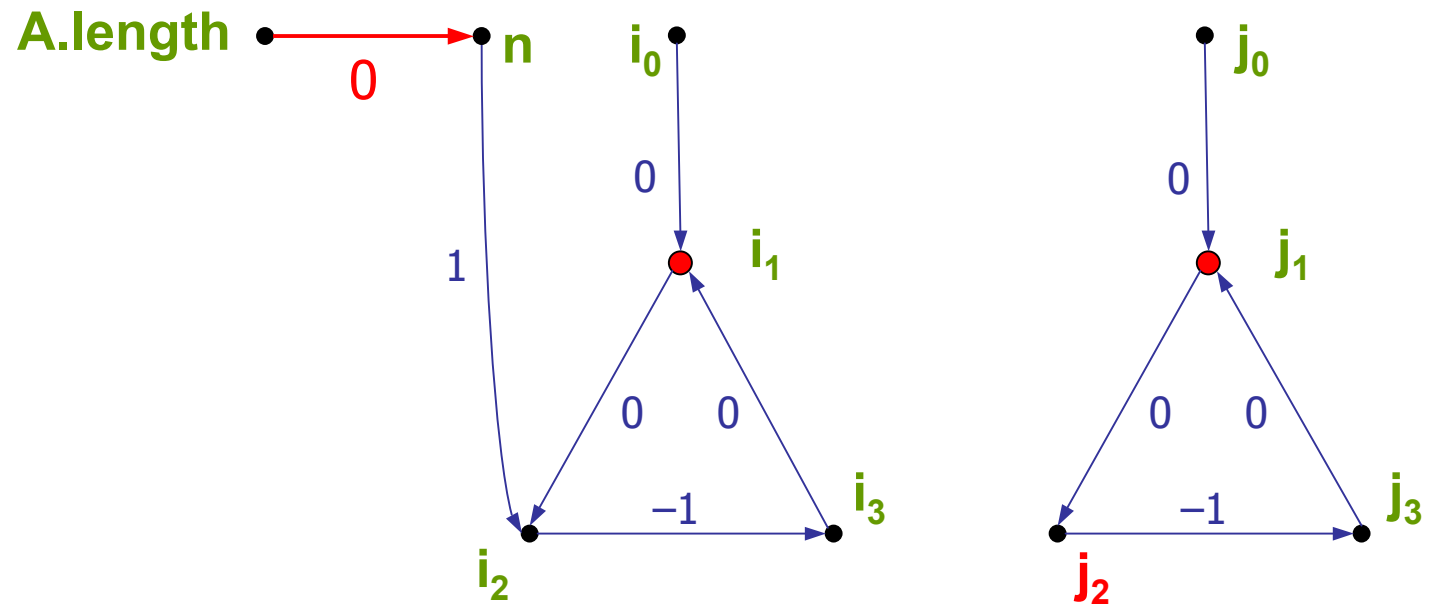
When does ABCD fail?

A.length •



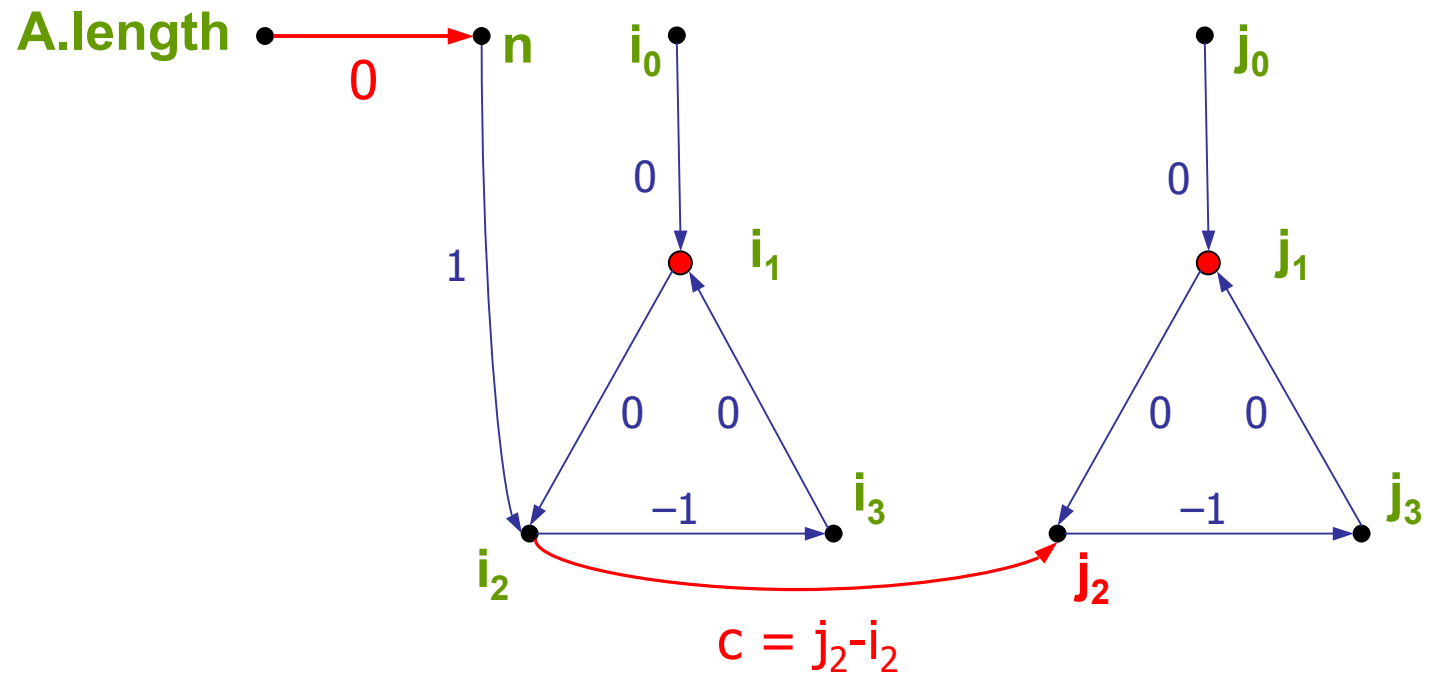
```
for (i=j=0 ; i<n; i++,j++) {
    .. A[j] ..
}
```

ABCDE



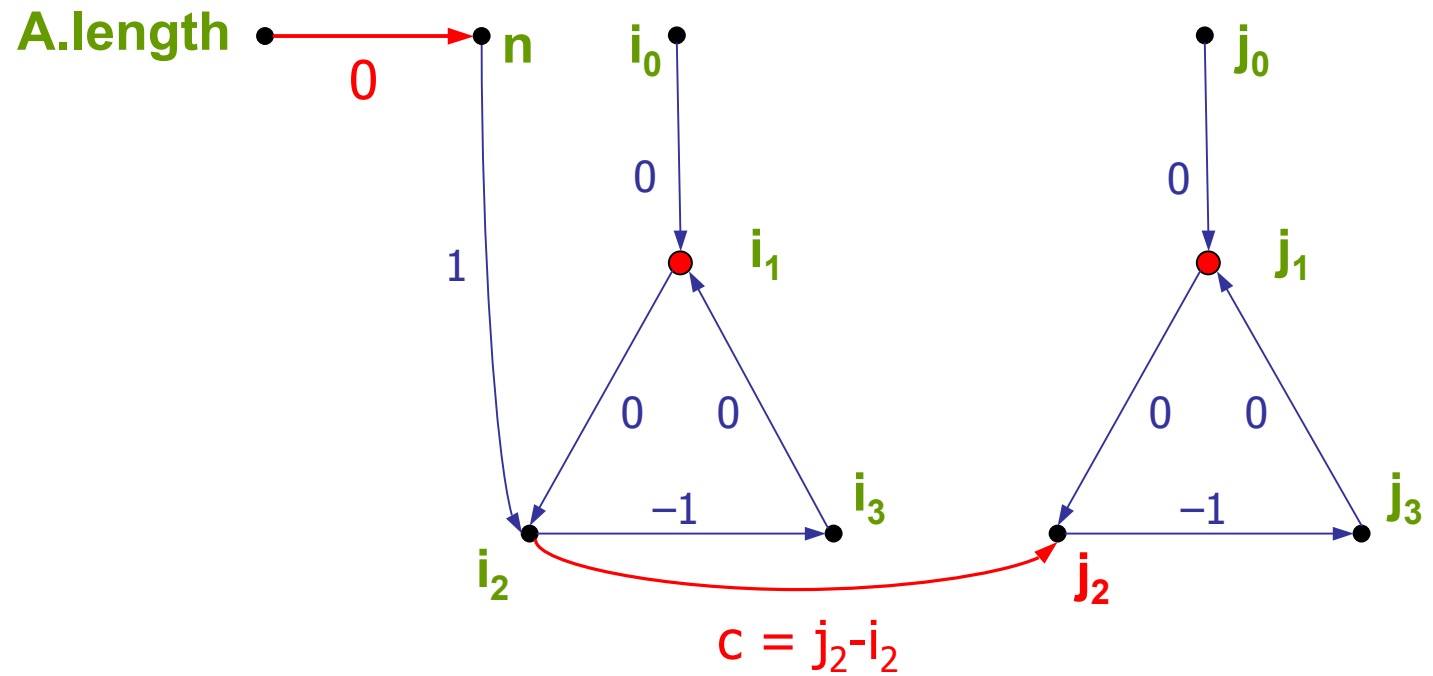
```
for (i=j=0 ; i<n; i++,j++) {  
    .. A[j] ..  
}
```

ABCDE



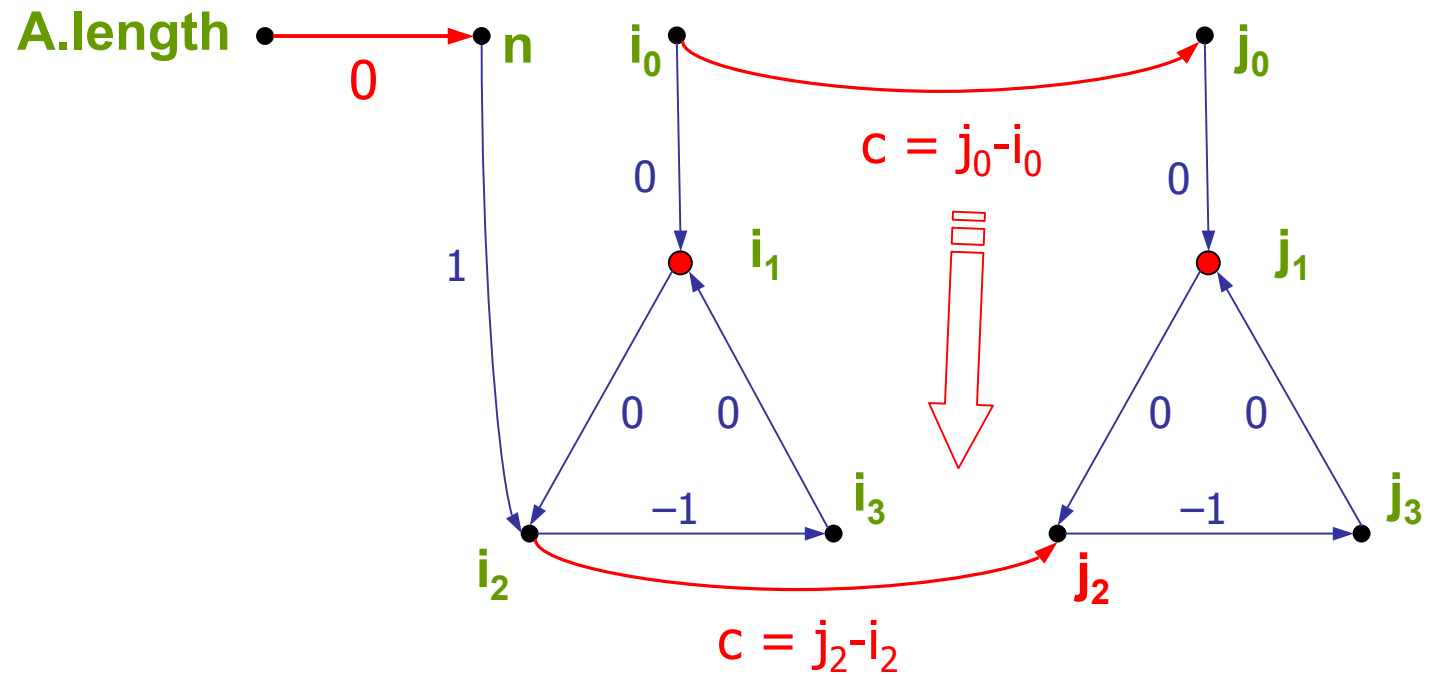
```
for (i=j=0 ; i<n; i++,j++) {  
    .. A[j] ..  
}
```

ABCDE



```
for (i=j=0 ; i<n; i++,j++) {  
    if (n <= A.length - (j2-i2))  
        .. A[j] ..  
}
```

ABCDE



```
if (n <= A.length - (j0 - i0))  
for (i=j=0 ; i<n; i++,j++) {  
  
    .. A[j] ..  
}
```

ABCD is simple

24

- yet powerful ...

A complex example

from paper

25

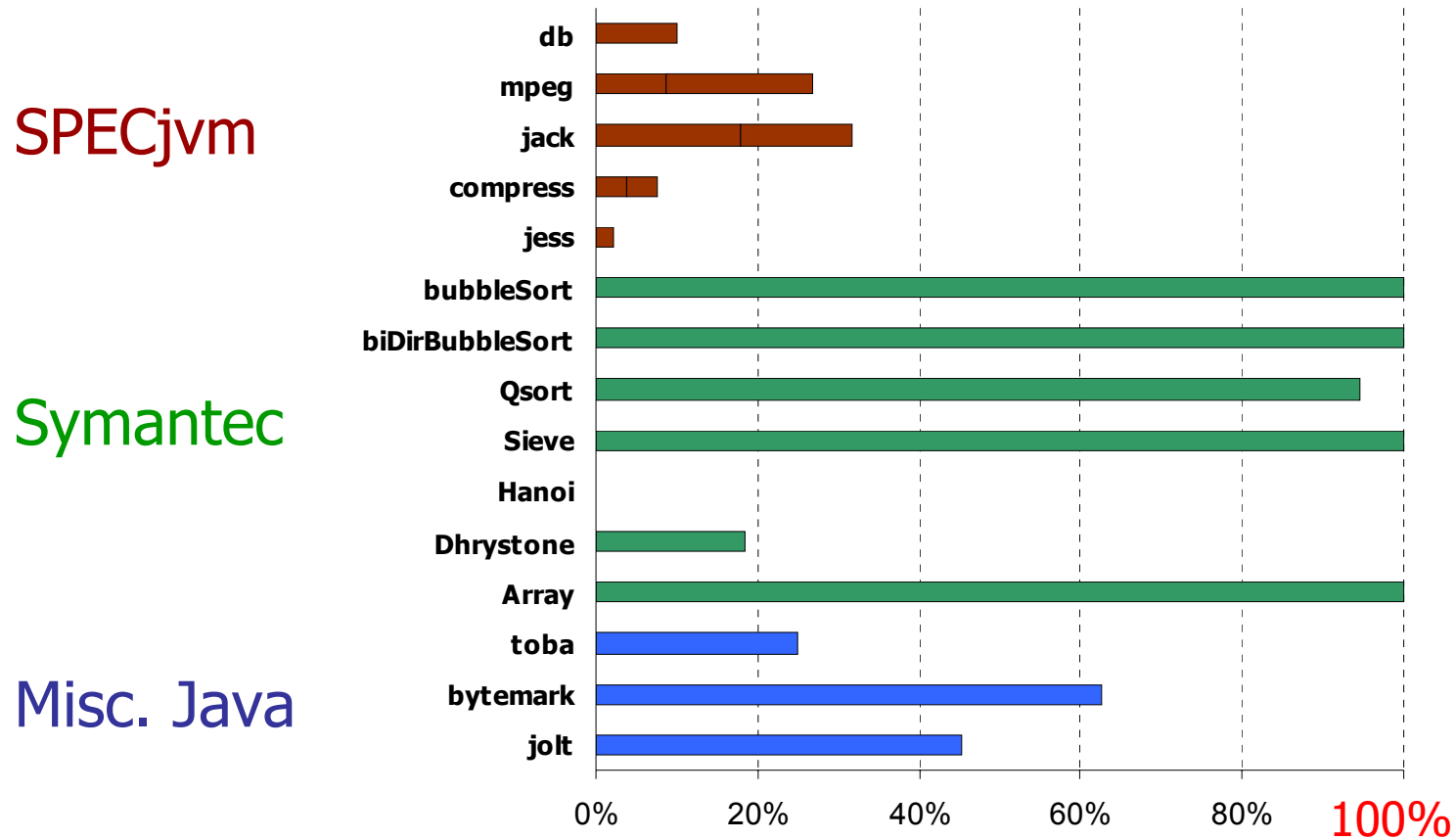
```
limit = a.length
st = -1
while (st < limit)
    st++
    limit - -
    for (j = st; j < limit; j++) {
        A[j] += A[j+1]
    }
}
```

A complex example from paper

26

```
limit = a.length
st = -1
while (st < limit)
    st++
    limit --
    for (j = st; j < limit; j++) {
        A[j] += A[j+1]
    }
}
```

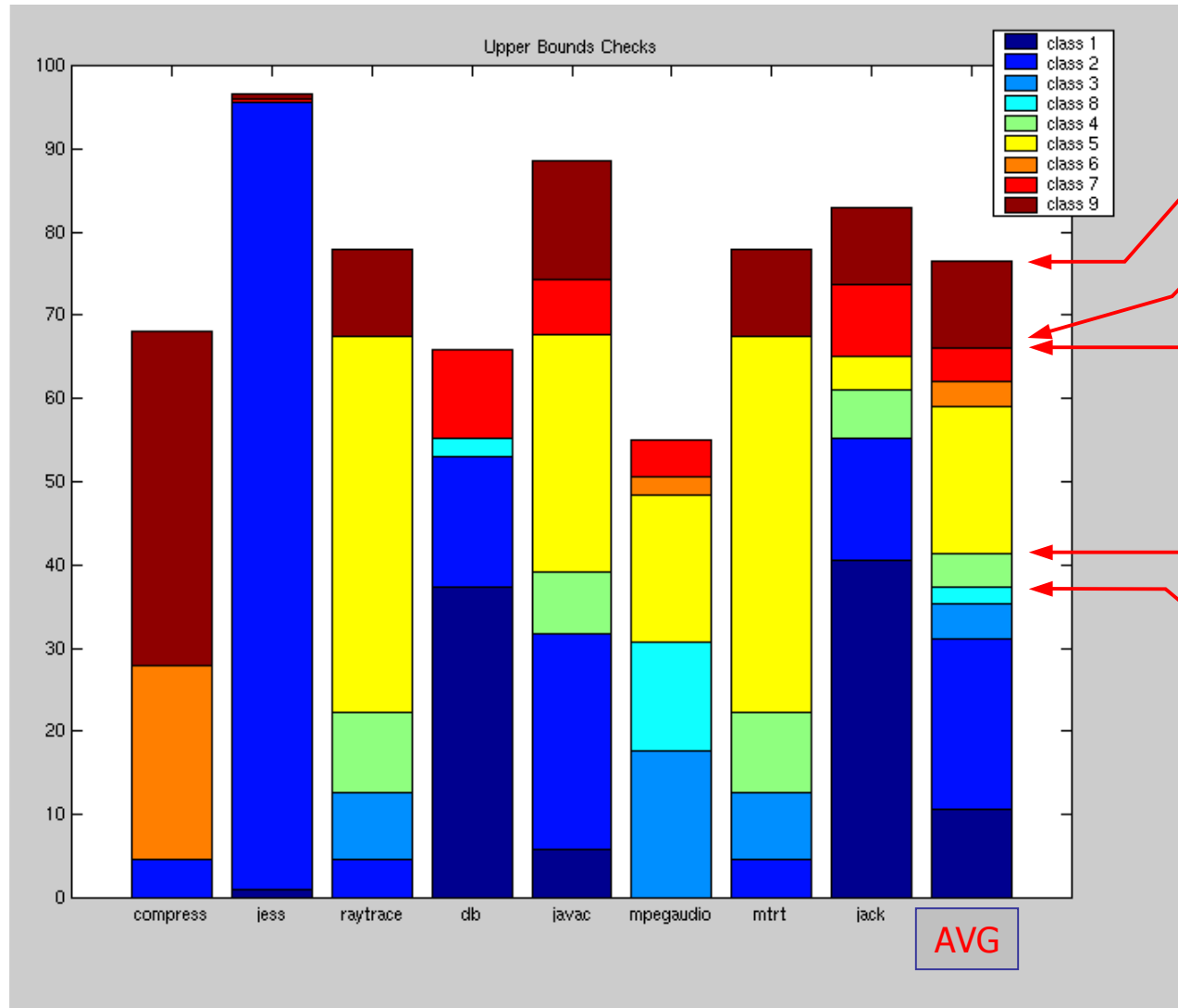
How powerful is ABCD?



checks removed [% of all dynamic checks]

Classification of hot checks

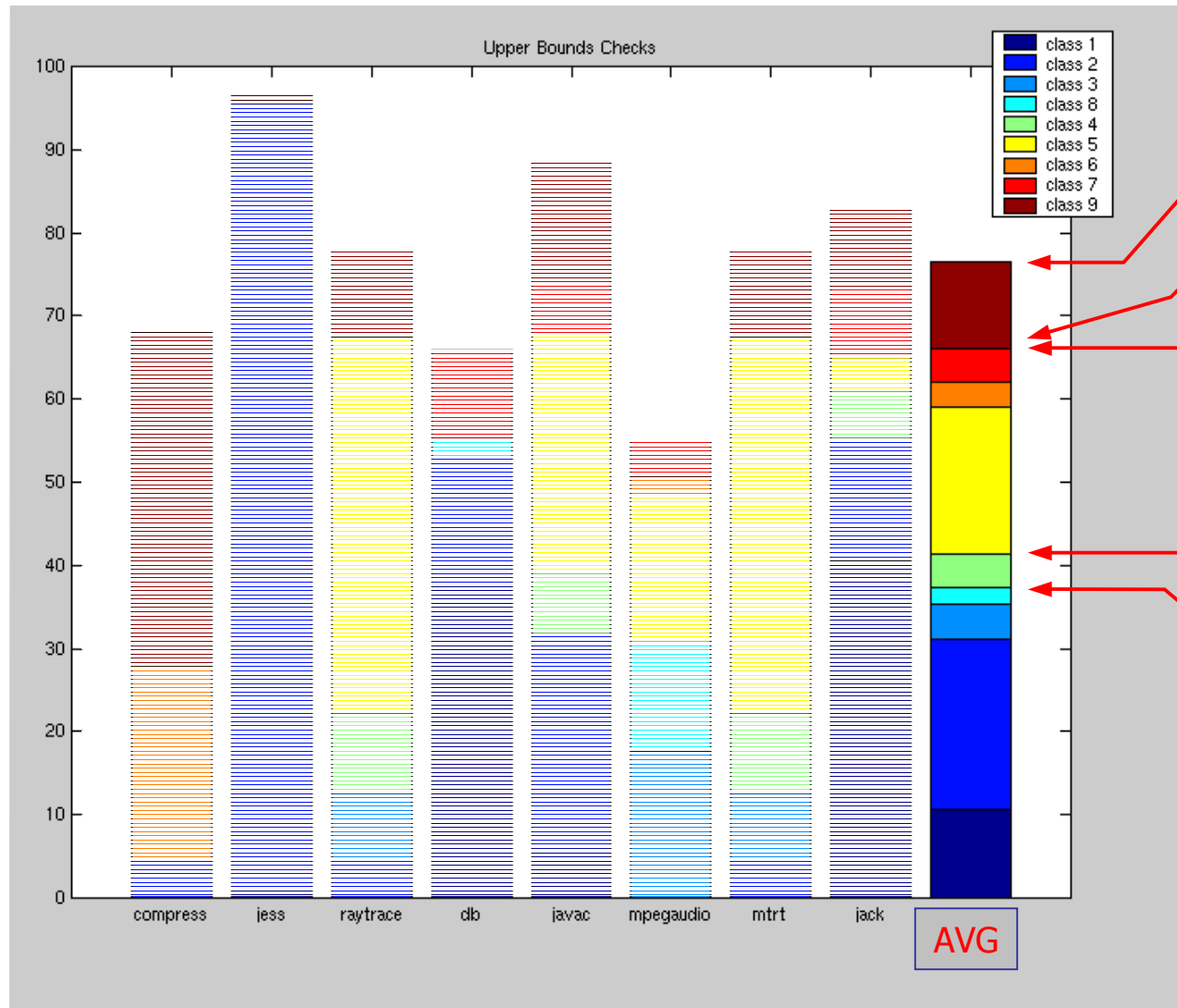
checks removed [% of all dynamic checks]



examined
"removable"
ABCDE
ABCD
JIT-like

Classification of hot checks

checks removed [% of all dynamic checks]



examined
"removable"
ABCDE
ABCD
JIT-like

Summary

29

- **Current speedup:** modest, up to about 5%
 - **direct cost:** in Jalapeno, checks already very efficient
 - **indirect cost:** few global optimizations implemented (11/99)
 - **Analysis time**
 - 4 ms / check = visit 10 SSA nodes / check
 - recall 20 checks yields 80% dynamic coverage
 - ⇒ 80 ms to analyze a large benchmark !
 - **Precision:**
 - can be improved with few extensions (ABCD → ABCDE)
 - remaining checks appear beyond compiler analysis
- 👉 **Use ABCD for your bounds-check optimization**