Program Slicing

Xiangyu Zhang

What is a slice?

```
Void main ( ) {
  int I=0;
  int sum=0;
  while (I<N) {
      sum=add(sum,I);
      I=add(I,1);
   }
  printf ("sum=%d\n",sum);
  printf("I=%d\n",I);
```

S: …. = f (v)

- **Slice** of v at S is the set of statements involved in computing v's value at S.

[Mark Weiser, 1982]

- Data dependence
- Control dependence

Why Slicing

- Debugging
- Testing
- Differencing
- Program understanding
- Software maintenance
- Complexity measurement / Functional Cohesion
- Program integration
- Reverse engineering
- Software Quality Assurance

Old!

What Now

- Security
  - Malware detection;
  - Software piracy
  - …
- Software Transactional Memory
- Architecture
  - Value speculation
- Program optimization
  - PRE
- Data Lineage
- More to come

A program implement multiple semantic functions. All are not relevant!

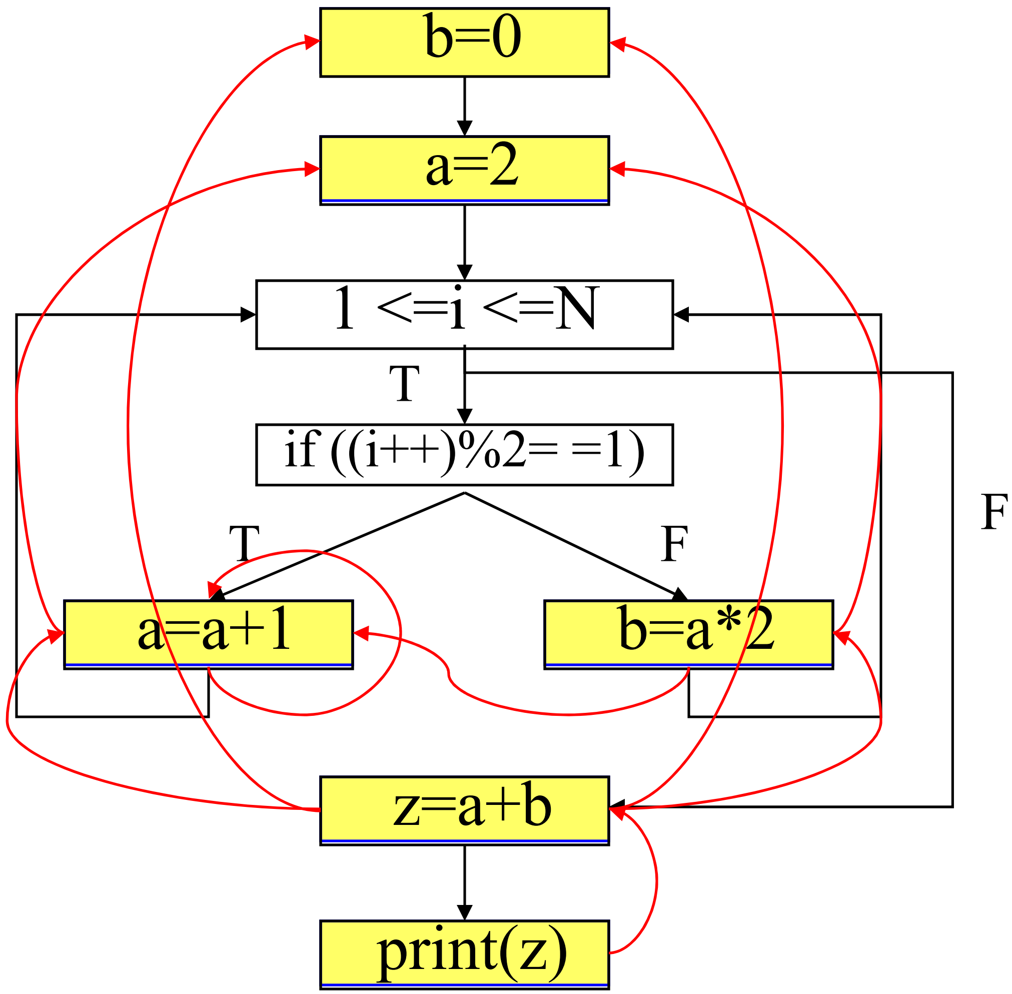
Outline

- Slicing ABC
- Dynamic slicing
  - Efficiency
  - Effectiveness
  - Challenges

Slicing Classification

- Static vs. Dynamic
- Backward vs. Forward
- Executable vs. Non-Executable
- More

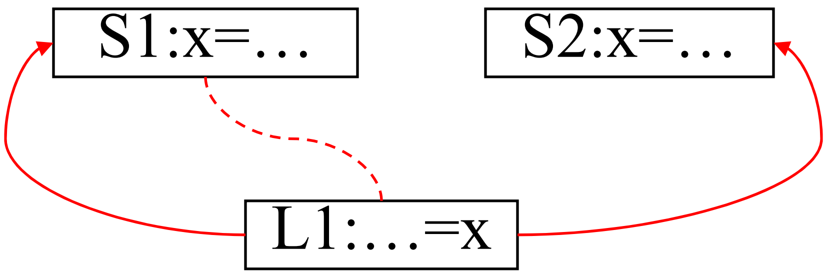
How to do slicing?



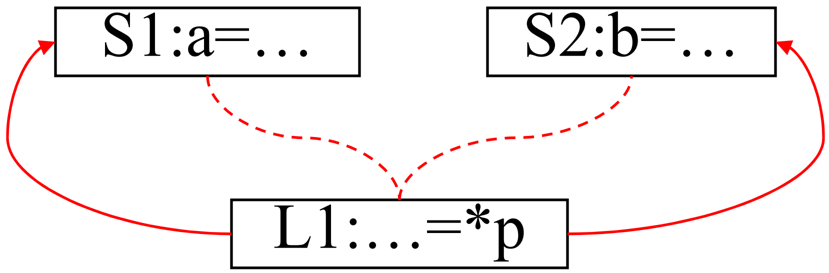
- Static analysis
  - Input insensitive
  - May analysis
- Dependence Graph
- Characteristics
  - Very fast
  - Very imprecise

Why is a static slice imprecise?

- All possible program paths



- Use of Pointers – static alias analysis is very imprecise



- Use of function pointers – hard to know which function is called, conservative expectation results in imprecision

Dynamic Slicing

- Korel and Laski, 1988
- Dynamic slicing makes use of all information about a particular execution of a program and computes the slice based on an execution history (trace)
  - Trace consists control flow trace and memory reference trace
- A dynamic slice query is a triple
  - <Var, Input , Execution Point>
- Smaller, more precise, more helpful to the user

Dynamic Slicing Example -background

```
1: b=0
2: a=2
3: for i= 1 to N do
4:  if ((i++)%2==1) then
5:       a = a+1
    else
6:       b = a*2
    endif
   done
7: z = a+b
8: print(z)
```

For input N=2,

1_1:	b=0	[b=0]
2_1:	a=2	
3_1:	for i = 1 to N do	[i=1]
4_1:	if ((i++) %2 == 1) then	[i=1]
5_1:	a=a+1	[a=3]
3_2:	for i=1 to N do	[i=2]
4_2:	if (i%2 == 1) then	[i=2]
6_1:	b=a*2	[b=6]
7_1:	z=a+b	[z=9]
8_1:	print(z)	[z=9]

Issues about Dynamic Slicing

- Precision – perfect
- Running history – very big (GB)
- Algorithm to compute dynamic slice - slow and very high space requirement.

Backward vs. Forward

```
1 main( )
2 {
3   int i, sum;
4   sum = 0;
5   i = 1;
6   while(i <= 10)
7       {
8        sum = sum + 1;
9        ++ i;
10        }
11       Cout<< sum;
12       Cout<< i;
13       }
```

An Example Program & its forward slice w.r.t. <3, sum>

Executable vs. Non-Executable

```
program Example;
begin
  a := 17;
  b := 18;
  P(a, b, c, d);
  write(d)
end

procedure P(v, w, x, y);
  x := v;
  y := w
end
```

(a)

```
program Example;
begin
  a := 17;
  b := 18;
  P(a, b, c, d);

end

procedure P(v, w, x, y);
        ;
  y := w
end
```

(b)

```
program Example;
begin
         ;
  b := 18;
  P(a, b, c, d);
  write(d)
end

procedure P(v, w, x, y);
        ;
  y := w
end
```

(c)

Comments

- Want to know more?
  - Frank Tip's survey paper (1995)
- Static slicing is very useful for static analysis
  - Code transformation, program understanding, etc.
  - Points-to analysis is the key challenge
  - Not as useful in reliability as dynamic slicing
- We will focus on dynamic slicing
  - Precise
    - good for reliability.
  - Solution space is much larger.
  - There exist hybrid techniques.

Outline

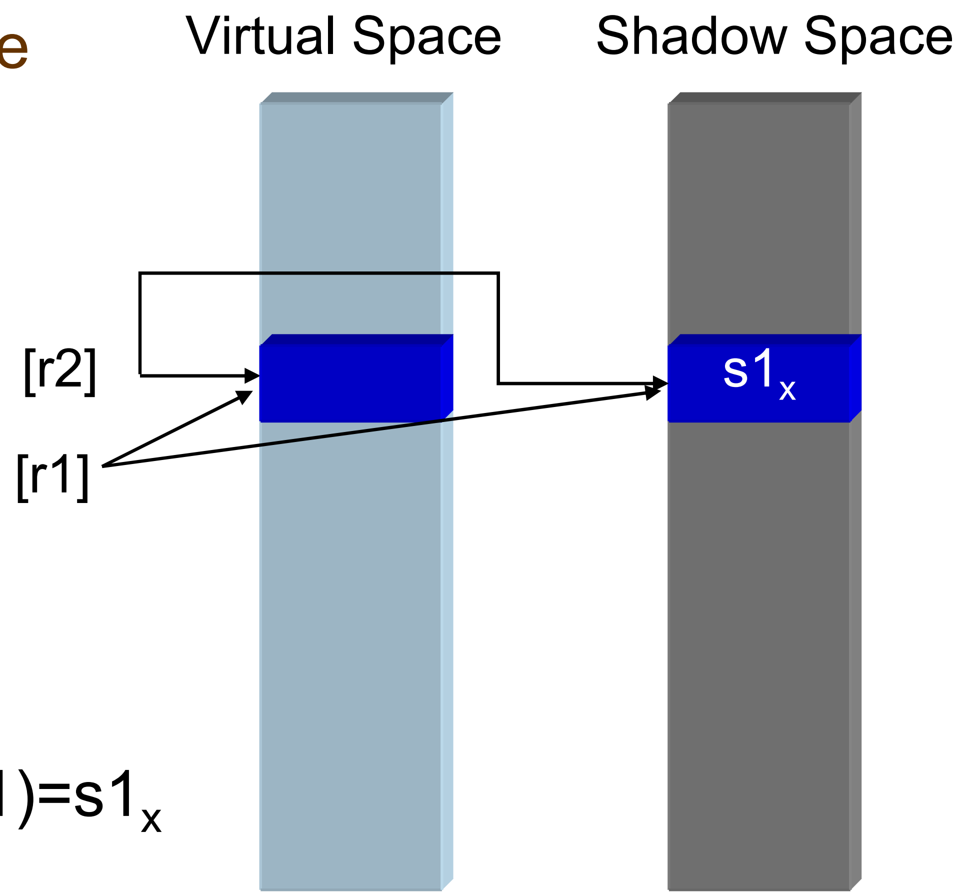
- Slicing ABC
- Dynamic slicing
  - Efficiency
  - Effectiveness
  - Challenges

Efficiency

- How are dynamic slices computed?
  - Execution traces
    - control flow trace -- dynamic control dependences
    - memory reference trace -- dynamic data dependences
  - Construct a dynamic dependence graph
  - Traverse dynamic dependence graph to compute slices

How to Detect Dynamic Dependence

- Dynamic Data Dependence
  - Shadow space (SS)
    - Addr → Abstract State



$s1_x$: ST r1, [r2] ⇒ $SS(r2)=s1_x$

$s2_y$: LD [r1], r2 ⇒ $s2_y \rightarrow SS(r1)=s1_x$

Dynamic control dependence is more tricky!

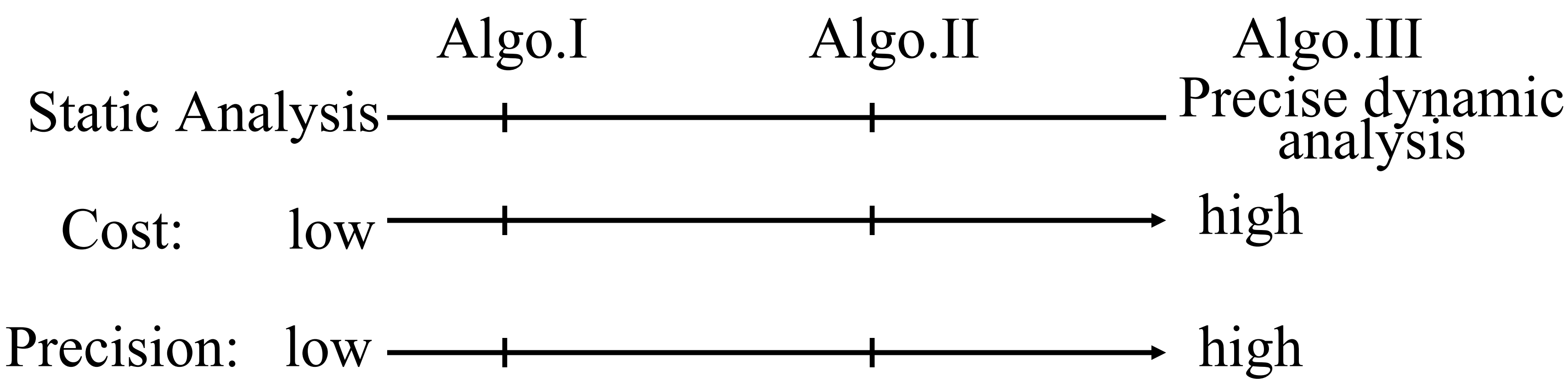
Dynamic Dependence Graph Sizes

Program	Statements Executed (Millions)	Dynamic Dependence Graph Size(MB)
300.twolf	140	1,568
256.bzip2	67	1,296
255.vortex	108	1,442
197.parser	123	1,816
181.mcf	118	1,535
134.perl	220	1,954
130.li	124	1,745
126.gcc	131	1,534
099.go	138	1,707

- On average, given an execution of 130M instructions, the constructed dependence graph requires 1.5GB space.

Conventional Approaches

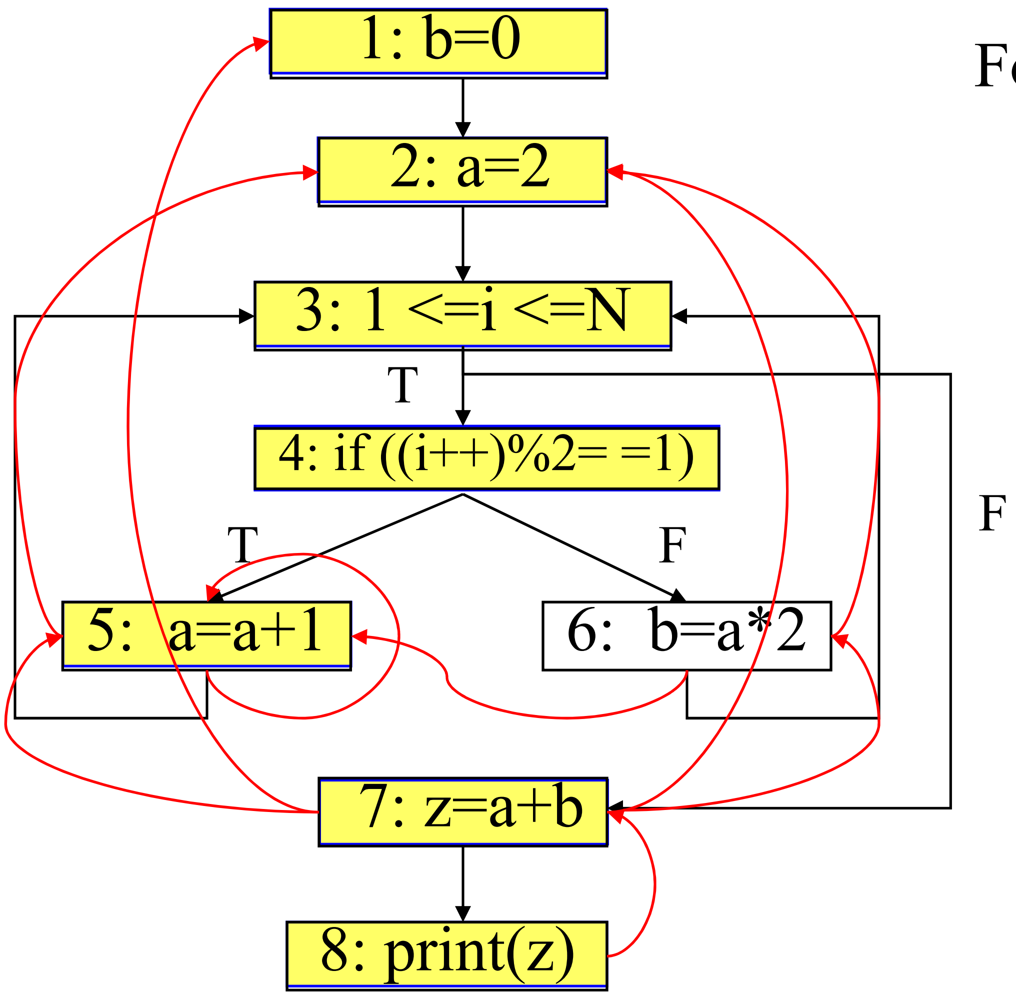
- [Agrawal &Horgan, 1990] presented three algorithms to trade-off the cost with precision.



Algorithm One

- This algorithm uses a static dependence graph in which all executed nodes are marked dynamically so that during slicing when the graph is traversed, nodes that are not marked are avoided as they cannot be a part of the dynamic slice.
- Limited dynamic information - fast, imprecise (but more precise than static slicing)

Algorithm I Example



For input N=1, the trace is:

1_1

2_1

3_1

4_1

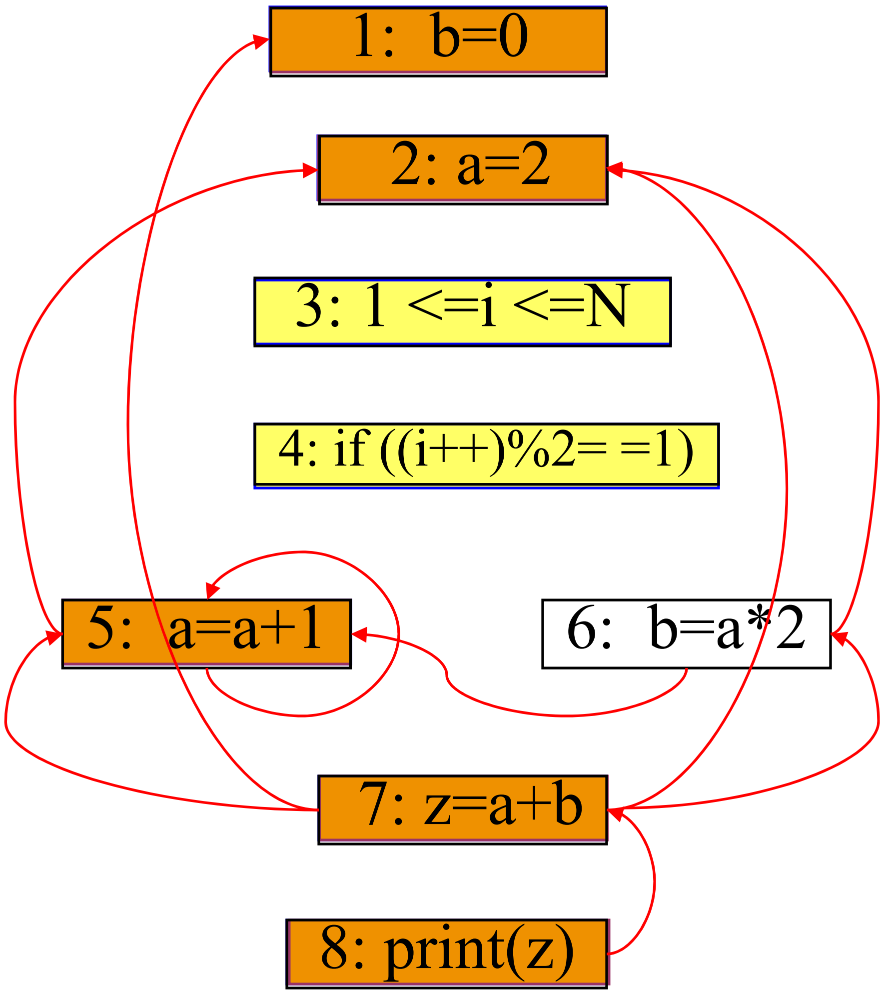
5_1

3_2

7_1

8_1

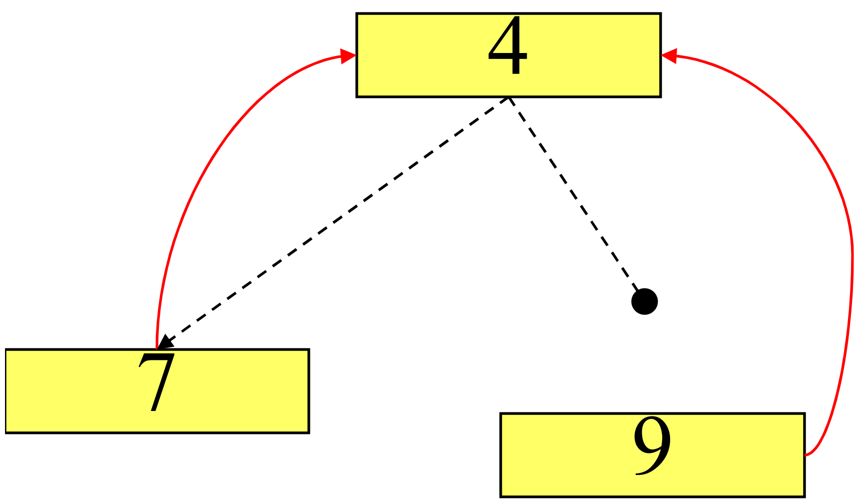
Algorithm I Example



DS={1,2,5,7,8}

Precise!

Imprecision introduced by Algorithm I



Input N=2:

```
1   for  (a=1; a<N; a++) {
2      …
3      if (a % 2== 1) {
4            b=1;
5      }
6      if (a % 3 ==1) {
7            b= 2* b;
8      } else {
9            c=2*b+1;
       }
    }
```

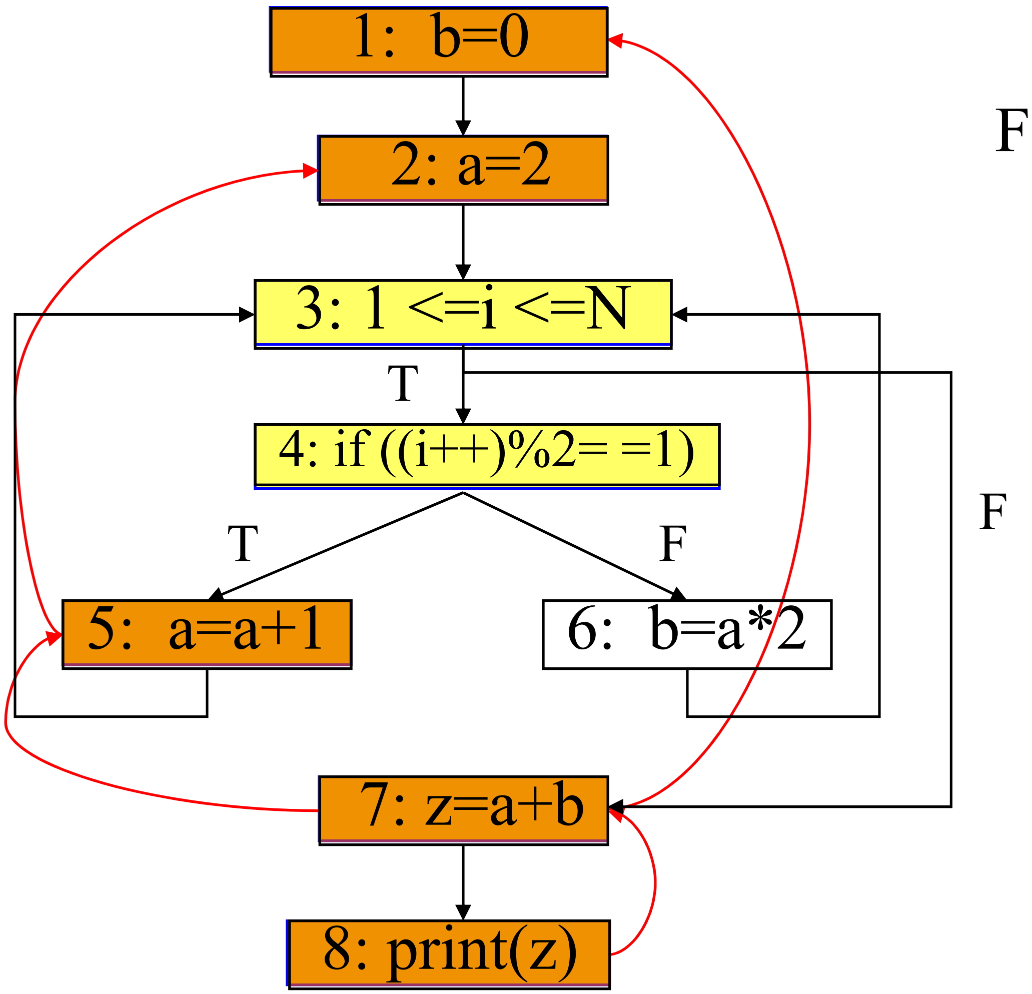
Killed definition counted as reaching!

Aliasing!

Algorithm II

- A dependence edge is introduced from a load to a store if during execution, at least once, the value stored by the store is indeed read by the load (mark dependence edge)
- No static analysis is needed.

Algorithm II Example



For input N=1, the trace is:

1_1

2_1

3_1

4_1

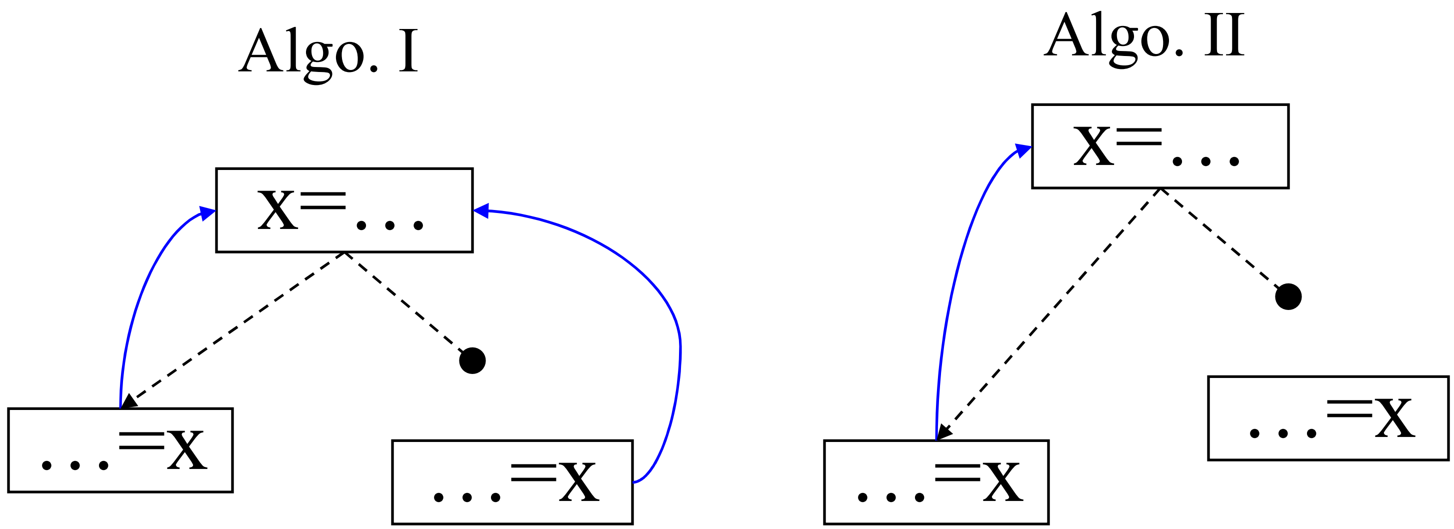
5_1

7_1

8_1

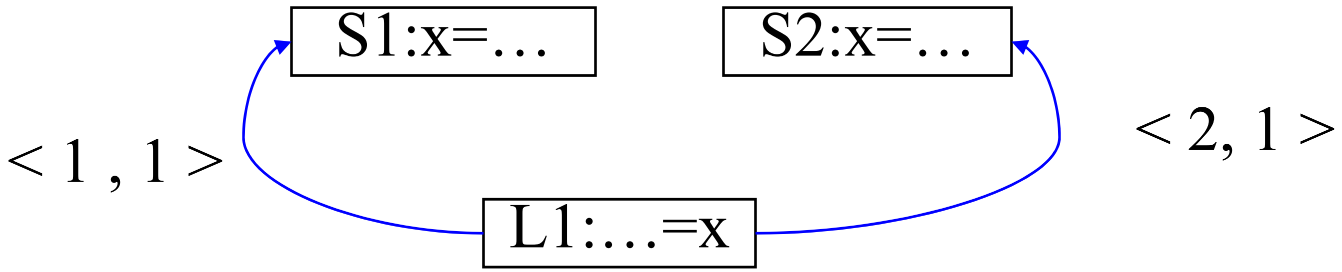
Algorithm II – Compare to Algorithm I

- More precise



Imprecision introduced by Algorithm II

- A statically distinct load/store may be executed several times during program execution. Different instances of a load may be dependent on different store instructions or different instances of a store instructions.



- Algo. 2 uses unlabeled edges. Therefore, upon inclusion of the load in the slice it will always include both the stores.

Algorithm III

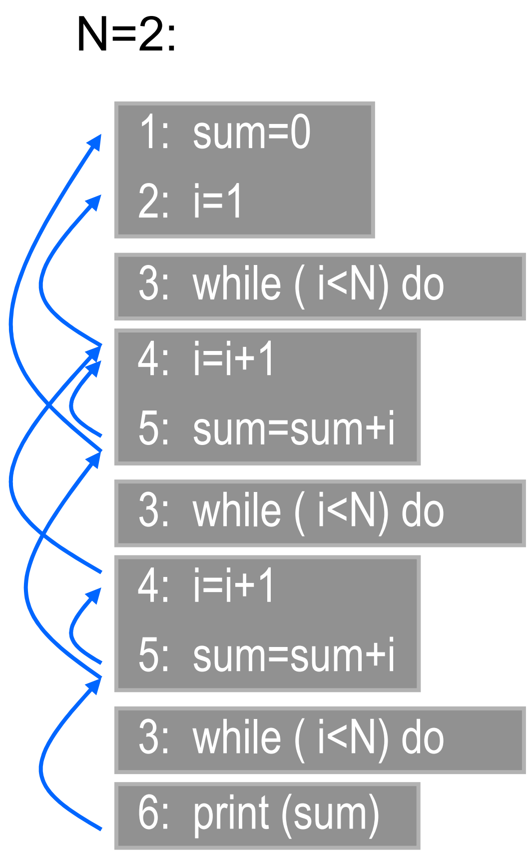
- First preprocess the execution trace and introduces labeled dependence edges in the dependence graph. During slicing the instance labels are used to traverse only relevant edges.

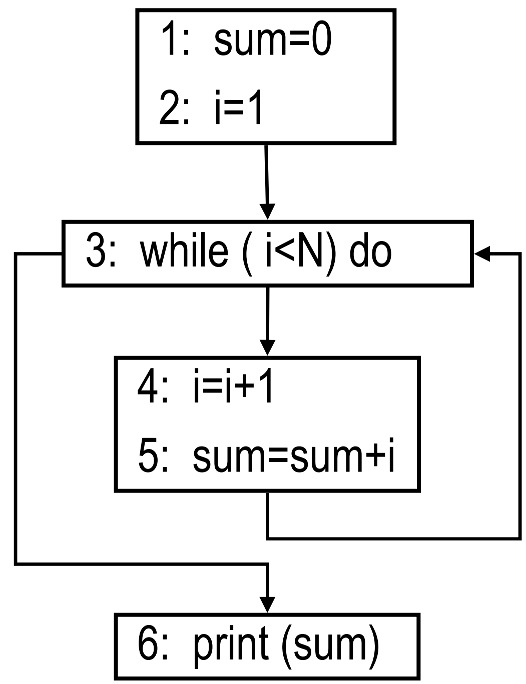
Dynamic Dependence Graph Sizes (revisit)

Program	Statements Executed (Millions)	Dynamic Dependence Graph Size(MB)
300.twolf	140	1,568
256.bzip2	67	1,296
255.vortex	108	1,442
197.parser	123	1,816
181.mcf	118	1,535
134.perl	220	1,954
130.li	124	1,745
126.gcc	131	1,534
099.go	138	1,707

- On average, given an execution of 130M instructions, the constructed dependence graph requires 1.5GB space.

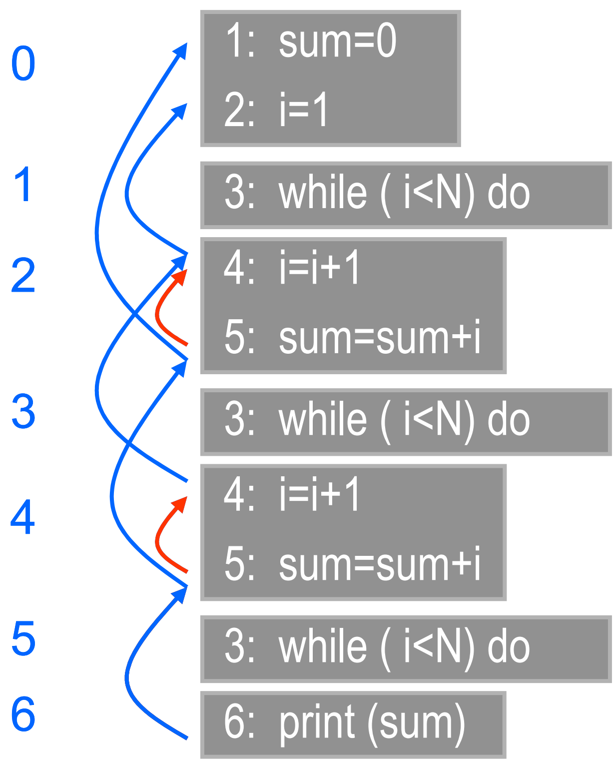
Dynamic Dep. Graph Representation

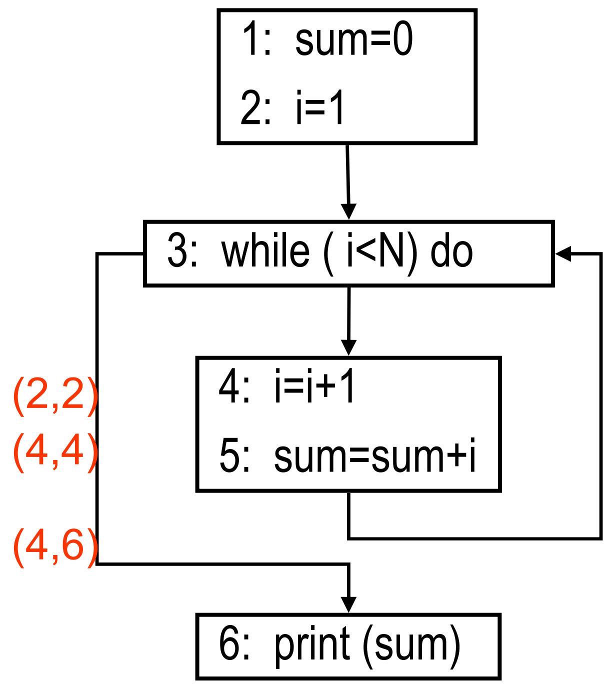




Dynamic Dep. Graph Representation

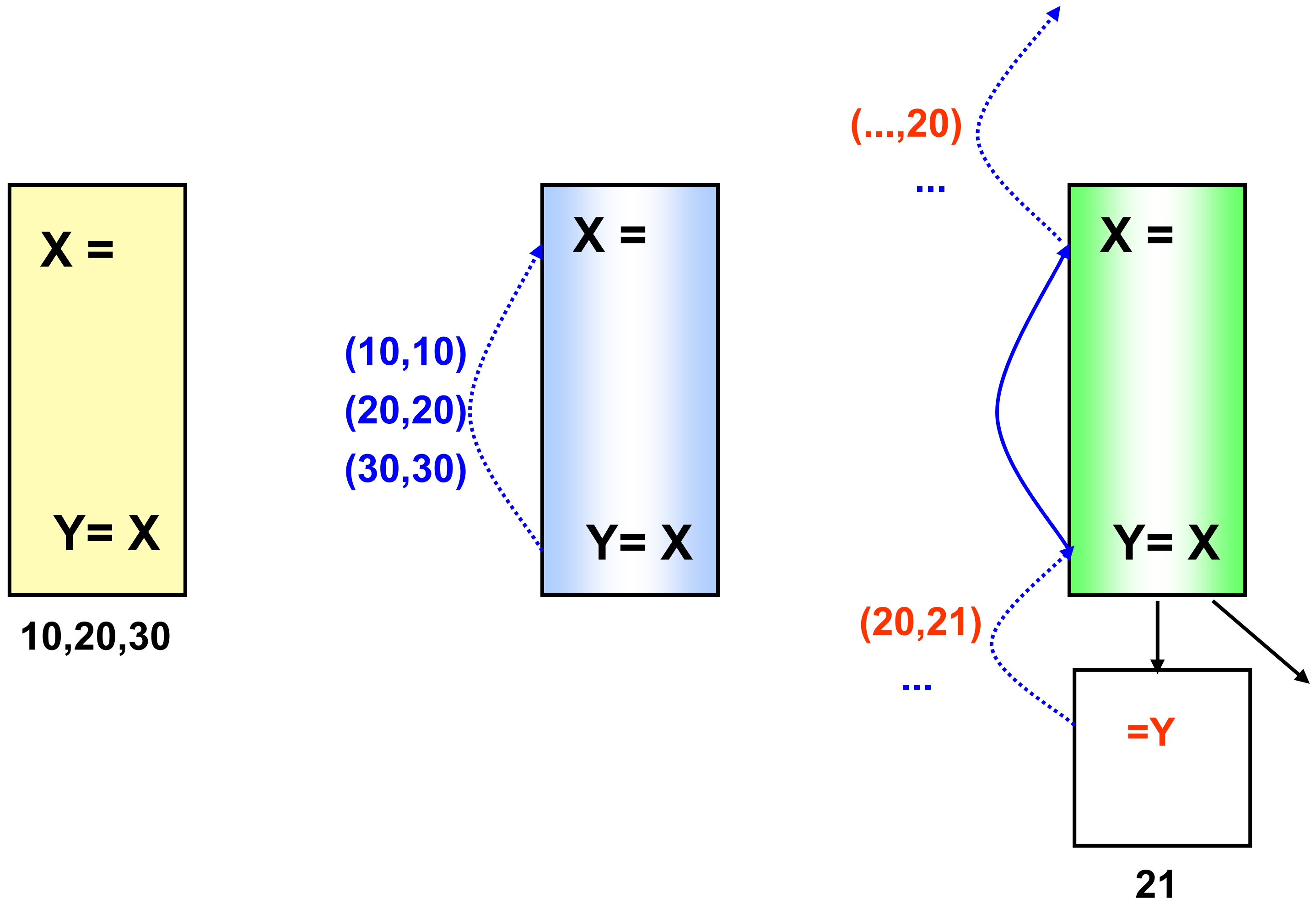
Timestamps N=2:



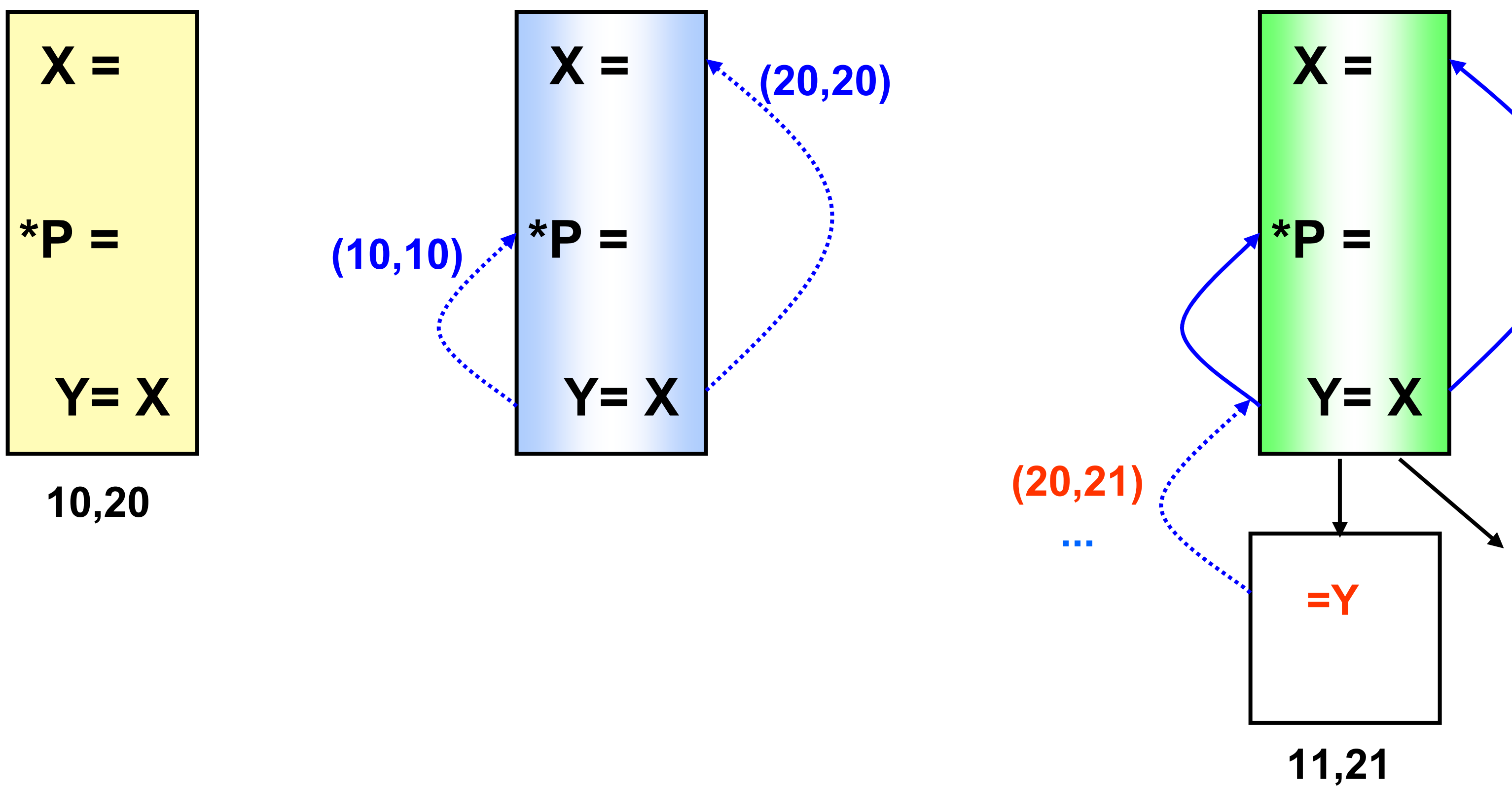


- A dynamic dep. edge is represented as by an edge annotated with a pair of timestamps <definition timestamp, use timestamp>

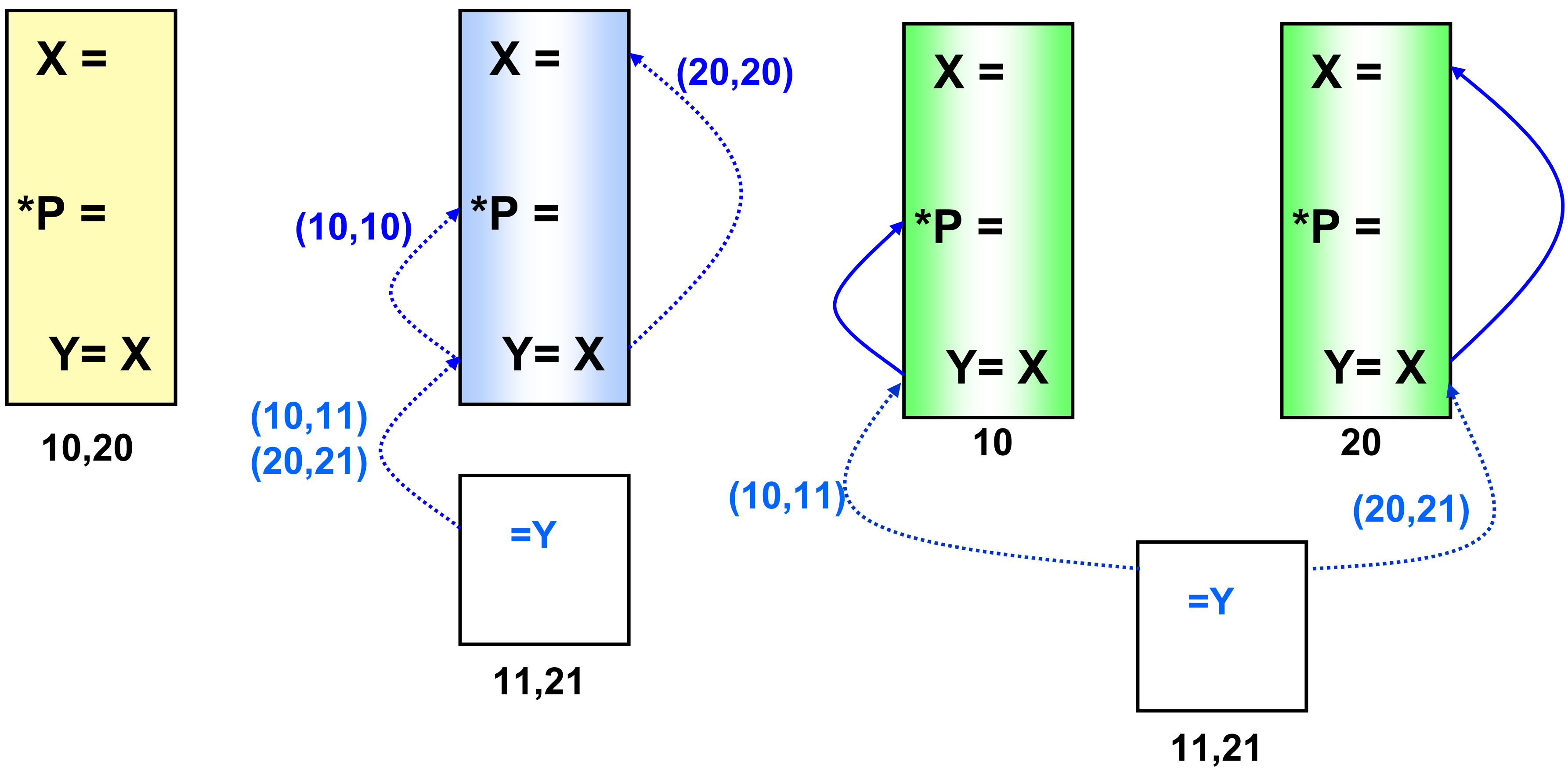
Infer: Local Dependence Labels: Full Elimination



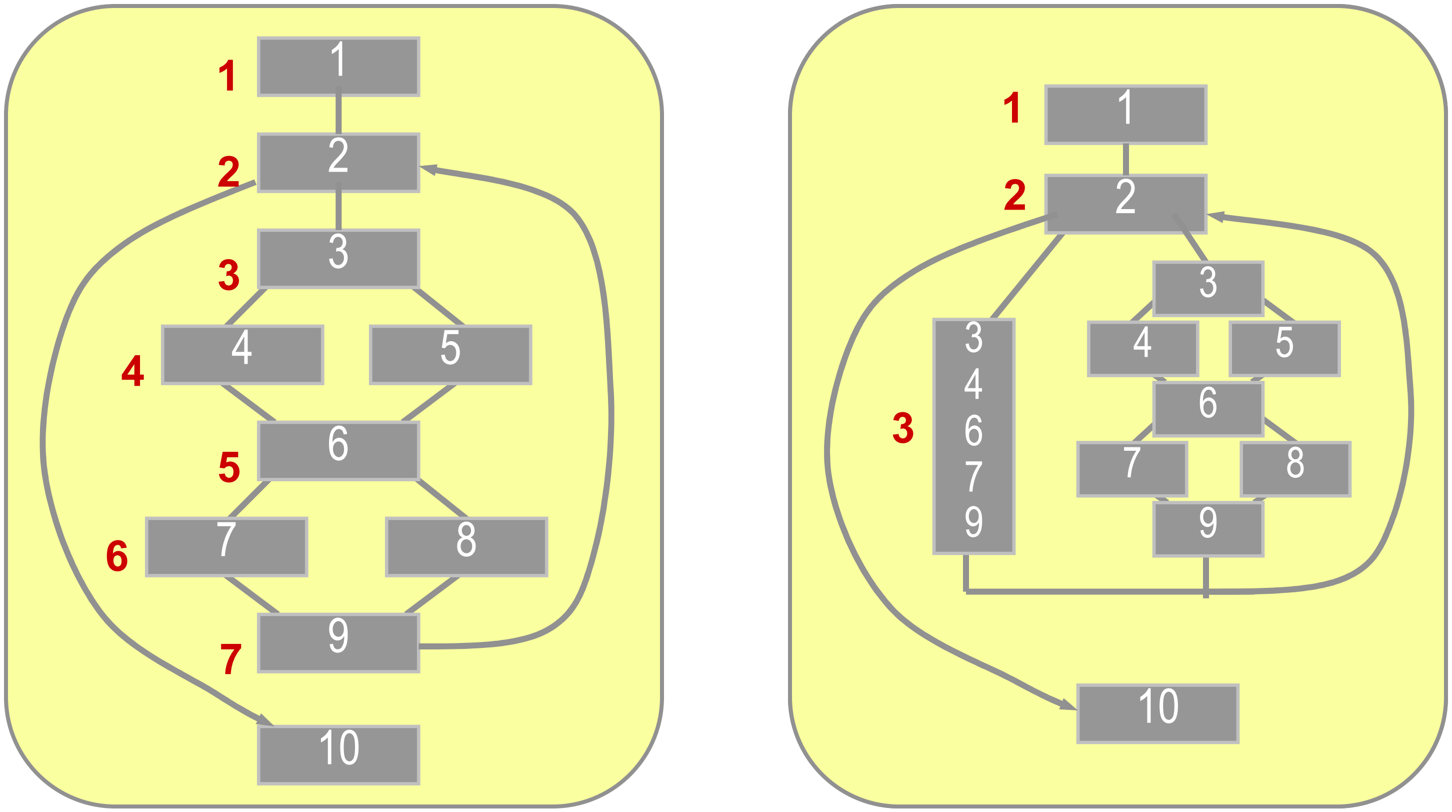
Transform: Local Dependence Labels: Elimination In Presence of Aliasing



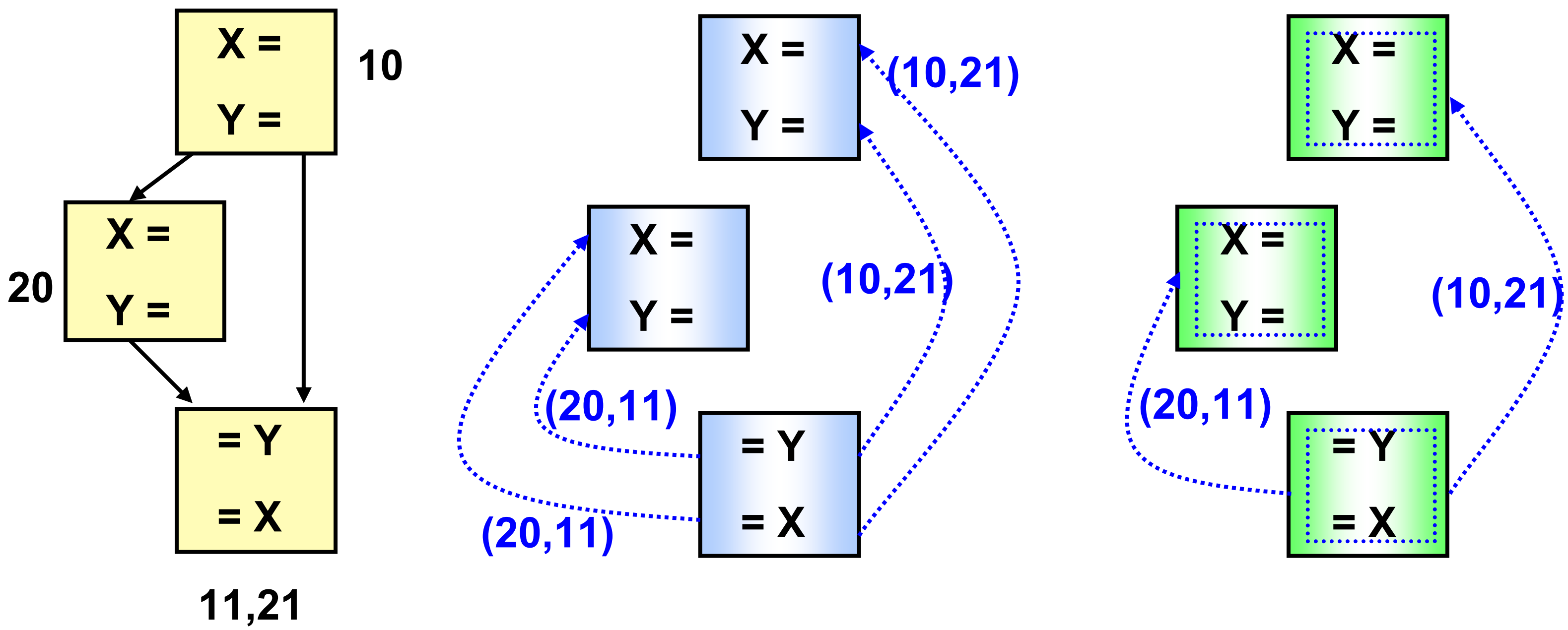
Transform: Local Dependence Labels: Elimination In Presence of Aliasing



Transform: Coalescing Multiple Nodes into One



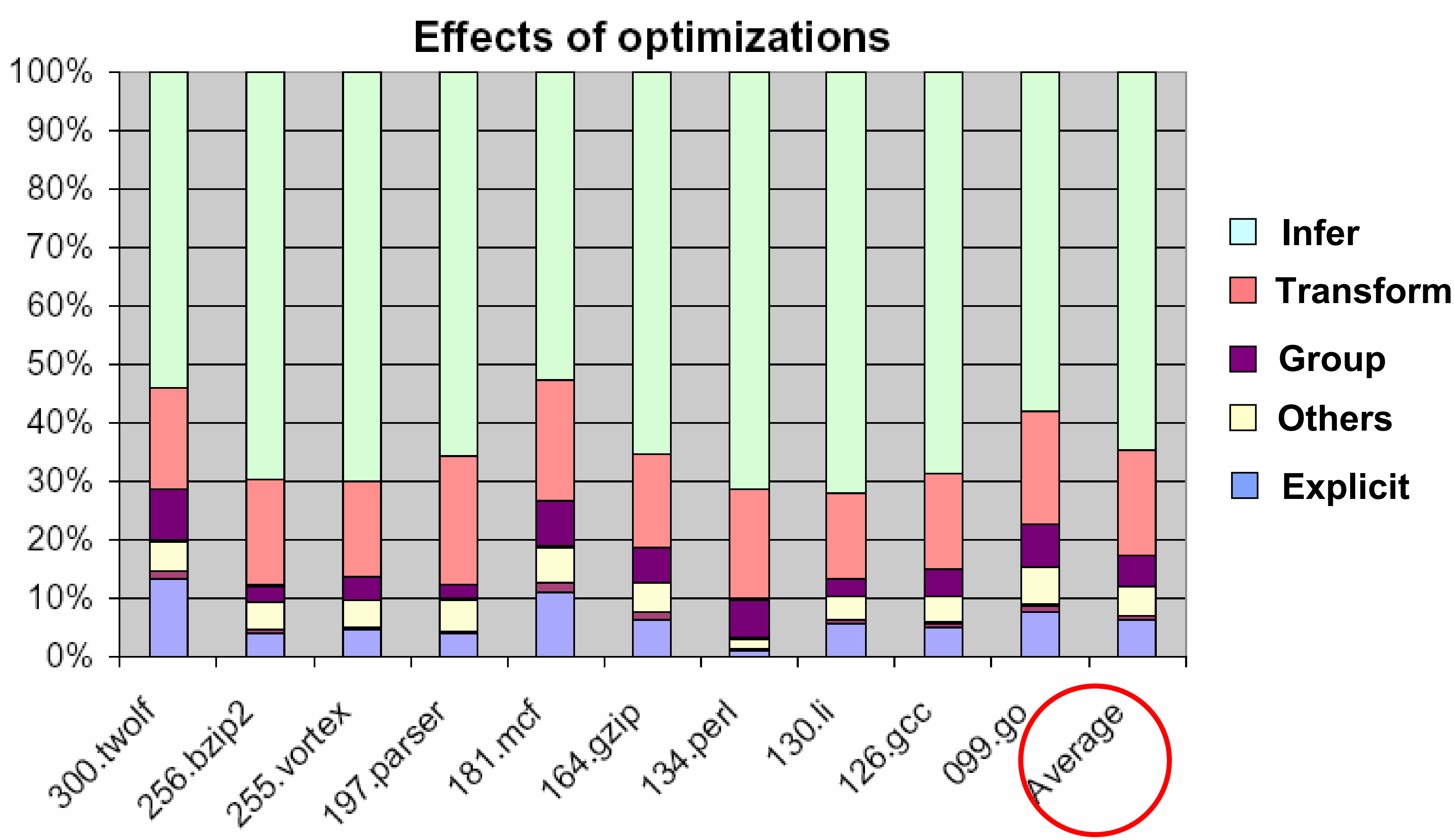
Group: Labels Across Non-Local Dependence Edges



Space: Compacted Graph Sizes

Program	Graph Size (MB)		Before / After	Explicit Dependences (%)
	Before	After		
300.twolf	1,568	210	7.72	13.40
256.bzip2	1,296	51	25.68	3.89
255.vortex	1,442	65	22.26	4.49
197.parser	1,816	70	26.03	3.84
181.mcf	1,535	170	9.02	11.09
164.gzip	835	52	16.19	6.18
134.perl	1,954	21	93.40	1.07
130.li	1,745	97	18.09	5.53
126.gcc	1,534	75	20.54	4.87
099.go	1,707	131	13.01	7.69
Average	1,543	94	25.2	6.21

Breakdowns of Different Optimizations



Efficiency: Summary

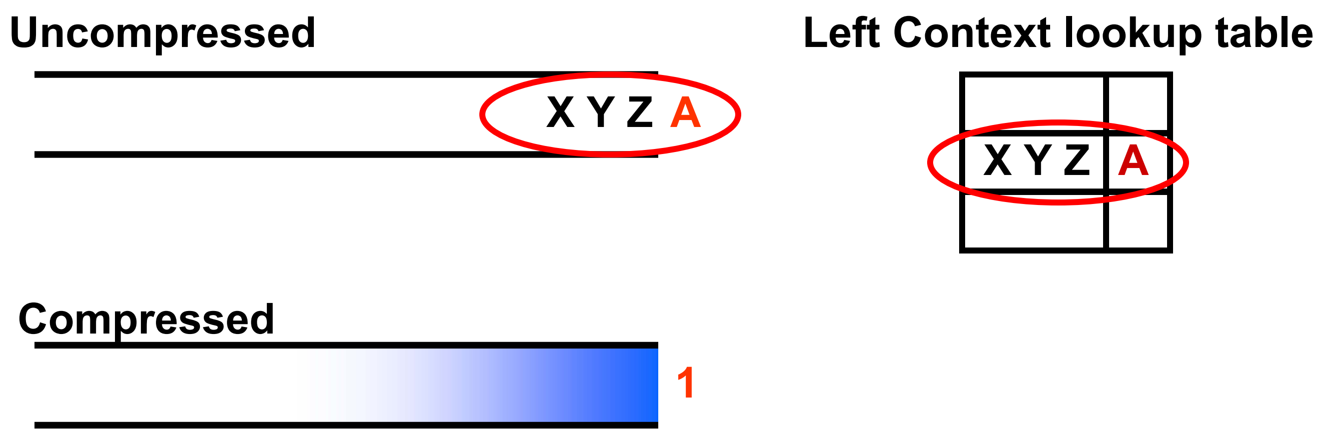
- For an execution of 130M instructions:
  - space requirement: reduced from 1.5GB to 94MB (I further reduced the size by a factor of 5 by designing a generic compression technique [MICRO'05]).
  - time requirement: reduced from >10 Mins to <30 seconds.

Generic Compression

- Traversable in compressed form
  - Sequitur
  - Context-based
    - Using value predictors;(M. Burtsher and M. Jeeradit, PACT2003)
- Bidirectional!!
  - Queries may require going either direction
  - The system should be able to answer multiple queries

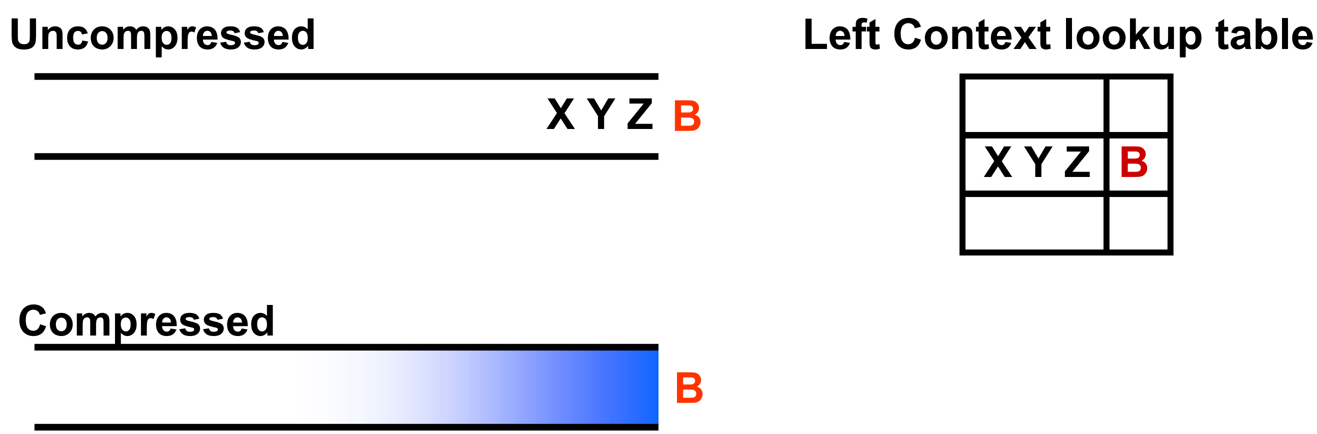
Compression using value predictors

- Value predictors
  - Last n values;
  - FCM (finite context method).
    - Example, FCM-3



Compression using value predictors

- Value predictors
  - Last n values;
  - FCM (finite context method).
    - Example, FCM-3

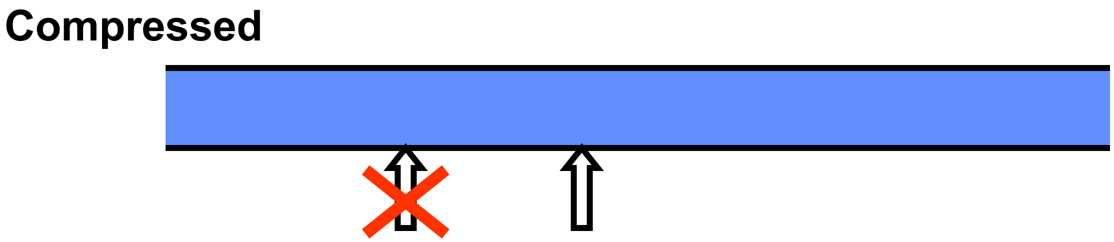


Length(Compressed) = n/32 + n*(1- predict rate)

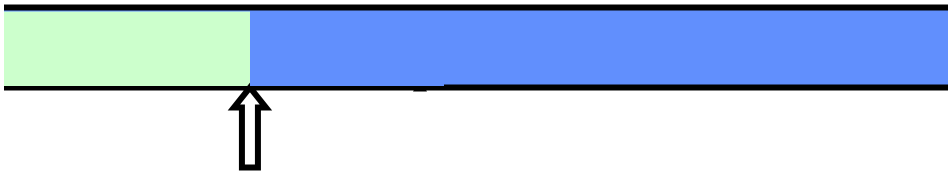
Only forward traversable;

Enable bidirectional traversal - idea

Previous predictor compression:

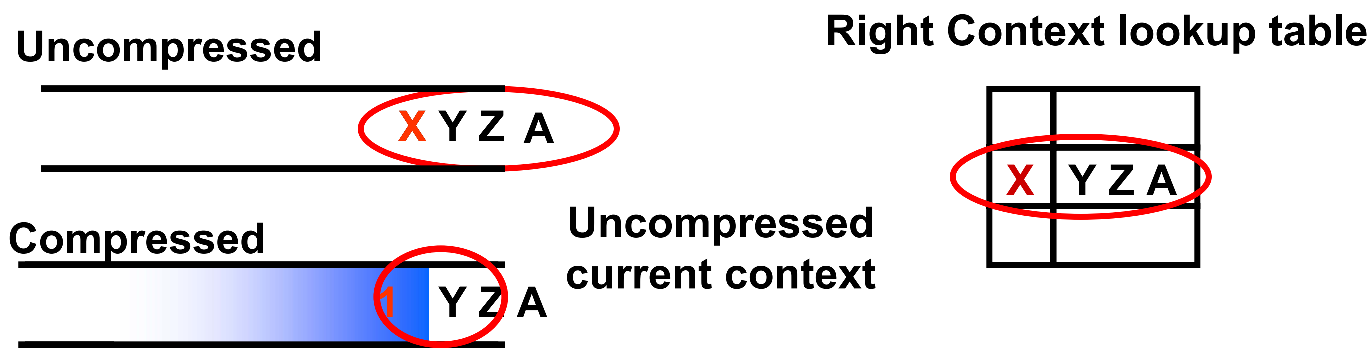


Bidirection idea:

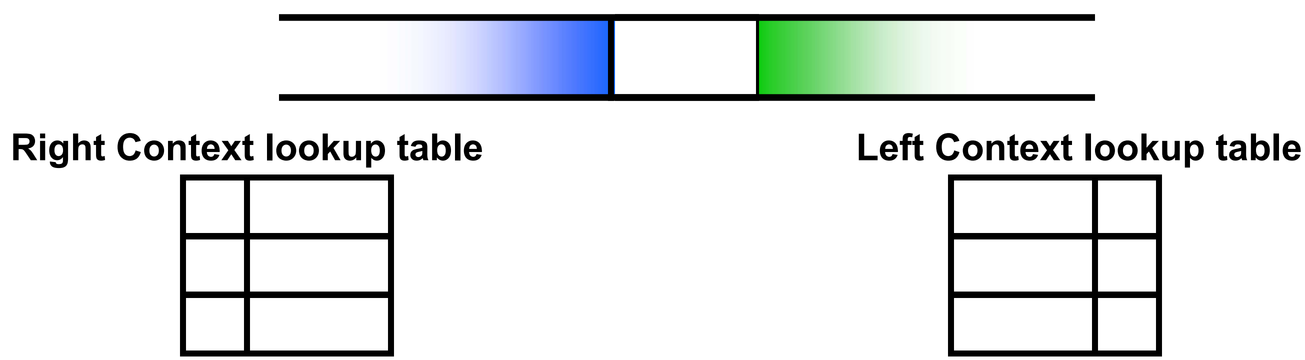


Enable bidirectional traversal

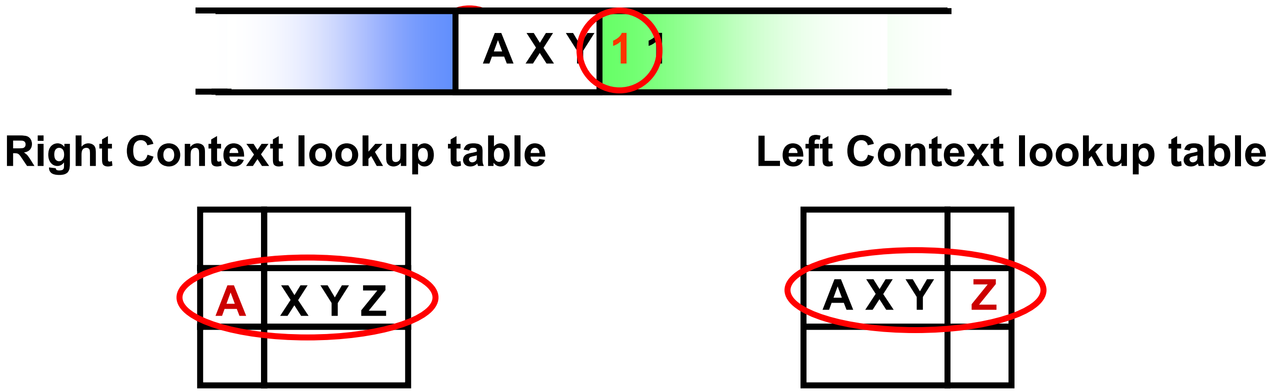
- Forward compressed, backward traversed (uncompressed) FCM
  - Traditional FCM is forward compressed, forward traversed



- Bidirectional FCM



Bidirectional FCM - example



Outline

- Slicing ABC
- Dynamic slicing
  - Dynamic slicing practices
  - Efficiency
  - Effectiveness
  - Challenges

The Real Bugs

- Nine logical bugs
  - Four unix utility programs
    - grep 2.5, grep 2.5.1, flex 2.5.31, make 3.80.
- Six memory bugs [AccMon project (UIUC)]
  - Six unix utility programs
    - gzip, ncompress, polymorph, tar, bc, tidy.

Classic Dynamic Slicing in Debugging

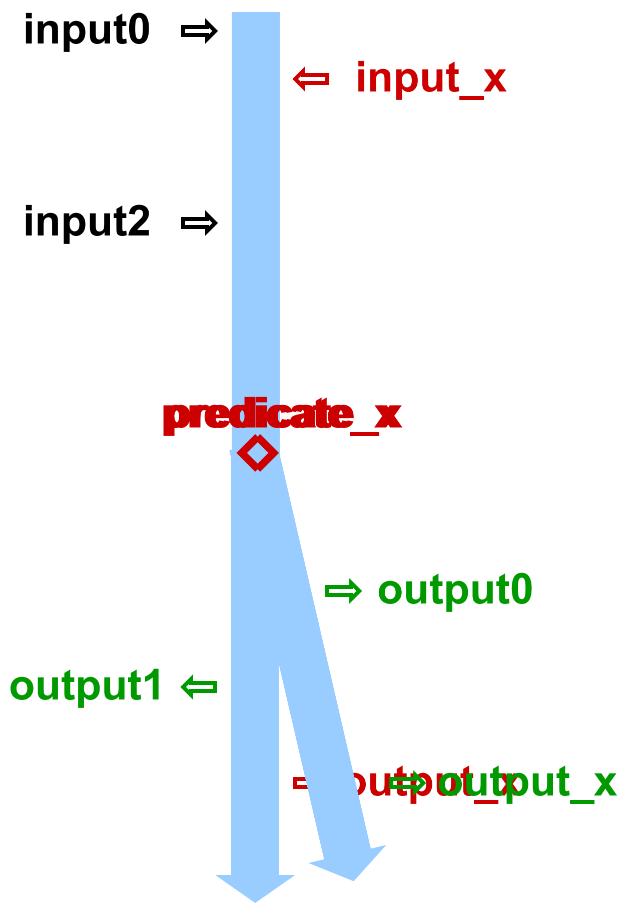
Buggy Runs	LOC	EXEC (%LOC)	BS (%EXEC)
flex 2.5.31(a)	26754	1871 (6.99%)	695 (37.2%)
flex 2.5.31(b)	26754	2198 (8.2%)	272 (12.4%)
flex 2.5.31(c)	26754	2053 (7.7%)	50 (2.4%)
grep 2.5	8581	1157 (13.5%)	NA
grep 2.5.1(a)	8587	509 (5.9%)	NA
grep 2.5.1(b)	8587	1123 (13.1%)	NA
grep 2.5.1(c)	8587	1338 (15.6%)	NA
make 3.80(a)	29978	2277 (7.6%)	981 (43.1%)
make 3.80(b)	29978	2740 (9.1%)	1290 (47.1%)
gzip-1.2.4	8164	118 (1.5%)	34 (28.8%)
ncompress-4.2.4	1923	59 (3.1%)	18 (30.5%)
polymorph-0.4.0	716	45 (6.3%)	21 (46.7%)
tar 1.13.25	25854	445 (1.7%)	105 (23.6%)
bc 1.06	8288	636 (7.7%)	204 (32.1%)
Tidy	31132	1519 (4.9 %)	554 (36.5%)

2.4-47.1% EXEC

Avg 30.9%

Looking for Additional Evidence

Buggy Execution



- Classic dynamic slicing algorithms investigate bugs through negative evidence of the *wrong output*

- Other types of evidence:
  - ☒ *Failure inducing input*
  - ☒ *Critical Predicate*
  - ☑ *Partially correct output*

- Benefits of More Evidence
  - ✓ *Narrow the search for fault*
  - ✓ *Broaden the applicability*

Negative: Failure Inducing Input [ASE'05]

iname[1025]: aaaaaaaaa...aaaaa

strcpy.c

```
           ...
(2) → 40:  for (; (*to = * from)!=0; ++from; ++to);
           ...
```

gzip.c

```
             ...
      193:   char *env;
      198:   CHAR ifname[1024];
             ...
(1) → 844:   strcpy (ifname, iname);
             ...
(3) → 1344:  ... free(env), ...
```

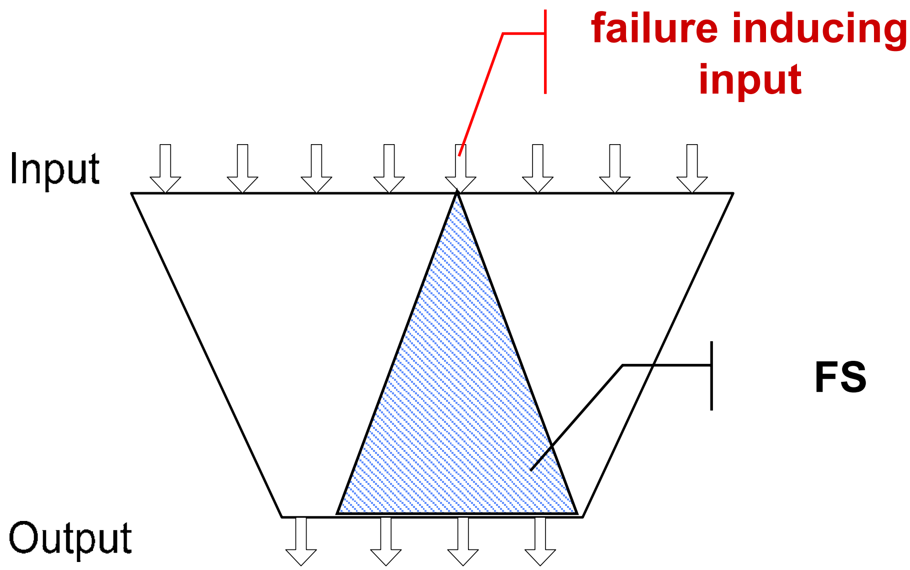
X

The rationale

Negative: Failure Inducing Input [ASE'05]

Given a failed run:

- Identify a minimal failure inducing input ([Delta Debugging - Andreas Zeller])
  - This input should affect the root cause.
- Compute forward dynamic slice (FS) of the input identified above



(a)

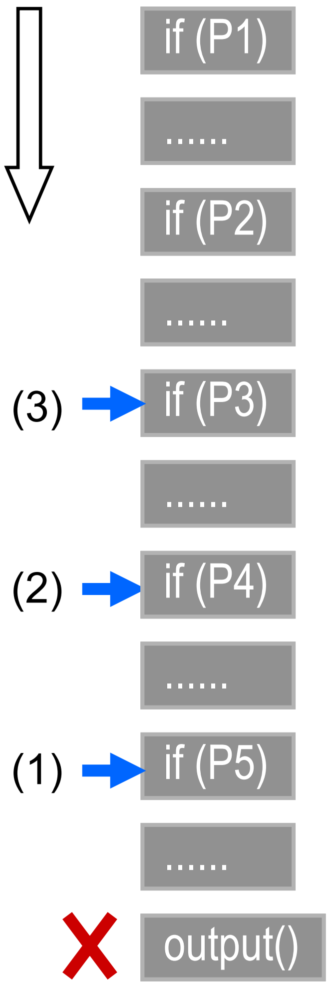
Negative: Critical Predicate [ICSE'06]

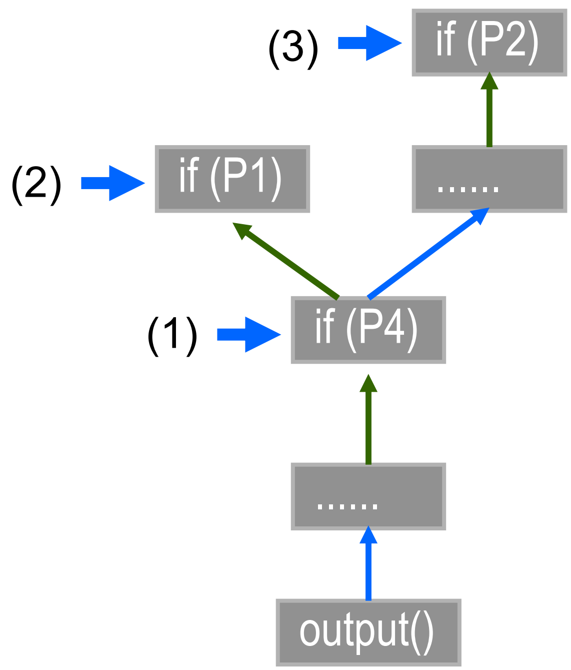
```
base = …
. . .
base[…] = ...
. . .
for ( i = 0; i <= lastdfa; ++i )
    {
    . . .
    int offset = base[i+1];
    . . .
    chk[offset] = EOB_POSITION;
    . . .
    chk[offset - 1] = ACTION_POSITION;
    . . .
    }

for ( i = 0; i <= tblend; ++i )
    {
    . . .
    else if ( chk[i] == ACTION_POSITION )
        printf("%7d, %5d,", 0 , …);
    . . .
    else    /* verify, transition */
        printf("%7d, %5d," , chk[i], …);
    . . .
    }
```

The rationale

Searching Strategies

Execution Trace:





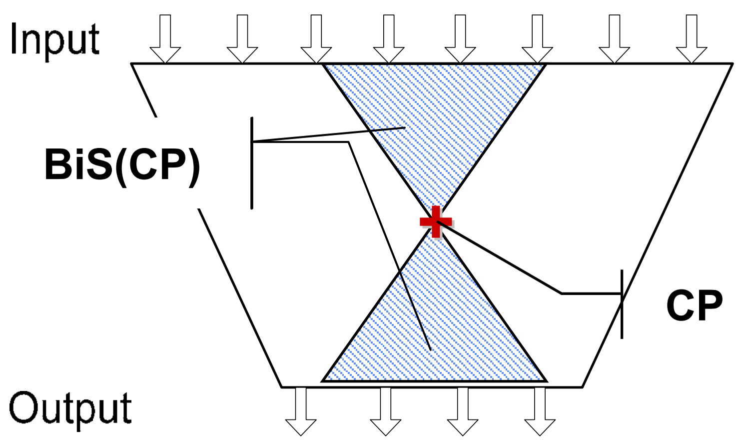
Dependence Distance Ordering

Control Flow Distance Ordering

Slicing with Critical Predicate

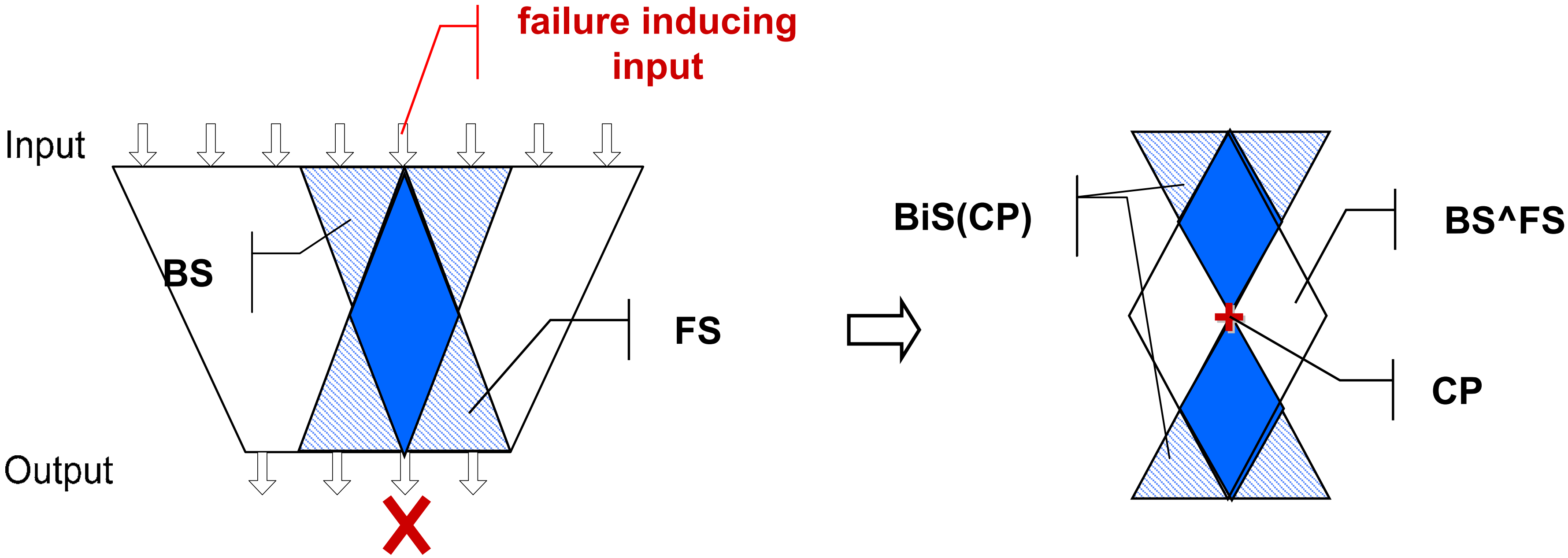
Given a failed run:

- Identify the critical predicate
  - ❖ The critical predicate should AFFECT / BE AFFECTED BY the root cause.
- Compute bidirectional slice (BiS) of the critical predicate



(a)

All Negative Evidence Combined



Negative Evidences Combined in Slicing

Buggy Runs	BS	BS^FS^BiS (%BS)
flex 2.5.31(a)	695	27 (3.9%)
flex 2.5.31(b)	272	102 (37.5%)
flex 2.5.31(c)	50	5 (10%)
grep 2.5	NA	86 (7.4%*EXEC)
grep 2.5.1(a)	NA	25 (4.9%*EXEC)
grep 2.5.1(b)	NA	599 (53.3%*EXEC)
grep 2.5.1(c)	NA	12 (0.9%*EXEC)
make 3.80(a)	981	739 (81.4%)
make 3.80(b)	1290	1051 (75.3%)
		Average=36.0% * (BS)
gzip-1.2.4	34	3 (8.8%)
ncompress-4.2.4	18	2 (14.3%)
polymorph-0.4.0	21	3 (14.3%)
tar 1.13.25	105	45 (42.9%)
bc 1.06	204	102 (50%)
tidy	554	161 (29.1%)

Positive Evidence

- Correct outputs produced in addition to wrong output.
- $BS(O_{wrong}) – BS(O_{correct})$ is problematic.

```
……
10. A = 1 (Correct: A=3)
……
20. B = A % 2
……
30. C = A + 2
……
40. Print (B)
41. Print (C)
```

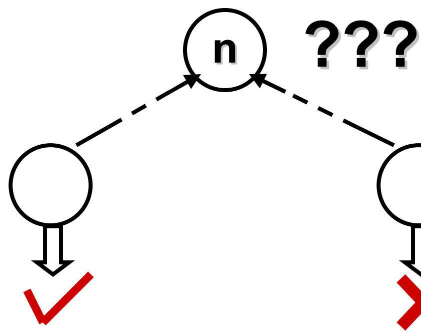
BS(C@41)= {10, 30, 41}

BS(B@40)= {10, 20, 40}

BS(C@41)-BS(B@40)

= {30,41}

Confidence Analysis [PLDI'06]



- Assign a confidence value to each node, *C(n) = 1* means *n* must contain the correct value, *C(n) = 0* means there is no evidence of *n* having the correct value. Given a threshold *t*, BS should only contain the nodes *C(n) < t* .
  - If a node *n* can only reach the correct output, *C(n) = 1*.
  - If a node *n* can only reach the wrong output, *C(n) = 0.*
  - If a node *n* can reach both the correct output and the wrong output, the CONFIDENCE of the node *n* is defined as:

$$C(n) = 1 - \log_{|range(n)|} |Alt(n)|$$

  - *Alt(n)* is a set of possible LHS values at *n*, assigning any of which to *n* does not change any same correct output.
    - $|Alt(n)| >= 1$;
    - $C(n)=1$ when $|Alt(n)| = 1$.

Confidence Analysis: Example

- If a node *n* can only reach only the correct output, *C(n) = 1*.
- If a node *n* can only reach the wrong output, *C(n) = 0.*
- If a node *n* can reach both the correct output and the wrong output, the CONFIDENCE of the node *n* is defined as:

$$C(n) = 1 - \log_{|range(n)|} |Alt(n)|$$

- *Alt(n)* is a set of possible LHS values at *n*, assigning any of which to *n* produces the same correct output.

……

10. A = 1 (Correct: A=3) $C(10) = 1 - \log_{|range(A)|} \frac{|range(A)|}{2} = \log_{|range(A)|} 2$

……

20. B = A % 2 $C(20) = 1$

……

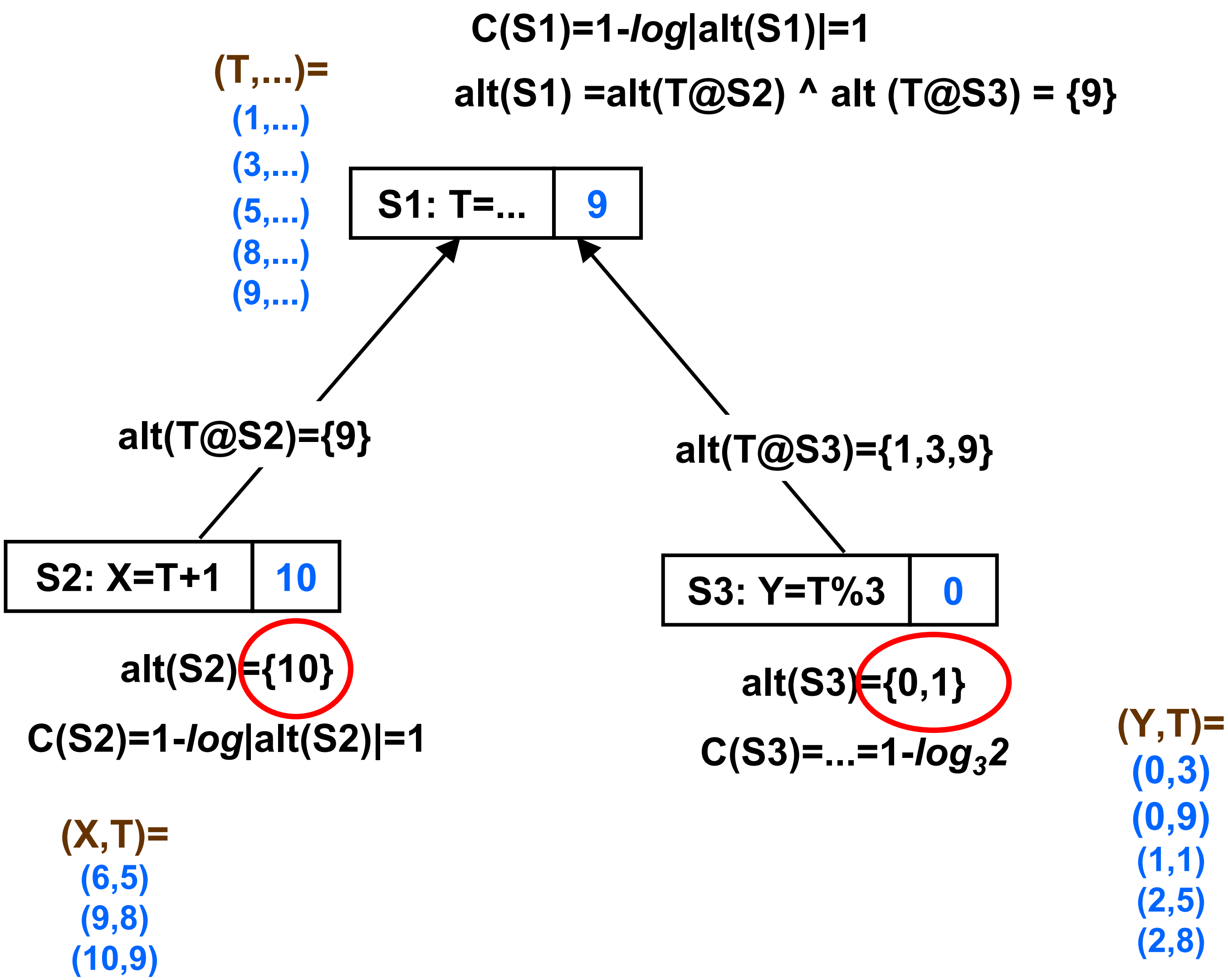
30. C = A + 2 $C(30) = 0$

……

40. Print (B) $C(40) = 1$

41. Print (C) $C(41) = 0$

Computing Alt(n)



Evaluation on injected bugs

- We pruned the slices by removing all the statements with C(n)=1

Program	BS	Pruned Slice	Pruned Slice / BS
print_tokens	110	35	31.8%
print_tokens2	114	55	48.2%
replace	131	60	45.8%
schedule	117	70	59.8%
schedule2	90	58	64.4%
gzip	357	121	33.9%
flex	727	27	3.7%

Average=41.1%

Effectiveness

- BS=30.9% *EXEC
- BS^FS^BiS = 36% * BS
  - For many memory type bugs, slices can be reduced to just a few statements.
- Pruned Slice = 41.1% * BS
  - For some benchmarks, the pruned slices contain only the dependence paths leading from the root cause to the wrong output.

Comments

- False positive
  - FS > PS / Chop > DS
- False negative
  - DS > FS=PS=Chop
- Cost
  - PS/Chop > FS > DS

Challenges

- Execution omission errors

Input x=-1

```
y=10
if (x>0) /*error, should be x<0*/
    y=y+1
print(y)
```

- For long running programs, multithreading programs

- Making slices smaller
  - More evidence?

Next

- Background (done)
- Ideas, papers (start from next lecture)
- Will try to schedule a lecture on static tools.
  - Probably in late March.