プログラムスライス

井上克郎

大阪大学/

奈良先端科学技術大学院大学

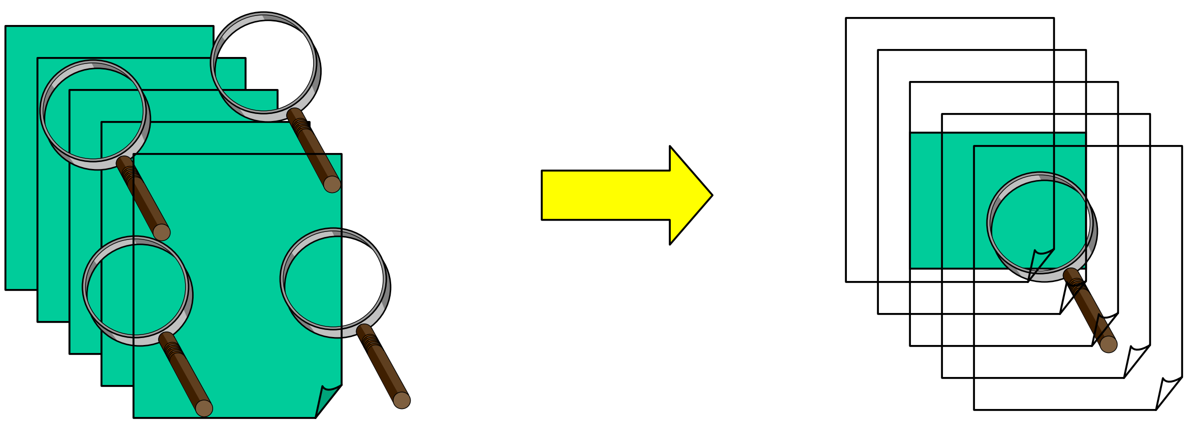
Empirical Evaluation of the Program Slicing for Fault Localization

Background of Research

- Software Systems are becoming large and complex
  - Debugging, testing, and maintaining costs are increasing
- To reduce development costs, techniques for improving efficiency of such activities are essential

Localization

- Handling large source programs is difficult
- If we could select specific portions in the source programs and we can concentrate our attentions only to those portions, the performance of the activities would increase



Program Slicing

- A technique of extracting all program statements affecting the value of a variable
- Specify a variable concerned and extract the affecting statements
- Developers can concentrate their attentions to the extracted statements

Slicing: Extraction

Slice: Collection of extracted statements

Static Slicing

- All statements possibly affecting the value of *Slice Criterion* (a variable concerned)
- Method

  (1) Construct Program Dependence Graph (PDG)

  Nodes: statements in program

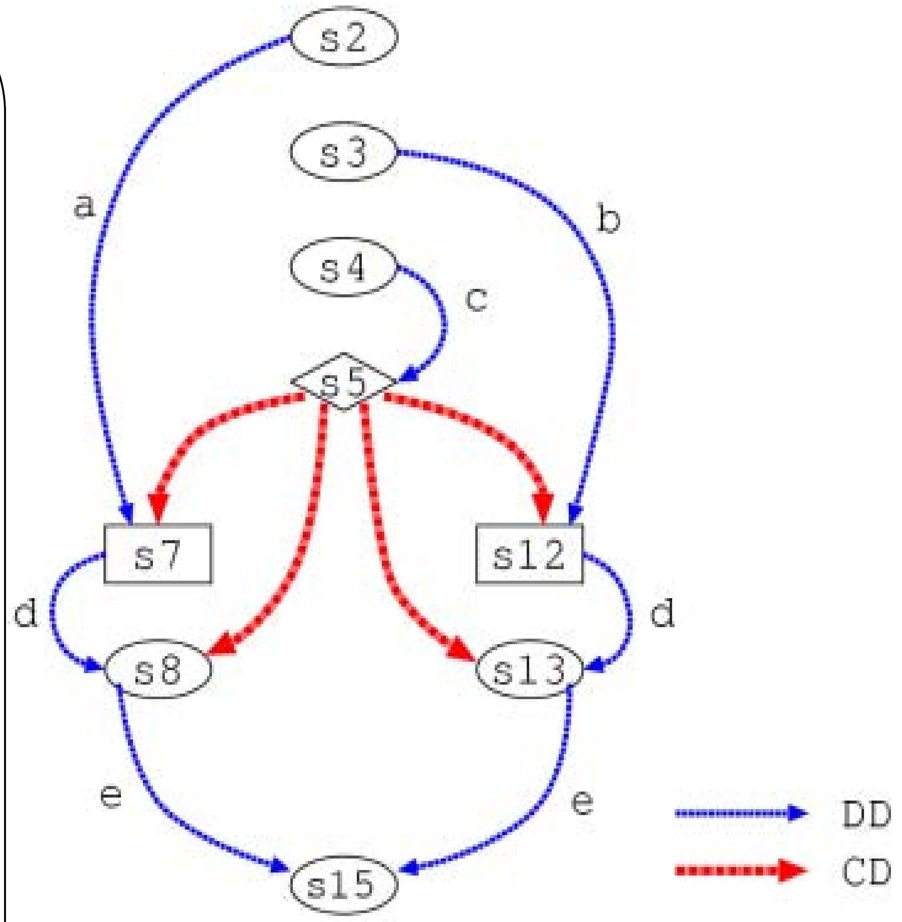
  Edges:

  - *Data Dependence (DD)*: variable definition and its reference
  - *Control Dependence (CD)*: predicate and statement dominated by the predicate

  (2) Collect all reachable nodes on PDG to a slice criterion (statement, variable)

Example of PDG

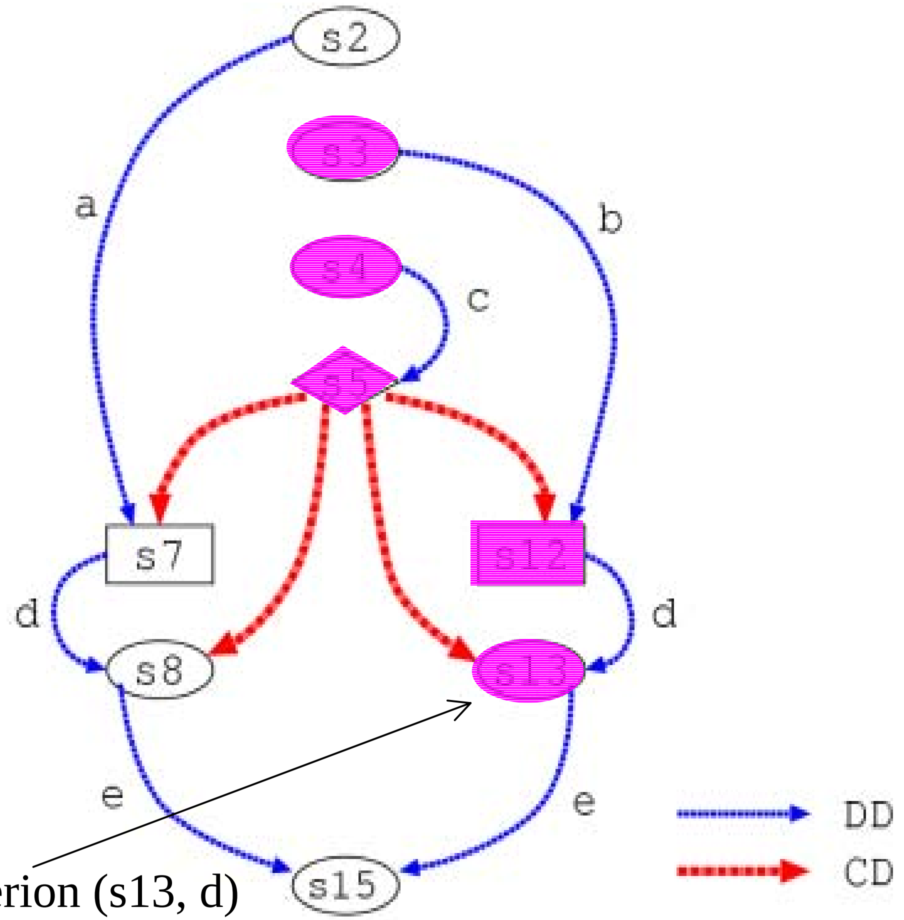
```
s1  begin
s2    a:=3;
s3    b:=3;
s4    readln(c);
s5    if c=0 then
s6    begin
s7      d:=functionA(a);
s8      e:=d
s9    end;
s10   else
s11   begin
s12     d:=functionB(b);
s13     e:=d
s14   end;
s15   writeln(e)
s16 end.
```



Example of Static Slice

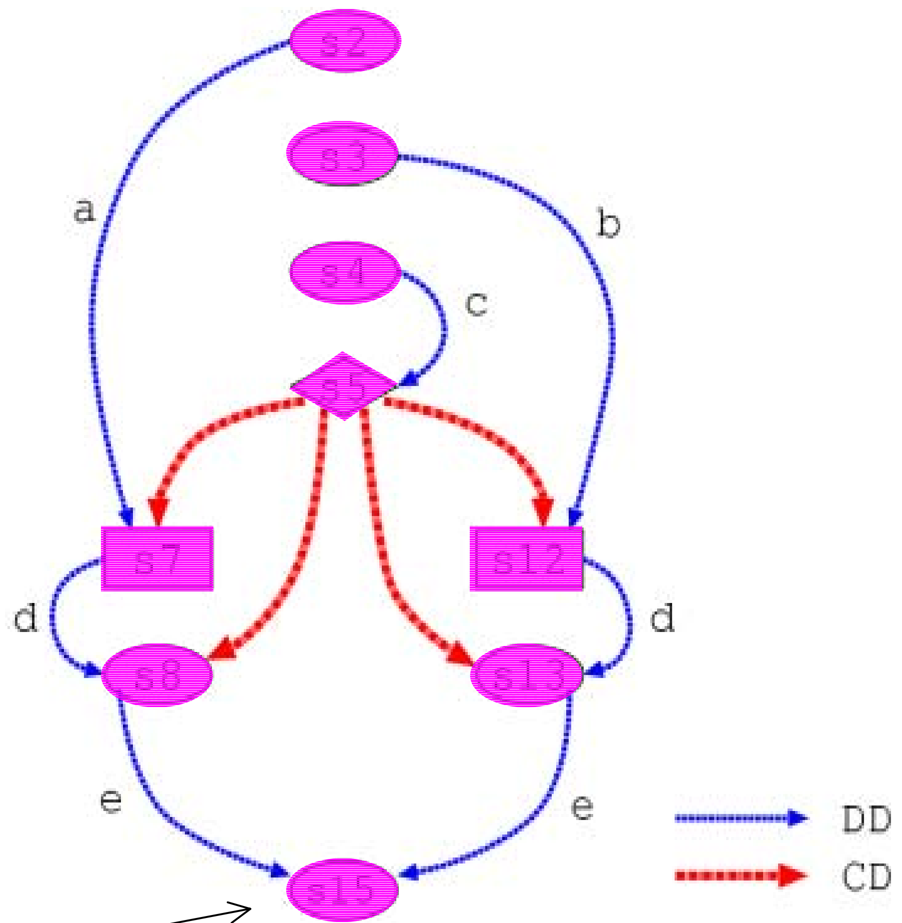
```
s1  begin
s2    a:=3;
s3    b:=3;
s4    readln(c);
s5    if c=0 then
s6    begin
s7      d:=functionA(a);
s8      e:=d
s9    end;
s10   else
s11   begin
s12     d:=functionB(b);
s13     e:=d
s14   end;
s15   writeln(e)
s16 end.
```

Slicing criterion (s13, d)



Example of Static Slice (2)

```
s1 begin
s2   a:=3;
s3   b:=3;
s4   readln(c);
s5   if c=0 then
s6   begin
s7     d:=functionA(a);
s8     e:=d
s9   end;
s10  else
s11  begin
s12    d:=functionB(b);
s13    e:=d
s14  end;
s15  writeln(e)
s16 end.
```

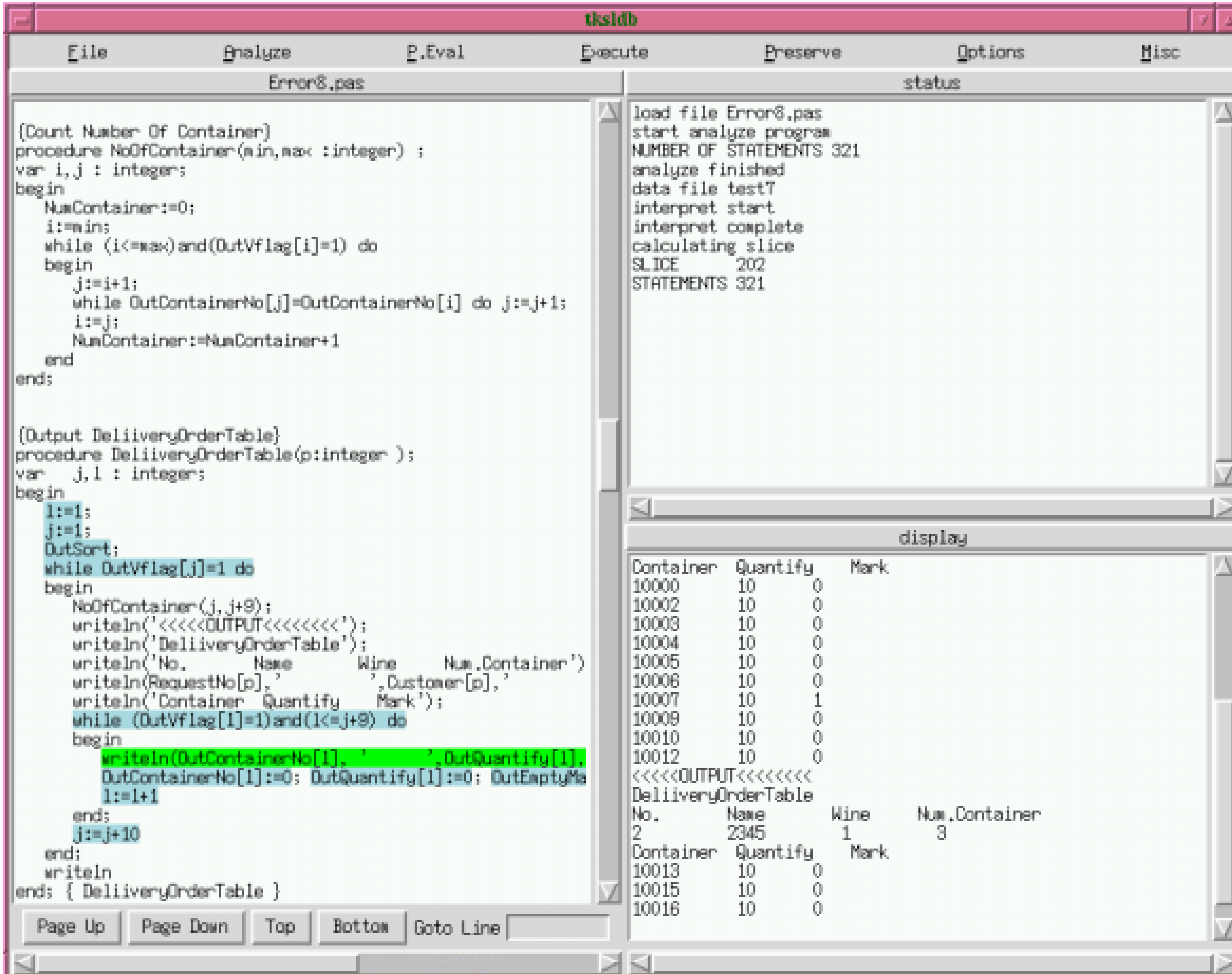


Slicing criterion (s15, e)

Debug Supporting Tool

- Target language: subset of Pascal
  - conditional, assignment, iterative, input/output, procedure-call, compound statement etc.
  - variables : integer, string, boolean, and arrays of them.
- Functions:
  - Calculate static slice.
  - Step execution, referring the values of variables, setting the breakpoints, etc

Snapshot of Osaka Slicing System



Objective

- We aim to empirically evaluate the potential usefulness of the program slicing to the fault localization.

Process of Experiment

- Step1: To conduct the experiments efficiently, we construct software debugging support tool based on the static slicing.

- Step2: We conduct two experimental projects to evaluate the usefulness of the slicing for fault localization.
  - Experiment 1: (with debugging support tool)
  - Experiment 2:

Experiment 1
Objective

- We empirically evaluate the following two hypotheses:
  - (H1)Using slicing technique reduces the fault localizing effort.
  - (H2)There exists some kinds of faults that are localized effectively.

Experiment 1 Overview

	Group1 (Three subjects)	Group2 (Three subjects)
Trial1	PG1/Slicing-based fault localization	PG1/Conventional debugger-based fault localization
Trial2	PG2/Conventional debugger-based fault localization	PG2/ Slicing-based fault localization

Experiment 1
Programs

- W used two programs (PG1 and PG2) which were developed based on the same specification for the inventory control program at wine shop.

- Since they were developed independently, their data structures and the algorithms were not identical.

Experiment 1
Type of Faults

Faults in PG1

- Lack of the output processing,
- Illegal assignment,
- Illegal conditional statement,
- Omission of the initialization,
- Lack of the procedure call,
- Wrong data renovation,
- Wrong parameter for the procedure call, and
- Wrong execution order for some procedure calls.

Faults in PG2

- Illegal conditional statement,
- Illegal conditional statement,
- Wrong reference to array variable,
- Wrong execution order for some procedure calls,
- Wrong parameter for the procedure call,
- Lack of the procedure call,
- Wrong data renovation,
- Illegal output,
- Wrong registration for database.

Experiment 1
- Analysis for (H1) -

	Group1	Group2
Trial1	122 (min.) (slice)	155 (min.) (no slice)
Trial2	133 (min.) (no slice)	114 (min.) (slice)

The group that used the slicing technique could localize the faults effectively.

Experiment 1
- Analysis for (H2) -

	Type of fault	Average time to localize
Trial1	Illegal conditional statement	Group1: 14 (min.) Group2: 33 (min.)
	Lack of procedure call	Group1: 19 (min.) Group2: 34 (min.)
	Wrong data renovation	Group1: 12 (min.) Group2: 19 (min.)
Trial2	Illegal conditional statement	Group2: 19 (min.) Group1: 34 (min.)
	Wrong registration for database	Group2: 12 (min.) Group1: 19 (min.)

This difference is confirmed by the the Welch test (α=0.05)

Slicing is effective to localize these faults.

Experiment 2
- Objective -

- In Experiment 1, we could not collect enough subjects to statistically confirm all hypotheses because of its expensiveness.

- To resolve this limitation, we have also carried out an inexpensive experiment, called Experiment 2, which aimed to examine usefulness of the slicing to the fault localization for small scale programs with more subjects and less management effort.

Experiment 2
Overview

	Group1 (15 subjects)	Group2 (19 subjects)
Target	Six programs (P1-P6) Slicing-based fault localization	Six programs(P1’-P6’) /Conventional debugger-based fault localization

P1-P6: programs with slicing information
P1’-P6’: only programs

Experiment 2
Programs

- Six kinds of Pascal programs each of which includes one fault (illegal conditional or illegal assignment statement)
  – (P1)Factorization,
  – (P2)Decision whether the input number is a prime number,
  – (P3)Construction of a Triangle of Pascal,
  – (P4)Numerical operations,
  – (P5)Permutation ,
  – (P6)Sorting.

Experiment 2
- Analysis for (H1) -

	Group1	Group2
Average time	40.73 (min.) (slice)	49.11(min.) (no slice)

This difference is confirmed by the the Welch test (α=0.05)

The group that used the slicing technique could localize the faults effectively.

Experiment 2
- Analysis for (H2) -

Type of fault	Average time to localize
Illegal conditional statement in P3	Group1: 7.13 (min.) Group2: 11.63min.)
Illegal conditional statement in P6	Group1: 3.07 (min.) Group2: 4.53 (min.)

These faults are included in such programs that it is very difficult to grasp the correspondence its algorithm to its code.

Findings

- We have empirically evaluated the potential usefulness of the program slicing to the fault localization.

- Number of subjects are small. However, we would say that the program slicing is useful for the fault localization.

Lightweight Semi-Dynamic Slicing Methods

Dynamic Slicing

- All statements actually affecting the value of a slice criterion for <u>an execution with a particular input data</u>
- Useful for debugging with testcase
- Method

  (1) Execute program with an input data and record the execution trace

  (2) Determine DD and CD on each statement of the trace

  (3) Collect reachable statements to a slice criterion (input-data, execution-point, variable)

Example of Dynamic Slicing

```
s1  begin
s2    a:=3;
s3    b:=3;
s4    readln(c);
s5    if c=0 then
s6    begin
s7      d:=functionA(a);
s8      e:=d
s9    end;
s10   else
s11   begin
s12     d:=functionB(b);
s13     e:=d
s14   end;
s15   writeln(e)
s16 end.
```

Source

```
e1  begin
e2    a:=3;
e3    b:=3;
e4    readln(c);
e5    if c=0 then
e6    begin
e7      d:=functionA(a);
e8      e:=d
e9    end;





e15   writeln(e)
e16 end.
```

(1) Execute Trace with Input c=0

Example of Dynamic Slicing (cont.)

```
e1  begin
e2    a:=3;
e3    b:=3;
e4    readln(c);
e5    if c=0 then
e6    begin
e7      d:=functionA(a);
e8      e:=d
e9    end;
e15   writeln(e)
e16 end.
```

Slicing Criterion (c=0, s15, e)

```
s1  begin
s2    a:=3;
s3    b:=3;
s4    readln(c);
s5    if c=0 then
s6    begin
s7      d:=functionA(a);
s8      e:=d
s9    end;
s10   else
s11   begin
s12     d:=functionB(b);
s13     e:=d
s14   end;
s15   writeln(e)
s16 end.
```

DD

CD

(2) Determine DD and CD

(3) Collect Statements

Static and Dynamic Slicing

- Analysis cost: static < dynamic
  - Recording execution trace is exhaustive
  - Determining DD & CD on execution trace is expensive

PDG << Execution Trace

- Slice size: static > dynamic
  - Static slicing considers all possible flows
  - Dynamic slicing only considers one trace

Unifying Static and Dynamic Information

Static information

+

Lightweight dynamic information

⬇

Efficient and effective slicing

Approach to Call-Mark(CM) Slicing

- (static slice — unexecuted program statements)
- Unexecuted statements are explored by
  - Checking activation of procedure/function calls
  - Delete unexecuted call statements and associated statements
  - The associated statements: execution dependency (statically determined)

Execution Dependency and CED

- s_1 is *executionally dependent* (ED) on s_2 iff s_1 cannot be executed when s_2 is not executed
  - Easily obtained by flow analysis

```
s2
...
...
s1
```

- CED(s) is a set of caller statements on which s is executionally depending
  - If any of CED(s) is known to be unexecuted, then we know that s is never executed
  - Also by flow analysis

```
functionX
...
functionY
...
...
s
...
```

Example of CED

```
s1   functionA ;
s2   if a=1 then
s3         begin
s4         b:= c;
s5         functionB ;
```

ED
(part of)

CED(s2) = {s1}

CED(s4) = {s1, s5}

Steps for Call-Mark Slicing

(1) Construct PDG and compute CED for each statement (pre-execution analysis)

(2) Prepare a flag for each call statement, and execute program with input data. Mark the flag if the call statement is executed

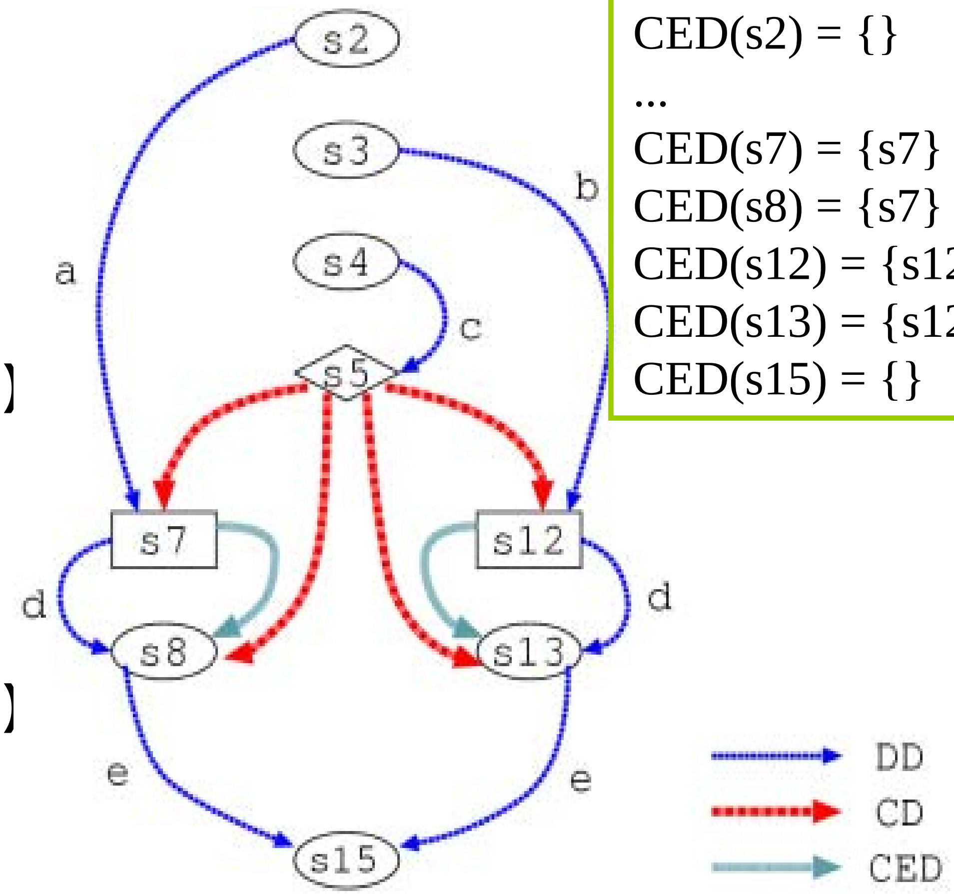
(3) Delete unexecuted nodes and associated edges from PDG

if any flag in CED(*s*) is not marked, *s* is know to be unexecuted

(4) Collect reachable statements to a slice criterion

Example of Call-Mark Slice (Step1)

```
s1  begin
s2    a:=3;
s3    b:=3;
s4    readln(c);
s5    if c=0 then
s6    begin
s7      d:=functionA(a)
s8      e:=d
s9    end;
s10   else
s11   begin
s12     d:=functionB(b)
s13     e:=d
s14   end;
s15   writeln(e)
s16 end.
```



CED(s2) = {}
...
CED(s7) = {s7}
CED(s8) = {s7}
CED(s12) = {s12}
CED(s13) = {s12}
CED(s15) = {}

Example of Call-Mark Slice (Step2)

```
s1  begin
s2    a:=3;
s3    b:=3;
s4    readln(c);
s5    if c=0 then
s6    begin
s7      d:=functionA(a);
s8      e:=d
s9    end;
s10   else
s11   begin
s12     d:=functionB(b);
s13     e:=d
s14   end;
s15   writeln(e)
s16 end.
```

Execution with input c=0

X Flag(s7)

Flag(s12)

Flag(s7) : marked
Flag(s12) : not marked

Example of Call-Mark Slice(Step3 & 4)

CED(s2) = {}

...

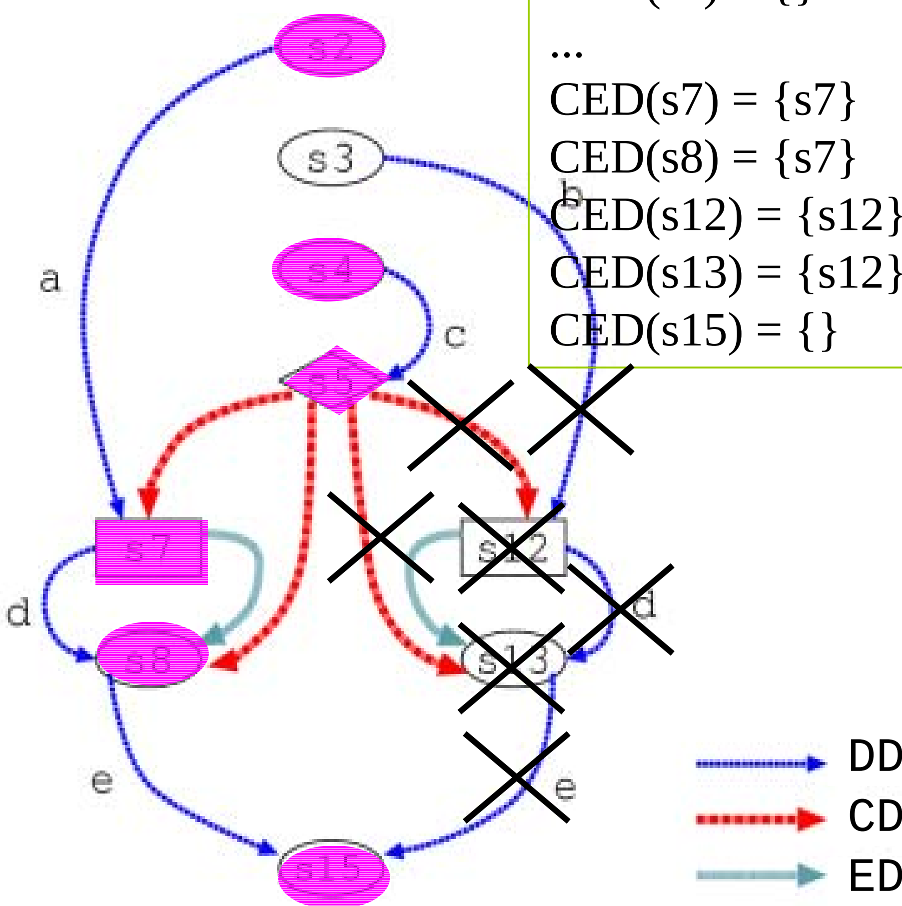
CED(s7) = {s7}

CED(s8) = {s7}

CED(s12) = {s12}

CED(s13) = {s12}

CED(s15) = {}



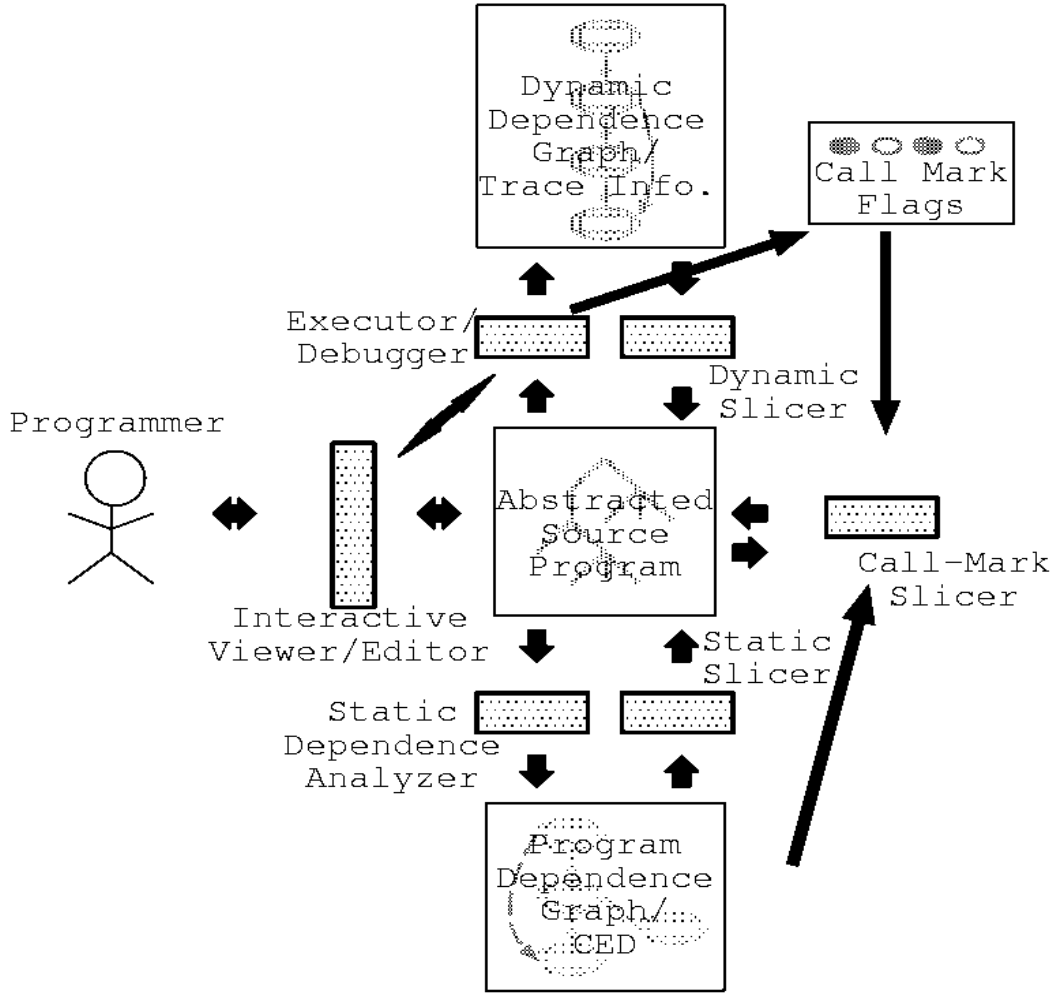

```
s1  begin
s2    a:=3;
s3    b:=3;
s4    readln(c);
s5    if c=0 then
s6    begin
s7      d:=functionA(a);
s8      e:=d
s9    end;
s10   else
s11   begin
s12     d:=functionB(b);
s13     e:=d
s14   end;
s15   writeln(e)
s16 end.
```

Slice criterion (c=0, s15, e)

Implementation of Call-Mark Slicing

- Implement steps (1) - (4)
  - flag <--> each call statement
- Flags are not necessary to be associated with caller codes
- Modify calling mechanism and do not modify other codes
  - Steal the return addresses from the calling stack
  - Determine which caller statements are actually executed

Architecture of Osaka Slicing System



Osaka Slicing System

Approach to Dependence-Cache Slicing

- Limitation of static analyses for arrays and pointer variables

```
1:  a[0]:=0;
2:  a[1]:=3;
3:  readln(b);
4:  a[b]:=2;
5:  c:=a[0]+4;
6:  writeln(c);
```

Overview of Dependence-Cache Slicing

- Control dependences are analyzed statically
- Data dependences are collected dynamically at program execution
  - Use dependence cache for each variables
- PDG_{DC} is constructed when program halts
- PDG_{DC} is traversed from a slice criterion

Dependence Analysis

Input: b=0

```
      1:  a[0]:=0;
      2:  a[1]:=3;
b     3:  readln(b);
a[0]  4:  a[b]:=2;
c     5:  c:=a[0]+4;
      6:  writeln(c);
```

Change of Caches

	a[0]	a[1]	b	c
1:	1	-	-	-
2:	1	2	-	-
3:	1	2	3	-
4:	4	2	3	-
5:	4	2	3	5
6:	4	2	3	5

Evaluation (1)

- Experiments with several sample programs

Size of Various Slices (lines of code)

Program	Static	Call-Mark	D-Cache	Dynamic
P1(85lines)	21	17	15	5
P2(387lines)	182	162	16	5
P3(871lines)	187	166	61	8

Evaluation (2)

Pre-Execution Analysis Time

(*ms* by Celeron 450MHz with 128MB)

Program	Static	Call-Mark	D-Cache	Dynamic
P1	11	14	5	N/A
P2	213	215	19	N/A
P3	710	698	48	N/A

Evaluation (3)

Execution Time
(*ms* by Celeron 450MHz with 128MB)

Program	Static	Call-Mark	D-Cache	Dynamic
P1	47	47	51	174
P2	43	43	45	4,540
P3	4,700	4,731	4,834	216,464

Evaluation (4)

Slice Construction Time

(*ms* by Celeron 450MHz with 128MB)

Program	Static	Call-Mark	D-Cache	Dynamic
P1	0.4	0.6	0.3	76.0
P2	1.9	1.8	0.7	101.0
P3	3.0	3.0	1.2	24,969.3

Discussions

- Analysis cost:
  static $\leq$ call-mark < d-cache << dynamic
  Slice size:
  static > call-mark > d-cache > dynamic
- Reasonable slice results with reasonable analysis time
- Promising approach to get effective program localization

Related Works

- Optimized Approaches for Dynamic Slicing(Agrawal & Horgan)
  - Still large execution overhead
- Hybrid Slicing(Guputa & Soffa)
  - Collect all traces between break points and proc. calls
  - Need to specify break points/ Trace can be huge
- Parametric Slicing(Field & Ramalingam)
  - Generalize static and dynamic slicing by symbolic execution with input data subset
  - Practicability and usefulness are unknown

On-Going Works

- Compiler-based lightweight semi-dynamic slicing environment
- Java program analysis
  - Bytecode analyses
  - Alias analysis for Java programs
  - GUI for alias information

課題

- プログラムスライスの研究動向調査
- プログラムスライスの原点の論文の要約
- プログラムスライスの応用に関する調査　田中、岡本
  - 応用分野
  - 商用システム

```
1:  prod :=1 ;
2:  sum :=0 ;
3:  x   :=1 ;
4:  while x=<10 do begin
5:      prod:= prod*x ;
6:      sum := sum + x ;
7:      x:= x +1 ;
        end;
8:  mean    := sum/10 ;
9:  writeln(prod,sum,mean);
```

```
1  get(low,high,step,A)
2  min:=A[low];
3  max:=A[low];
4  sum:=A[low];
5  i:=low+step;
6  While i=< high do
7          if max<A[i] then
8                  min:=A[i] ;
          end if
9          if min>A[i] then
10                 min:=A[i];
            end if
11          sum:=sum+A[i];
12           i:=i+step ;
      end loop;
13  put(min,max,sum);
```