Lecture 5. Program Dependencies and Program Slicing

Wei Le

2016.3

Agenda

- Motivation and applications
- Dependencies: control and data dependencies
- Dependency Graphs: control dependence graphs, data dependence graphs, program dependence graphs, system dependence graphs
- Program Slicing: control and data slice
  - Static slice algorithms
  - Dynamic slicing algorithms
- Program Chopping
- Program Dicing
- Path Slicing [1995:PLDI:Jhala]
- Thin Slicing [2007:PLDI:Sridharan:Bodik]
- Taint Analysis
- Impact Analysis

Applications and History

Examples of Applications:

- Optimize programs: Can I move this code out of the loop?
- Parallelize programs: Can I run the two pieces of code in parallel?
- Debugging: which sequence of statements lead to this error state?
- ...

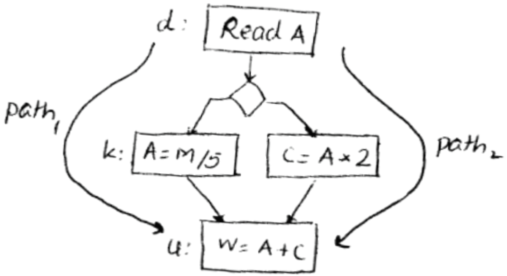
History

- Dependence graphs and compiler optimizations [1981:Kuck:POPL]
- Program graph and its use in optimizations [1987:Ferrante:TOPLAS]
- Interprocedural Slicing Using Dependence Graphs [1988:Horwitz:PLDI]

Preliminary: Reaching Definition

- **Reaching definition**:
  - **Definition d of variable v**: a statement that assigns a value to v;
  - **Use of variable v**: reference to value of v in an expression evaluation.
  - Definition d is **killed** along a path between two points if there exists an assignment to variable v along the path.
  - Definition d of variable v **reaches** a point p if there exists a path from immediately after d to p such that definition d is not killed along the path.

Preliminary: Reaching Definition



d reaches u along $path_2$ & d does not reach u along $path_1$

Since there exists a path from d to u along which d is not killed (i.e., $path_2$), d reaches u.

Preliminary: Reaching Definition

- Unambiguous Definition: X = .;
- Ambiguous Definition: *p = .; p may point to X
- For computing reaching definitions, typically we only consider "kills by unambiguous definitions"

Example:

x = ...
*p =
Does definition of X reach here? Yes

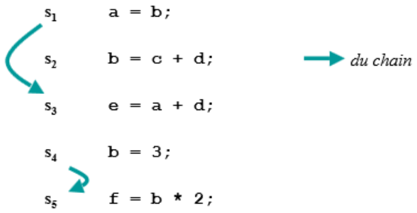
Data Dependence/Data Dependency

- Two statements are **data dependent**: the definition of a variable in a statement reaches the usage of the same variable at another statement.
- Data dependences are constraints on the order in which statement may be executed.
- Different types of data dependencies in the compiler field (typically IR or binary code), we say s_2 depends on s_1 [Kuck:1981:POPL]
  - s_1 writes memory that s_2 later reads (RAW)
  - s_1 reads memory that s_2 later writes (WAR)
  - s_1 writes memory that s_2 later writes (WAW)
  - s_1 reads memory that s_2 later reads (RAR) – input dependency
- In software engineering field, we typically compute dependencies of RAW, do source and IR level.

Represent Data Dependences

- **DU chains**: def-use chains (link each defs to uses)
  - pro: fast to get data dependencies
  - con: must be computed and updated, space overhead
- **SSA**: static single assignment (compiler)
  - each value produced in the program is represented using a variable
  - pro: allow analyses and transformations to be simpler and more efficient
  - con: may not be executable (requires extra translations to and from); space and time overhead
- **PDG**: program dependence graphs (software engineering) – data dependence graphs [1987:ferrante]: node is the statement, edge is the dependency relation
- Others

DU Chain: Example



SSA

Idea

- Each variable has only one static definition
- Makes it easier to reason about values instead of variables
- Similar to the notion of functional programming

Transformation to SSA

- Rename each definition
- Rename all uses reached by that assignment

Example

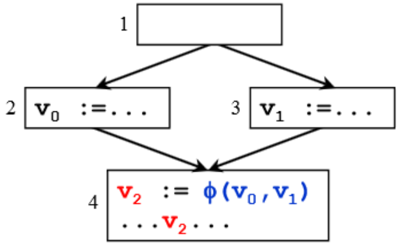
$$\begin{array}{rl} & \texttt{v := ...} \\ \texttt{...} & \texttt{:= ... v ...} \\ & \texttt{v := ...} \\ \texttt{...} & \texttt{:= ... v ...} \end{array} \Longrightarrow \begin{array}{rl} & \texttt{v}_0\ \texttt{:= ...} \\ \texttt{...} & \texttt{:= ... v}_0\ \texttt{...} \\ & \texttt{v}_1\ \texttt{:= ...} \\ \texttt{...} & \texttt{:= ... v}_1\ \texttt{...} \end{array}$$

SSA

Merging Definitions

– ϕ-functions merge multiple reaching definitions

Example



Transformation to SSA:

- place ϕ function
- rename variables

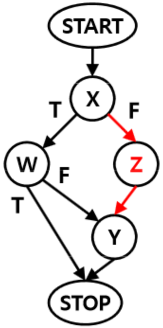
Data Dependece Graphs

It is acyclic unless there is a loop in the program

Control Dependence

Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X iff

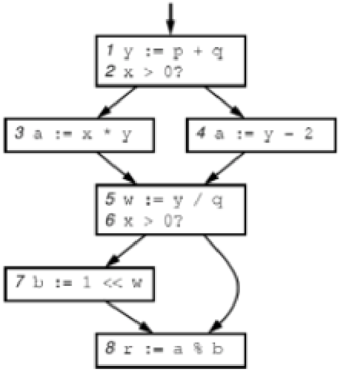
- There exists a directed path **P** from X to Y with any Z in P post-dominated by Y and
- X is not post-dominated by Y



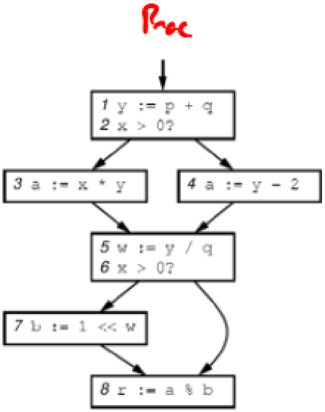
*If **Y** is control dependent on **X** then **X** must have **two exits**.*

- from condition 1: Y is not control dependent on any node between X and Y; that is, X is the first node Y is control dependent on
- the first statement control dependent on entry of the program

Control Dependence: Example

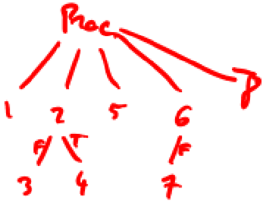


Control Dependence: Example



Control dependence relation

3 depends on 2
4 " " 2
7 " " 6



Control Dependence Edge

- $v_1 \rightarrow_c v_2$
- Case 1
  - v_1 : entry vertex
  - v_2 : component which is not nested within any loop
- Case 2
  - v_1 : control predicate
  - v_2 : component immediately nested within the loop or conditional whose predicate is represented by v_1
  - While loop : edge is labeled T (true)
  - Conditional statement : edge is labeled T (true) or F (false)

Control Dependence Graph

- **PDG**: program dependence graphs (software engineering) – control dependence graphs [1987:ferrante]

Program Dependence Graphs (PDG)

Node: statements
Edge: control and data dependence edges
Data Dependence + Control Dependence [1987:Ferrante:TOPLAS]

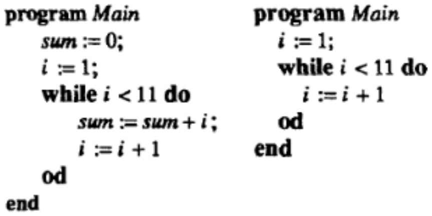
- **Data Dependence**
  - S2 depends on S1
    - Since variable A, the result of S1, is read in S2

```
S1: A = B * C
S2: D = A * E + 1
```

- **Control Dependence**
  - S2 depends on predicate A
    - Since the value of A determines whether S2 is executed

```
S1: if (A) then
S2:     B = C * D
    endif
```

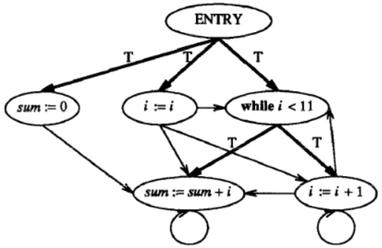
Program Dependence Graphs (PDG): Example

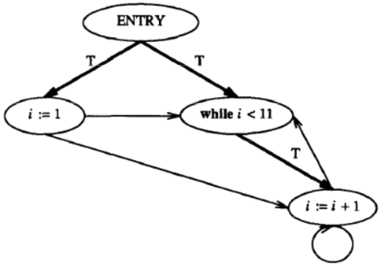


```
program Main
    sum := 0;
    i := 1;
    while i < 11 do
        sum := sum + i;
        i := i + 1
    od
end
```

```
program Main
    i := 1;
    while i < 11 do
        i := i + 1
    od
end
```

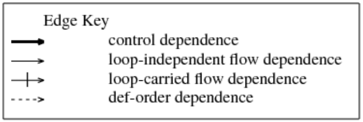
Program Dependence Graphs (PDG): Example



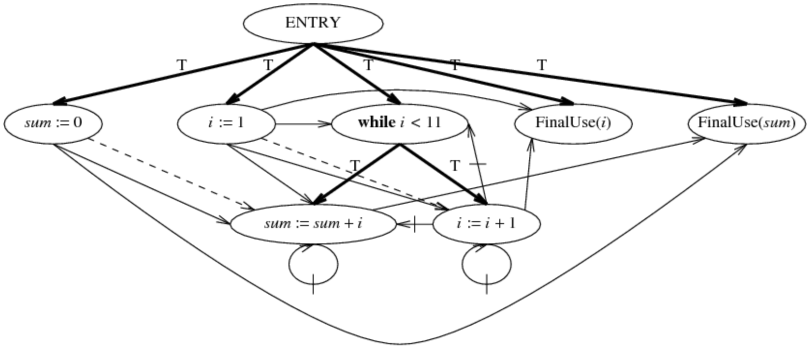


Program Dependence Graphs (PDG): Example

```
program Main
    sum := 0;
    i := 1;
    while i < 11 do
        sum := sum+i;
        i := i+1
    od
end(sum, i)
```



Edge Key
control dependence
loop-independent flow dependence
loop-carried flow dependence
def-order dependence



Construct PDG

Papers:

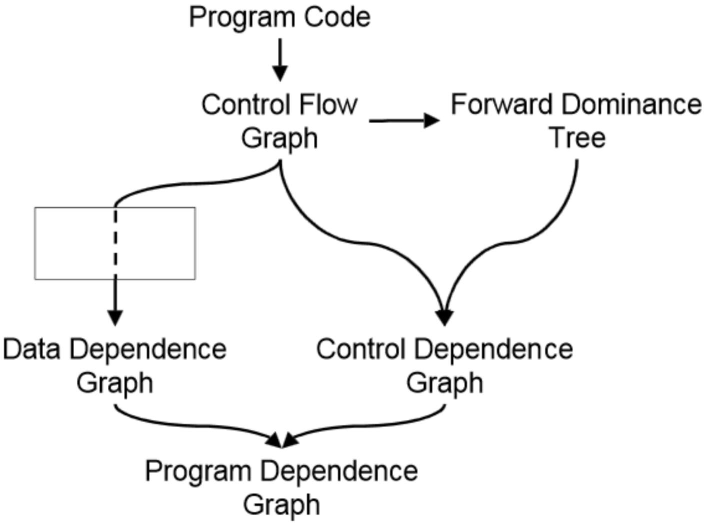
- The program dependence graph and its use in optimization, 1987, TOPLAS
- Efficiently computing static single assignment form and the control dependence graph, 1991, TOPLAS
- Interprocedural slicing using dependence graphs, 1988, PLDI

Tools: Frama-C (for C), Code Surfer (C/C++)

General approach:

- Data dependence: def-use relations (ambiguous definition vs unambiguous definition)
- Control dependence: control flow graphs, post dominators

Construct PDG



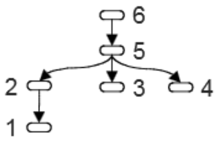
Forward Dominance (a.k.a. post dominance, inverse dominance)

- The *forward dominance tree, "FDT"*

```
Program Simple
1: read i
2: if ( i ==1)
3:    print "POS:"
   else
4:    i = 1
5: print i
6: end
```

- Vertices represent executable statements
- Arcs represent immediate forward dominance
- The root of the tree is the exit of the CFG

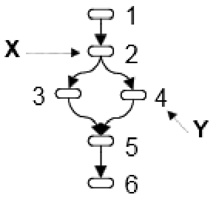
FDT



$$\begin{tikzcd} & 6 \arrow[d] & \\ 2 \arrow[d] & 5 \arrow[l] \arrow[d] \arrow[r] & 4 \\ 1 & 3 & \end{tikzcd}$$

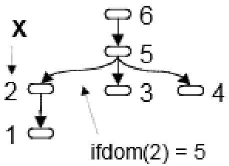
Construct PDG

- Y is *control dependent* on X ⇔ there is a *path in the CFG* from X to Y that doesn't contain the *immediate forward dominator* of X



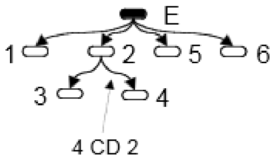
CFG

+



FDT

=



CDG

System Dependence Graphs (SDG)

(Optional Materials)

System Dependence Graphs (SDG)

SDG: an interprocedural dependence graph representation – a collection of method dependence graphs [1988:Horwitz, 1996:Larsen] (also used by Horwitz, Reps, Binkley)

- Program dependence graph: Represents the system's main program
- Procedure dependence graphs: Represent the system's auxiliary procedures
- Some additional edges
  - Edges that represent direct dependence between a call site and the called procedure
  - Edges that represent transitive dependences due to calls

System Dependence Graphs (SDG)

Five new vertices for SDG

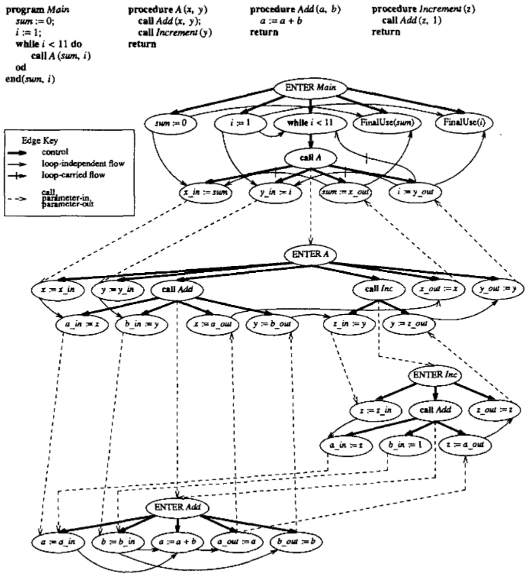
- Call-site vertex
- Actual-in:
  - Control dependent on call-site vertex
  - Copy values of actual parameters to call temporaries
- Actual-out:
  - Control dependent on call-site vertex
  - Copy from return temporaries
- Formal-in:
  - Control dependent on procedures entry vertex
  - Copy value of formal parameters from call temporaries
- Formal-out:
  - Control dependent on procedures entry vertex
  - Copy to return temporaries

System Dependence Graphs (SDG)

Three new edges for SDG

- Call edge
  - Call-site $\rightarrow$ Procedure-entry
  - From each call-site vertex to the corresponding procedure-entry vertex
- Parameter-in edge
  - Actual-in $\rightarrow$ Formal-in
  - From each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure
- Parameter-out edge
  - Formal-out $\rightarrow$ Actual-out
  - From each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site

System Dependence Graphs (SDG)



```
program Main
  sum := 0;
  i := 1;
  while i < 11 do
     call A (sum, i)
  od
end(sum, i)

procedure A (x, y)
  call Add (x, y);
  call Increment (y)
return

procedure Add (a, b)
  a := a + b
return

procedure Increment (z)
  call Add (z, 1)
return
```

Construct SDG

1. For each procedure of the system, construct its procedure dependence graph
2. For all call site, introduce a call edge from the call-site vertex to the corresponding procedure-entry vertex
3. For each actual-in vertex v at a call site, introduce a parameter-in edge from v to the corresponding formal-in vertex in the called procedure
4. For each actual-out vertex v at a call site, introduce a parameter-out edge to v from the corresponding formal-out vertex in the called procedure
5. Construct the linkage grammar corresponding to the system
6. Compute the subordinate characteristic graphs of the linkage grammar's nonterminals
7. At all call sites that call procedure P, introduce flow dependence edges corresponding to the edges in the subordinate characteristic graph for P

Program Slicing

- Definition
- Computing
- Application

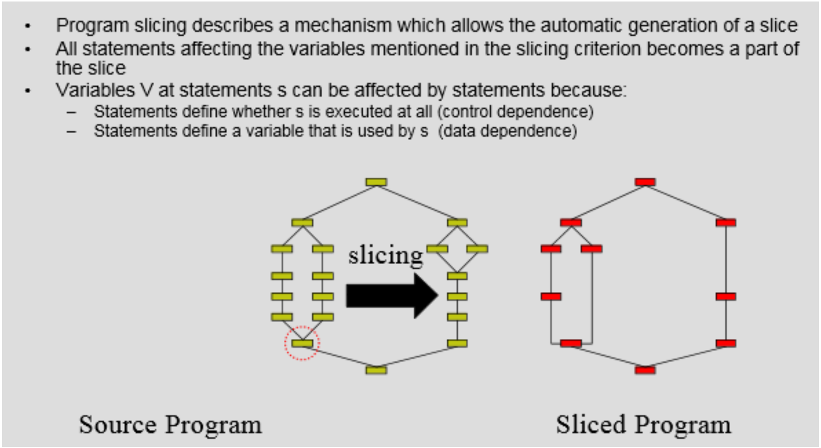
Origin of the Idea

Analysis technique introduced by Mark Weiser in his PHD thesis (1979)

- Idea derived when he was observing experienced programmers debugging a program
- Result: Every experienced programmer uses slicing to debug a program

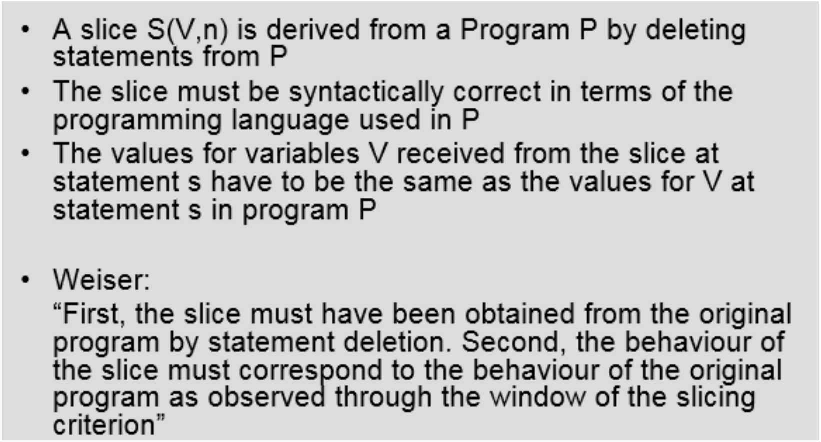
Program Slicing: Intuitive Understanding

Intuitively, the slice of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p



- Program slicing describes a mechanism which allows the automatic generation of a slice
- All statements affecting the variables mentioned in the slicing criterion becomes a part of the slice
- Variables V at statements s can be affected by statements because:
  - Statements define whether s is executed at all (control dependence)
  - Statements define a variable that is used by s (data dependence)

Program Slicing: Intuitive Understanding



- A slice S(V,n) is derived from a Program P by deleting statements from P
- The slice must be syntactically correct in terms of the programming language used in P
- The values for variables V received from the slice at statement s have to be the same as the values for V at statement s in program P

- Weiser:
  "First, the slice must have been obtained from the original program by statement deletion. Second, the behaviour of the slice must correspond to the behaviour of the original program as observed through the window of the slicing criterion"

Program Slicing: Intuitive Understanding

Produces programs, probably not executable [Ranjit Jhala: Path Slicing]

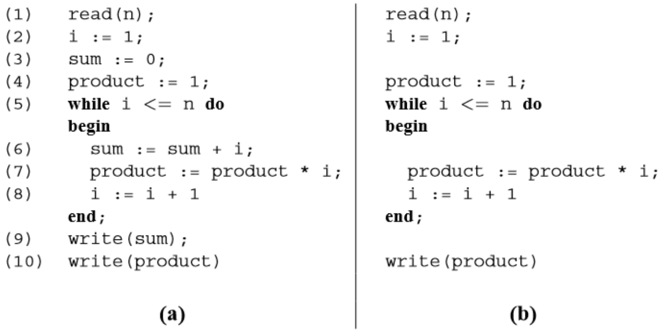
Program Slicing: Definition Roadmap

- backward and forward slice
- static and dynamic slice
- data and control slice
- slicing criterion
- path slicing
- thin slicing
- program dicing
- program chopping

Program Slicing: Definition

[1981:Weiser:ICSE] [1995:Tip]

- **(Backward) slice** of v at S is the set of statements involved in computing v's value at S
- A **slicing criterion** of a program P is a tuple $\langle i, V \rangle$, where i is a statement in P and V is a subset of the variables in P



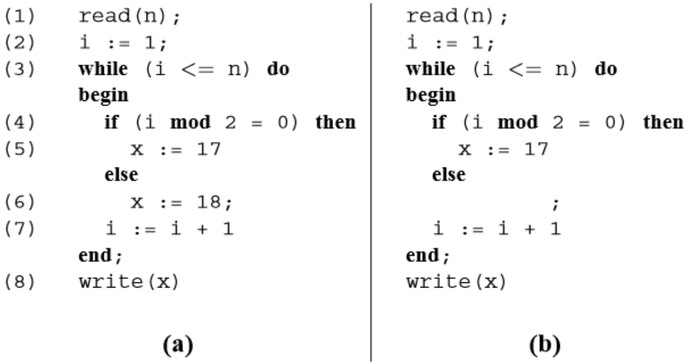
```
(1)   read(n);                       read(n);
(2)   i := 1;                        i := 1;
(3)   sum := 0;
(4)   product := 1;                  product := 1;
(5)   while i <= n do                while i <= n do
      begin                          begin
(6)     sum := sum + i;
(7)     product := product * i;        product := product * i;
(8)     i := i + 1                     i := i + 1
      end;                           end;
(9)   write(sum);
(10)  write(product)                 write(product)

              (a)                              (b)
```

(a) An example program. **(b)** A slice of the program w.r.t. criterion (10, `product`).

Program Slicing: Definition

- **static slice** and **dynamic slice**: **static slices** are computed without making assumptions regarding a program's input, whereas the computation of **dynamic slices** relies on a specific test case.



```
(1)   read(n);
(2)   i := 1;
(3)   while (i <= n) do
      begin
(4)     if (i mod 2 = 0) then
(5)       x := 17
        else
(6)       x := 18;
(7)     i := i + 1
      end;
(8)   write(x)

                (a)
```

```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else
             ;
  i := i + 1
end;
write(x)

          (b)
```

(a) Another example program. **(b)** Dynamic slice w.r.t. criterion ($n = 2, 8^1, x$)

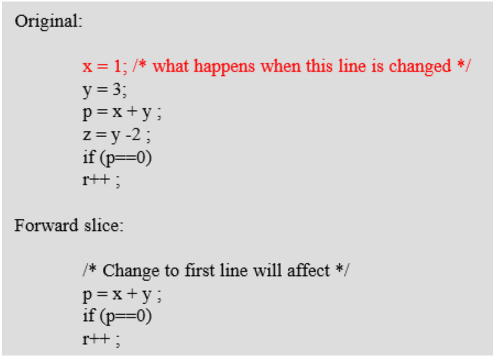
Here, the static slice is the entire program

Program Slicing: Definition

- A **data slice** is obtained by only taking (static or dynamic) data dependences into account; a **control slice** consists of the set of control predicates surrounding a language construct.
- The closure of all data and control slices w.r.t. an expression is the (static or dynamic) slice w.r.t. the set of variables used in the expression.

Program Slicing: Definition

Forward slice of a program with respect to a program point *p* and variable *x* consists of all statements and predicates of the program that might be affected by the value of *x* at point *p*


Original:

```
        x = 1; /* what happens when this line is changed */
        y = 3;
        p = x + y ;
        z = y -2 ;
        if (p==0)
        r++ ;
```

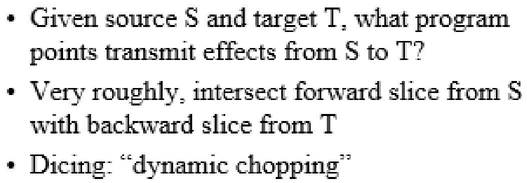
Forward slice:

```
        /* Change to first line will affect */
        p = x + y ;
        if (p==0)
        r++ ;
```

Advanced Definition: Dicing and Chopping

Combining two slices:

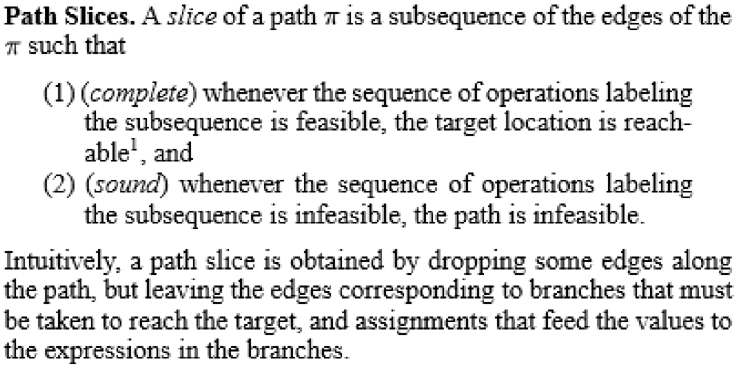
- **Program dicing** [lyle:weiser:1987] – used for fault localization: A program dice is a part of slice in which all stmts which are known to be correct have been removed.
- Program dicing: a method for combining the information of different slices. The basic idea is that, when a program computes a correct value for variable *x* and an incorrect value *y* for variable, the bug is likely to be found in statements that are in the slice w.r.t. *y*, but not in the slice w.r.t. *x*.
- **Program chopping**



- Given source S and target T, what program points transmit effects from S to T?
- Very roughly, intersect forward slice from S with backward slice from T
- Dicing: “dynamic chopping”

Advanced Definition: **Path Slicing**

- Path Slicing is an interesting technique to reduce the size of a counter example



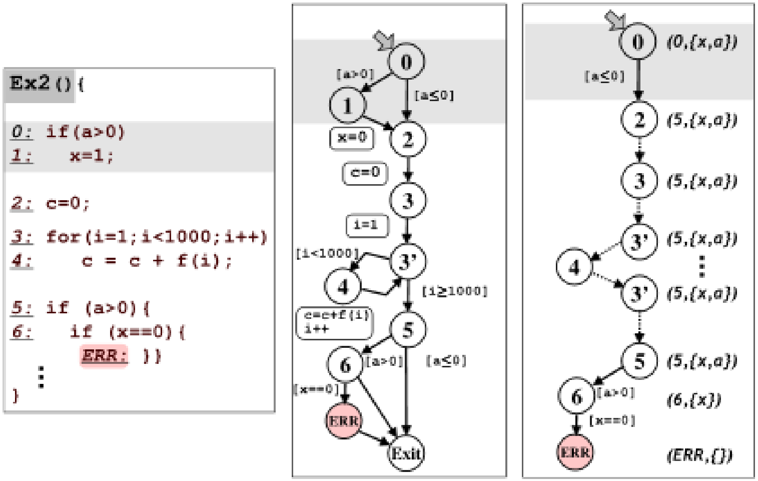
Path Slices. A *slice* of a path π is a subsequence of the edges of the π such that

(1) (*complete*) whenever the sequence of operations labeling the subsequence is feasible, the target location is reachable[1], and

(2) (*sound*) whenever the sequence of operations labeling the subsequence is infeasible, the path is infeasible.

Intuitively, a path slice is obtained by dropping some edges along the path, but leaving the edges corresponding to branches that must be taken to reach the target, and assignments that feed the values to the expressions in the branches.

Advanced Definition: Path Slicing



```
Ex2(){

0: if(a>0)
1:    x=1;

2: c=0;

3: for(i=1;i<1000;i++)
4:      c = c + f(i);

5: if  (a>0){
6:    if  (x==0){
        ERR:  }}
    ⋮
}
```

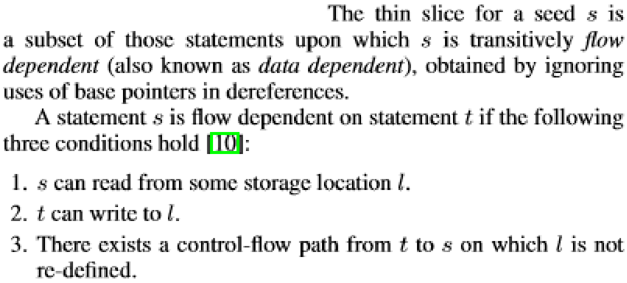
Figure 1. (A) Ex2 (B) CFA for Ex2 (C)Path, Slice

Advanced Definition: **Thin Slicing**

[2007:PLDI:Sridharan:Bodik]

- Motivation: Program slice is too large for human to consume – While a traditional slice includes all statements that may affect a point of interest, not all such statements appear equally relevant to a human
- A thin slice consists only of producer statements for the seed, i.e., those statements that help compute and copy a value to the seed. Statements that explain why producers affect the seed are excluded. For example, for a seed that reads a value from a container object, a thin slice includes statements that store the value into the container, but excludes statements that manipulate pointers to the container itself.
- 3.3 times fewer statements - debugging, 9.4 times fewer statements - program understanding

Advanced Definition: Thin Slicing

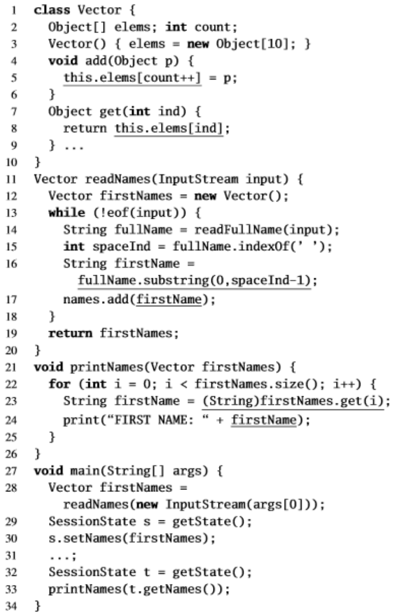


The thin slice for a seed s is a subset of those statements upon which s is transitively *flow dependent* (also known as *data dependent*), obtained by ignoring uses of base pointers in dereferences.

A statement s is flow dependent on statement t if the following three conditions hold [10]:

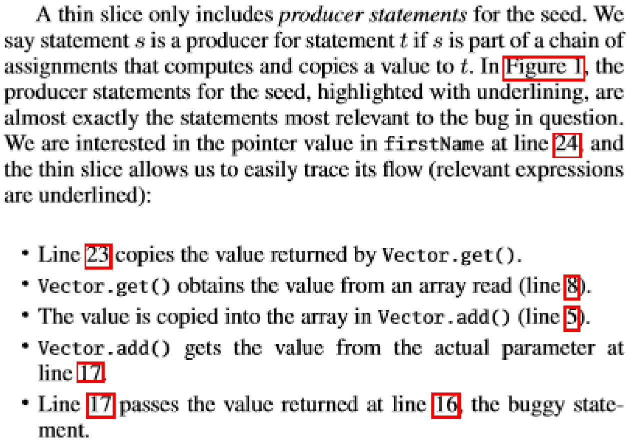
1. s can read from some storage location l.
2. t can write to l.
3. There exists a control-flow path from t to s on which l is not re-defined.

Advanced Definition: Thin Slicing



```
class Vector {
  Object[] elems; int count;
  Vector() { elems = new Object[10]; }
  void add(Object p) {
    this.elems[count++] = p;
  }
  Object get(int ind) {
    return this.elems[ind];
  } ...
}
Vector readNames(InputStream input) {
  Vector firstNames = new Vector();
  while (!eof(input)) {
    String fullName = readFullName(input);
    int spaceInd = fullName.indexOf(' ');
    String firstName =
      fullName.substring(0,spaceInd-1);
    names.add(firstName);
  }
  return firstNames;
}
void printNames(Vector firstNames) {
  for (int i = 0; i < firstNames.size(); i++) {
    String firstName = (String)firstNames.get(i);
    print("FIRST NAME: " + firstName);
  }
}
void main(String[] args) {
  Vector firstNames =
    readNames(new InputStream(args[0]));
  SessionState s = getState();
  s.setNames(firstNames);
  ...;
  SessionState t = getState();
  printNames(t.getNames());
}
```

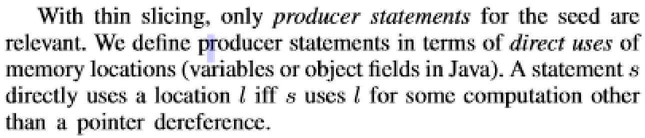
Advanced Definition: Thin Slicing



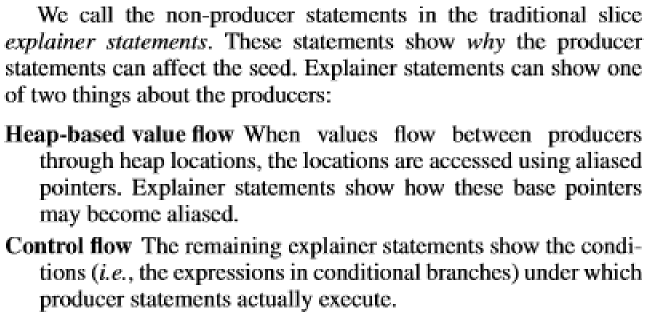
A thin slice only includes *producer statements* for the seed. We say statement s is a producer for statement t if s is part of a chain of assignments that computes and copies a value to t. In Figure 1, the producer statements for the seed, highlighted with underlining, are almost exactly the statements most relevant to the bug in question. We are interested in the pointer value in `firstName` at line 24, and the thin slice allows us to easily trace its flow (relevant expressions are underlined):

- Line 23 copies the value returned by `Vector.get()`.
- `Vector.get()` obtains the value from an array read (line 8).
- The value is copied into the array in `Vector.add()` (line 5).
- `Vector.add()` gets the value from the actual parameter at line 17.
- Line 17 passes the value returned at line 16, the buggy statement.

Advanced Definition: Thin Slicing



With thin slicing, only *producer statements* for the seed are relevant. We define producer statements in terms of *direct uses* of memory locations (variables or object fields in Java). A statement s directly uses a location l iff s uses l for some computation other than a pointer dereference.



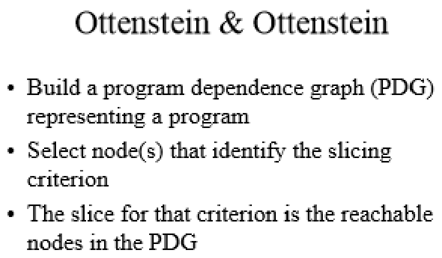
We call the non-producer statements in the traditional slice *explainer statements*. These statements show *why* the producer statements can affect the seed. Explainer statements can show one of two things about the producers:

Heap-based value flow When values flow between producers through heap locations, the locations are accessed using aliased pointers. Explainer statements show how these base pointers may become aliased.

Control flow The remaining explainer statements show the conditions (*i.e.*, the expressions in conditional branches) under which producer statements actually execute.

Compute Slice Using PDG: Highlevel Insight

Reachability on PDG



Ottenstein & Ottenstein

- Build a program dependence graph (PDG) representing a program
- Select node(s) that identify the slicing criterion
- The slice for that criterion is the reachable nodes in the PDG

Compute Slice Using PDG

- For vertex S of the Program Dependence Graphs G of a program P, the slice of G with respect to S is a graph (denoted as G/S) containing all vertices of which S has a transitive data or control dependence (i.e. all vertices that can reach S via data or control edges)
- The edges in the slice graph G/S are data dependence and control dependence of the original graph G that have source and targets in vertices in G/S

Compute Slice Using PDG

- Therefore the vertices of a slice G/S are:

  $V(G/S) = \{w \mid w \text{ in } V(G) \text{ and } w \rightarrow_{c,d}^{*} S\}$

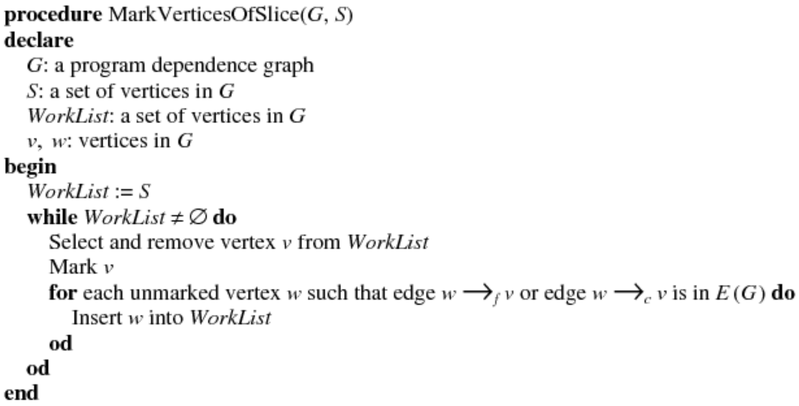
- We can extend the definition to a slice with respect to a set of vertices $S = \bigcup_i S_i$

  $V(G/S) = V(G/ \bigcup_i S_i) = \bigcup_i (V(G/S_i))$

- The edges in a slice are:

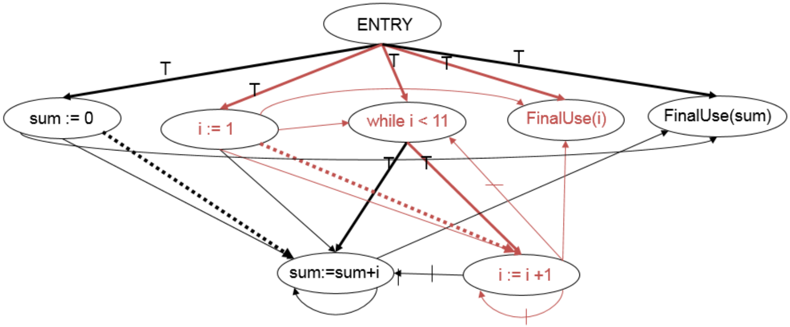
  $E(G/S) = \{u \rightarrow_d w \mid u \rightarrow w \text{ in } E(G) \text{ and } u, w \text{ in } V(G/S)\} \cup$
  $\{u \rightarrow_c w \mid u \rightarrow w \text{ in } E(G) \text{ and } u, w \text{ in } V(G/S)\}$

Compute Slice Using PDG



```
procedure MarkVerticesOfSlice($G$, $S$)
declare
   $G$: a program dependence graph
   $S$: a set of vertices in $G$
   $WorkList$: a set of vertices in $G$
   $v$, $w$: vertices in $G$
begin
   $WorkList := S$
   while $WorkList \neq \emptyset$ do
      Select and remove vertex $v$ from $WorkList$
      Mark $v$
      for each unmarked vertex $w$ such that edge $w \longrightarrow_f v$ or edge $w \longrightarrow_c v$ is in $E(G)$ do
         Insert $w$ into $WorkList$
      od
   od
end
```

Slicing on FinalUse(i)

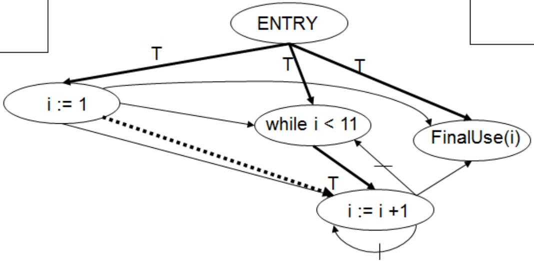


Slicing on FinalUse(i)

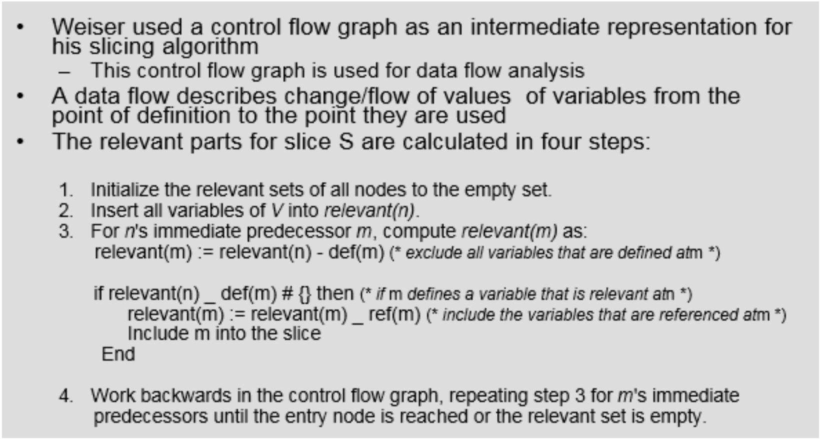
```
program Main
  sum := 0;
  i := 1;
  while i < 11 do
    sum := sum +i;
    i := i+1
  od
End(sum,i)
```

Slice on *FinalUse(i)*

```
program Main
  i := 1;
  while i < 11 do
     i := i+1;
  od
End(i)
```



Program Slicing as Dataflow Problem



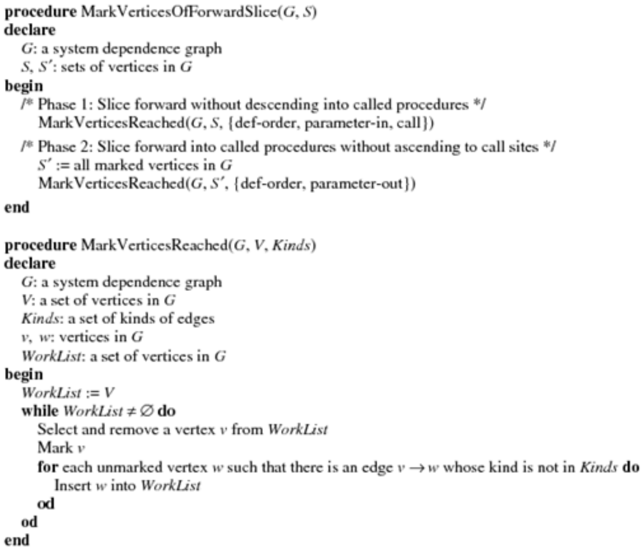
- Weiser used a control flow graph as an intermediate representation for his slicing algorithm
  - This control flow graph is used for data flow analysis
- A data flow describes change/flow of values of variables from the point of definition to the point they are used
- The relevant parts for slice S are calculated in four steps:

```
1. Initialize the relevant sets of all nodes to the empty set.
2. Insert all variables of V into relevant(n).
3. For n's immediate predecessor m, compute relevant(m) as:
   relevant(m) := relevant(n) - def(m) (* exclude all variables that are defined atm *)

   if relevant(n) _ def(m) # {} then (* if m defines a variable that is relevant atn *)
        relevant(m) := relevant(m) _ ref(m) (* include the variables that are referenced atm *)
        Include m into the slice
    End

4. Work backwards in the control flow graph, repeating step 3 for m's immediate
   predecessors until the entry node is reached or the relevant set is empty.
```

Compute Forward Slice



```
procedure MarkVerticesOfForwardSlice($G$, $S$)
declare
    $G$: a system dependence graph
    $S$, $S'$: sets of vertices in $G$
begin
    /* Phase 1: Slice forward without descending into called procedures */
        MarkVerticesReached($G$, $S$, {def-order, parameter-in, call})
    /* Phase 2: Slice forward into called procedures without ascending to call sites */
        $S'$ := all marked vertices in $G$
        MarkVerticesReached($G$, $S'$, {def-order, parameter-out})
end

procedure MarkVerticesReached($G$, $V$, $Kinds$)
declare
    $G$: a system dependence graph
    $V$: a set of vertices in $G$
    $Kinds$: a set of kinds of edges
    $v$, $w$: vertices in $G$
    $WorkList$: a set of vertices in $G$
begin
    $WorkList := V$
    while $WorkList \neq \emptyset$ do
        Select and remove a vertex $v$ from $WorkList$
        Mark $v$
        for each unmarked vertex $w$ such that there is an edge $v \rightarrow w$ whose kind is not in $Kinds$ do
            Insert $w$ into $WorkList$
        od
    od
end
```

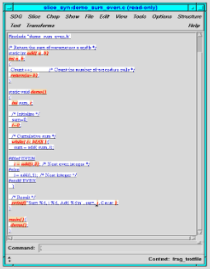
Compute Data and Control Slice

- Data slice: nodes that v transitively data dependent on – finding transitive data dependence on the data dependence graph
- Control slice: nodes that v transitive control dependent on – finding transitive control dependence on the control dependence graph

Tools

- Code Surfer (academic license)

  - Was used for C but no longer maintained
    - However commercial tool Codesurfer (http://www.grammatech.com/products/codesurfer/index.html) is derived from the Wisconsin program slicer
  - Developed and tested on Sun Sparc
  - Forward/backward-slicing, chopping, building and manipulating control flow graphs and program dependency graphs
  - Homepage:
  - http://www.cs.wisc.edu/wpis/slicing_tool/


slice_syn:demo_sum_even.c (read-only)
SDG Slice Chop Show File Edit View Tools Options Structure
Text Transforms Help
Command:
Context: frag_textfile

```
#include "demo_sum_even.h"

/* Return the sum of parameters a and b */
static int add( a, b)
int a, b;
{
 Count++;          /* Count the number of procedure calls */
 return(a+b);
}

static void demo()
{
 int sum, i;

 /* Initialize */
 sum=0;
 i=0;

 /* Cumulative sum */
 while( i< MAX ) {
   sum = add( sum, i);

#ifdef EVEN
   i = add(i,2); /* Next even integer */
#else
   i = add(i,1); /* Next integer */
#endif EVEN
 }

 /* Result */
 printf("Sum %d, i %d, Add %d\n", sum, i, Count );
}

main() {
 demo();
}
```

- Unravel

  - By John Lyle, Dolores Wallace, James Graham, Keith Gallagher, Joseph Poole, David Binkley
  - Runs on Sun Sparc
  - Slices programs in ANSI C
  - Has some restrictions (e.g. no goto statements)
    - Just backward slice at the moment
  - Performs work in reasonable time

  Homepage: http://hissa.nist.gov/unravel/

Dynamic Slicing

See Xiangyu Zhang's Slides

Program Slicing: Applications

Program understanding

- What is affected by what?

Program restructuring

- Isolate functionally distinct pieces of code

Program specialization and reuse

- Use slices to represent specialized pieces of code
- Only reuse relevant slices

Program differencing

- Compare slices to identify program changes

Program Slicing: Applications

Test coverage

- What new test cases would improve code coverage?
- What regression tests should be run after a change?

Model checking

- Reduce state space by removing irrelevant parts of the program

Automatic differentiation

- Activity analysis– what variables contribute to the derivative of a function?

Program Slicing: Applciations

Applications - backward slicing [1990:Horwitz:PLDI]

- Isolate individual computation threads within a program
- Understand complicated code, aid debugging
- Automatic parallelization
- Automatically integrating program variants [1987:Horwitz:POPL]
- Compute a safe approximation to the change in behavior: whether the modification on the program P interfere

Applications - forward slicing [1995:Tip]

- Show how a value computed at is being used subsequently, and can help the programmer ensure that establishes the invariants assumed by the later statements.
- Inspect the parts of a program that may be affected by a proposed modification, to check that there are no unforeseen effects on the program's behavior

Taint Analysis

Paper: Certification of programs for secure information flow

- security, binary code, a form of information flow – information flows from object x to object y, denoted $x \to y$, whenever information stored in x is transferred to, object y.
- forward slicing

- Identify **input dependent** variables at each program location
- Two kinds of dependencies:

Data dependencies

```
// x is tainted
   y = x ; z = y + 1 ; y = 3 ;
// z is tainted
```

Control dependencies

```
// x is tainted
   if (x > 0) y = 3 else y = 4 ;
// y is tainted
```

Applications of Taint Analysis (combined with forward symbolic execution)

Paper: All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)

- Unknown vulnerability detection
- Automatic input filter generation
- Malware analysis
- Test case generation

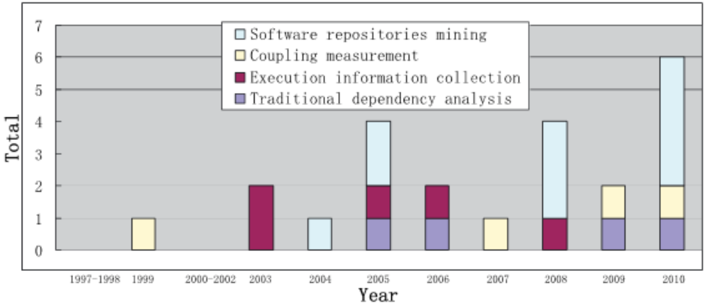
Static Taint and Dynamic Taint

See Futher Slides

Change Impact Analysis

Paper: a review of software change impact analysis
Paper: A survey of code-based change impact analysis techniques

- Software impact analysis identifies the effects of a software change request.



Distribution of the change impact analysis techniques from four perspectives.