C|ISPA Center for IT-Security, Privacy and Accountability

SAARLAND UNIVERSITY COMPUTER SCIENCE

Static Program Analysis

Program Dependence Graph

Winter Term 2014/15

Advanced Lecture (9 CP)

Christian Hammer

Scale

```
while(TRUE) {
  if ((p_ab[CTRL2] & 0x10)==0) {
    u = ((p_ab[PB] & 0x0f) << 8) + p_ab[PA];
    u_kg = u * kal_kg;}
  if ((p_cd[CTRL2] & 0x01) != 0) {
    for (idx=0;idx<7;idx++) {
      e_puf[idx] = p_cd[PA];
      if ((p_cd[CTRL2] & 0x10) != 0) {
        switch(e_puf[idx]) {
          case '+': kal_kg *= 1.1; break;
          case '-': kal_kg *= 0.9; break; } } }
      e_puf[idx] = '\0'; }
  printf("Artikel: %07.7s\n",e_puf);
  printf(" %6.2f kg ",u_kg);
}
```

Calibration factor can be changed!

CISPA Center for IT-Security, Privacy and Accountability

Program Slicing (Weiser '79)

- A program slice contains only those statements that potentially influence the execution of a given statement
- Irrelevant statements are removed (replaced by skip)
- Slicing criterion C = (n, V)
- Reduced program S is a slice if
  - it is a valid program
  - whenever P halts for a given input, S also halts for that input and computes the same values for the variables in V whenever the statement n is executed

Program Slicing

Original program

```
(1)  read(n);
(2)  i = 1;
(3)  sum = 0;
(4)  prod = 1;
(5)  while (i <= n) {
(6)    sum = sum + i;
(7)    prod = prod * i;
(8)    i++;
(9)  }
(10) write(sum);
(11) write(prod);
```

Slice for (10, {sum})

```
(1)  read(n);
(2)  i = 1;
(3)  sum = 0;
(4)
(5)  while (i <= n) {
(6)    sum = sum + i;
(7)
(8)    i++;
(9)  }
(10) write(sum);
```

Semantics of Slicing

- Slicing does not necessarily preserve program semantics
- May remove non-termination
  - Slice may no longer contain infinite loops
- Slice terminates while original program doesn't

- Non-standard semantics can be found that are preserved under program slicing [Danicic et al '07]

Slicing in the PDG

- Program representation called Program Dependence Graph
- Slicing becomes a reachability problem
- Linear to the number of statements (nodes)
- However, not necessarily executable slices
  (Extensions to make them executable again available)
- Slice: statements that (in-)directly affect slicing criterion
- PDG defined of CFG: nodes are the same
- Edges: Data and Control Dependence (vs. control flow)

Control Dependence

- One statement directly controls the execution of another
- In structured programs equivalent to “indentation level”

```
(1)  while (i <= n) {
(2)    sum = sum + i;
(3)    prod = prod * i;
(4)    i++;
(5)  }
(6)  write(sum);
```

- Statements in the while body are control dependent on the while predicate. The write is no longer dependent on the predicate.

Control Dependence Formally

- Standard definition in terms of post-dominance
- A node x in the CFG is post-dominated by node y if all paths from x to n_e pass through y
- A node y is control dependent on node x ($x \rightarrow_{cd} y$) if
  - $\exists$ path p from x to y in the CFG, such that y post-dominates every node in p (except for x), and
  - x is not post-dominated by y
- Extends intuitive notion to unstructured control flow
- Immediate post-dominator does not post-dominate any other post-dominator
- Induces a tree structure

CISPA Center for IT-Security, Privacy and Accountability

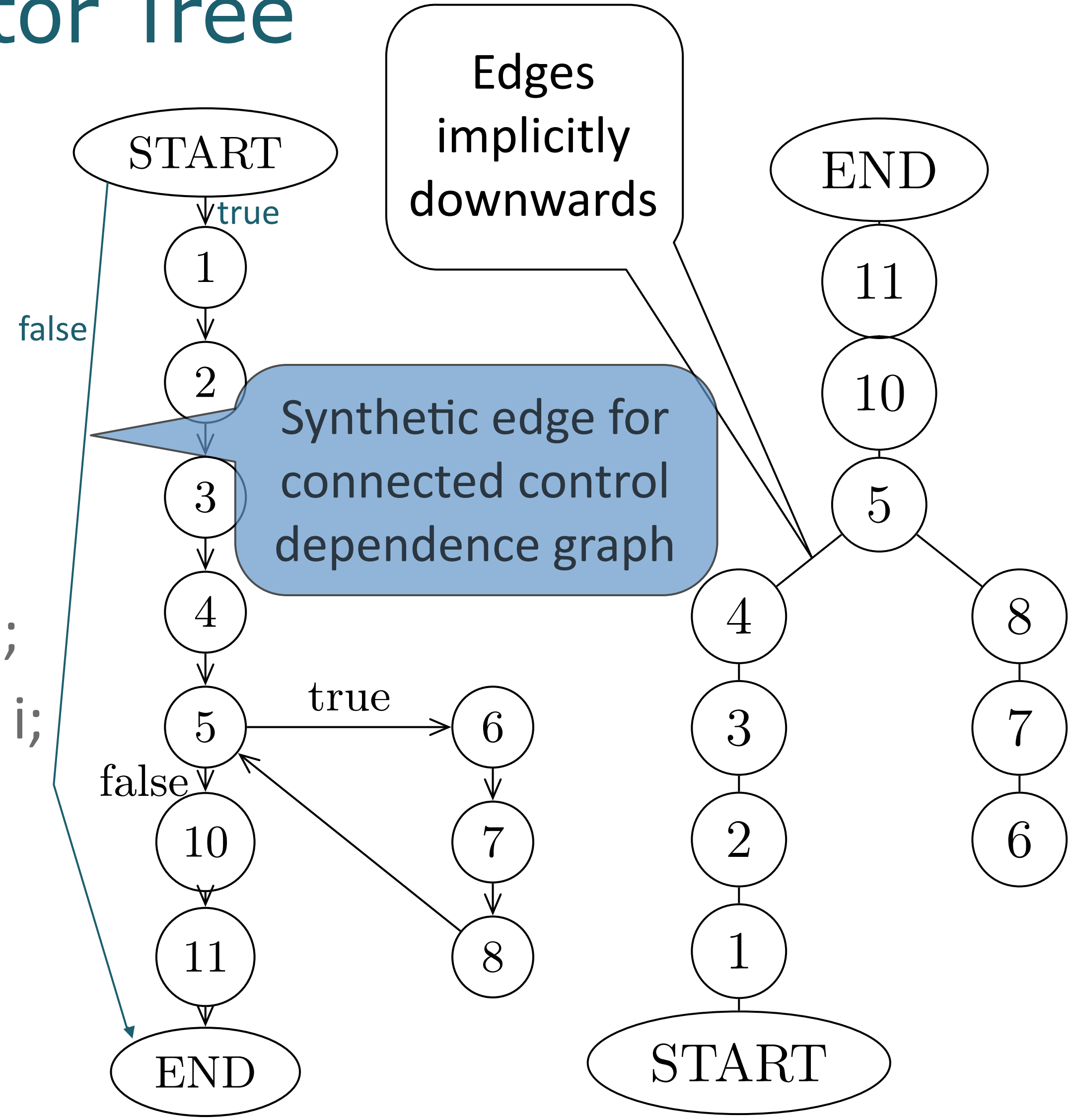
Computing Post-Dominators

- Several ways to compute post-dominators efficiently
- One way: using the following recursive equations
- maximal fixed point yields post-dominator relation
- Initialization: $pdom(end) = \{end\}$;
  $\forall n \in N \setminus \{end\} : pdom(n) = N$
- Iteration: $\forall n \in N \setminus \{end\}: pdom(n) = \{n\} \cup \left(\bigcap_{n \rightarrow_{cf} s} pdom(s)\right)$

- Alternative: Lengauer-Tarjan '79 algorithm for post-dominator tree (better worst-case complexity, but high constants)

Post-dominator Tree

(1) read(n);
(2) i = 1;
(3) sum = 0;
(4) prod = 1;
(5) while (i <= n) {
(6) sum = sum + i;
(7) prod = prod * i;
(8) i++;
(9) }
(10) write(sum);
(11) write(prod);

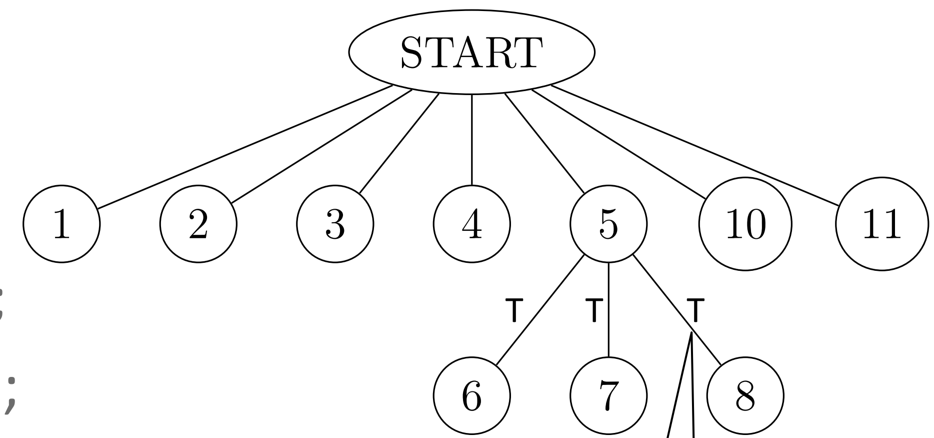


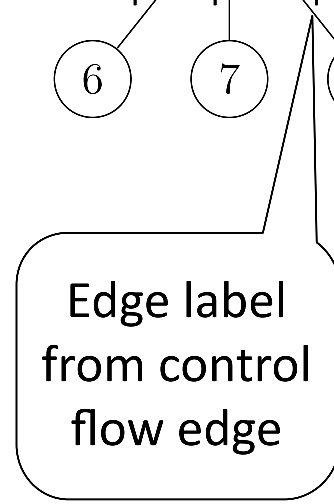
Computing Control Dependence

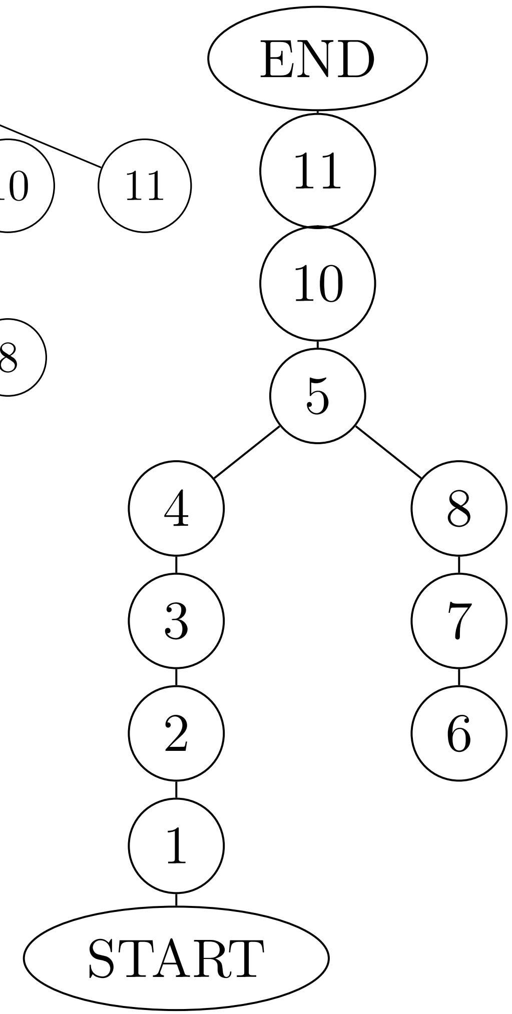
- Algorithm by Ferrante et al. ’87
- For every edge $x \rightarrow_{cf} y$ where x is not postdominated by y, one moves upwards from y in the post-dominator tree. Every node z visited before x’s parent is control dependent on x.
- The control dependence edge $x \rightarrow_{cd} z$ is labeled with $v(x, y)$, (label of the control flow edge)
- Synthetic edge is only used for computing post-dominators and control dependence, ignored for all other analyses

Control Dependence Graph

(1) read(n);
(2) i = 1;
(3) sum = 0;
(4) prod = 1;
(5) while (i <= n) {
(6) sum = sum + i;
(7) prod = prod * i;
(8) i++;
(9) }
(10) write(sum);
(11) write(prod);





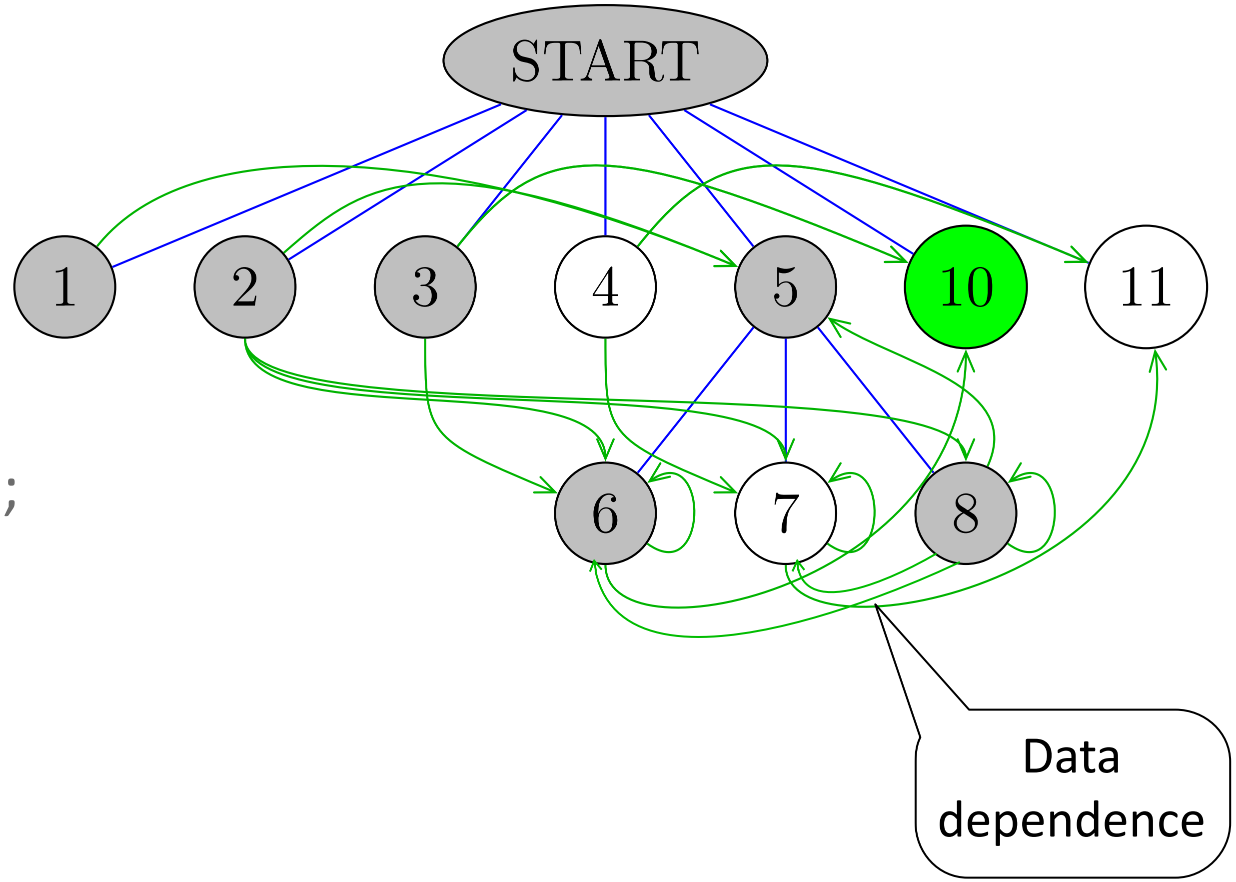


Data Dependence

- Known from optimizing compilers
- For slicing only “flow dependence” is relevant
- Called data dependence in the sequel
- $x \rightarrow_{dd} y$ means that a node x computes a value that may be used at node y in some feasible execution
- A node y is data dependent on node x ($x \rightarrow_{dd} y$) if
  - there exists a variable v with $v \in \mathrm{Def}(x)$ and $v \in \mathrm{Use}(y)$,
  - and ∃ path P in the CFG from x to y where the definition of v in x is not definitively killed (i.e. x is a reaching definition of y.)

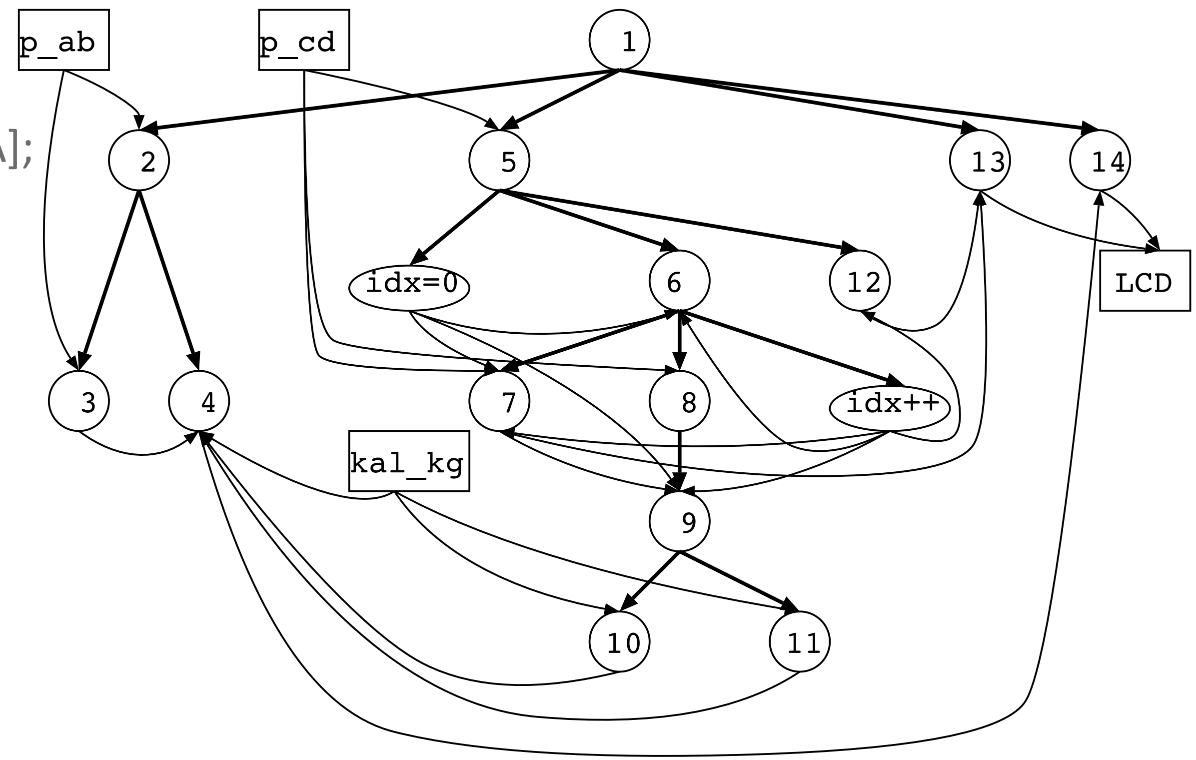
Program Dependence Graph

```
(1)  read(n);
(2)  i = 1;
(3)  sum = 0;
(4)  prod = 1;
(5)  while (i <= n) {
(6)    sum = sum + i;
(7)    prod = prod * i;
(8)    i++;
(9)  }
(10) write(sum);
(11) write(prod);
```



Scale

```
(1)  while(TRUE) {
(2)    if ((p_ab[CTRL2] & 0x10)==0) {
(3)      u = ((p_ab[PB] & 0x0f) << 8) + p_ab[PA];
(4)      u_kg = u * kal_kg;}
(5)    if ((p_cd[CTRL2] & 0x01) != 0) {
(6)      for (idx=0;idx<7;idx++) {
(7)        e_puf[idx] = p_cd[PA];
(8)        if ((p_cd[CTRL2] & 0x10) != 0) {
(9)          switch(e_puf[idx]) {
(10)           case '+': kal_kg *= 1.1; break;
(11)           case '-': kal_kg *= 0.9; break; } } }
(12)       e_puf[idx] = '\0'; }
(13) printf("Artikel: %07.7s\n",e_puf);
(14) printf(" %6.2f kg ",u_kg);
(15) }
```



Scale

```
(1)  while(TRUE) {
(2)    if ((p_ab[CTRL2] & 0x10)==0) {
(3)      u = ((p_ab[PB] & 0x0f) << 8) + p_ab[PA];
(4)      u_kg = u * kal_kg;}
(5)    if ((p_cd[CTRL2] & 0x01) != 0) {
(6)      for (idx=0;idx<7;idx++) {
(7)        e_puf[idx] = p_cd[PA];
(8)        if ((p_cd[CTRL2] & 0x10) != 0) {
(9)          switch(e_puf[idx]) {
(10)           case '+': kal_kg *= 1.1; break;
(11)           case '-': kal_kg *= 0.9; break; } } }
(12)       e_puf[idx] = '\0'; }
(13)   printf("Artikel: %07.7s\n",e_puf);
(14)   printf(" %6.2f kg ",u_kg);
(15) }
```