The Program Dependence Graph

Control flow and control dependences

CS502 Compilers

Program Dependence

- To understand the dependence relationship in a program, it is best to first examine it from a program execution trace
- A program execution trace is the complete sequence of instructions executed under a specific input, associated with the memory locations and registers visited by each instruction
  - An instruction may appear multiple times in the trace
  - We differentiate different instances of the same instruction by a sequence number

Data dependences and control dependences

- If instruction A defines a value used by instruction B (i.e. def(A) reaches use(B)), we say B has a flow dependence on A, or in short B has a dependence on A. Or, B is dependent on A.
- Such reaching definitions define a fundamental dependence relationship among program operations
  - It limits the freedom with which the operations may be reordered (or executed in parallel) for efficiency
  - It can guide software fault localization
- Flow dependences are transitive:
    - B is dependent on A, C is dependent on B ➔ C is dependent on A
    - We say there is a def/use chain from A to C

Other types of (data) dependences

- The reordering or parallel scheduling of operations may also be constrained by two other types of dependences involving data, i.e. involving memory locations
- **Anti dependences**
  - B is executed sometime after A
  - A reads from a memory location m, and B writes to m
  - We say B has an anti dependence on A, because
    - Reordering A and B may cause A to read a wrong value
- **Output dependences**
  - B is executed sometime after A
  - both A and B writes to memory location *m*
  - We say B has an output dependence on A, because
  - Reordering A and B may cause *m* to get a wrong value (that will be used by some other operations)

Control dependences

- In addition to data dependences listed above, a program operation B may also have a control dependence on a branching operation A
- If A is the ***nearest*** branch operation before B and A has multiple branch targets
- If changing the branch target for A may cause B ***not*** to be executed
- then we say B is control dependent on A, or B has a control dependence on A.
- Control dependences are transitive.

Dynamic Program Dependence Graph

- If we use a node to represent each instruction instance in the trace
- Use an edge to represent each dependence
- We obtain a dynamic dependence graph for the program under the specific input
- A dynamic dependence graph can be prohibitively large
- In practice, we maintain a moving snapshot for a specific purpose

Static program dependence graph

- For compilers and software engineering tools, it is important to build a static view of the dependences
  - Which takes all possible input into account
- In a (static) program dependence graph, each node represent all possible instances
- Each edge represents any possible edge between instances of two nodes under some possible input

Data dependences and control dependences

- If operation A defines a value that may be used by operation B (i.e. def(A) reaches use(B)), we say B has a flow dependence on A, or in short B has a dependence on A. Or, B is dependent on A.
- Such reaching definitions define a fundamental dependence relationship among program operations
  - It limits the freedom with which the compiler can reorder the operations (for efficiency or for information hiding)
    - Reordering includes parallel scheduling of operations
  - It can guide software fault localization
- Flow dependences are transitive:
    - B is dependent on A, C is dependent on B ➔ C is dependent on A
    - We say there is a def/use chain from A to C

Other types of (data) dependences

- The reordering or parallel scheduling of operations may also be constrained by two other types of dependences involving data, i.e. involving memory locations
- **Anti dependences**
  - *There is a control path from A to B*
  - A may read from a memory location m, and B may also write to m
  - We say B has an anti dependence on A, because
    - Reordering A and B may cause A to read a wrong value
- **Output dependences**
  - *There is a control path from A to B*
  - both A and B may write to memory location *m*
  - We say B has an output dependence on A, because
  - Reordering A and B may cause *m* to get a wrong value (that will be used by some other operations)

Control dependences

- In addition to data dependences listed above, a program operation B may also have a control dependence on a branching operation A
- Intuitively, if A is the ***nearest*** branch operation that has a control path leading to B and A has multiple branch targets
- If changing the branch target for A may cause B ***not*** to be executed
- then we say B is control dependent on A, or B has a control dependence on A.
- Control dependences are transitive.

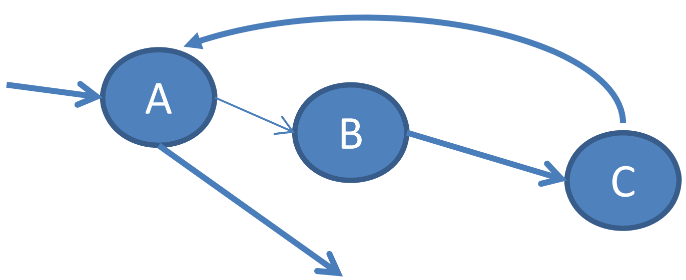
Control dependences

- In a (static) control flow graph, what "the nearest branch" means is yet to be formerly defined.
- How do we make a formal definition for (static) control dependences? We have the following "preliminary" definition:
- Node *a* is control-dependent on node *b* iff

  (1) *a* does not post dominate *b* (otherwise, no matter how control flows from *b* to exit, *a* will be executed)

  (2) A path exists from *b* to *a* such that every node on the path (excluding *b*) must be post dominated by *a*

- NOTE 1: Condition (2) is used for finding the nearest branch operation for *a*, i.e. *b*.
- NOTE 2: Condition (1) requires that a's execution is affected by b's branch target.
- NOTE 3: Unfortunately, Condition (1) prevents us from saying *a* being control dependent on *a* itself, in the case of *a* being a loop header:
  - In the following graph, according to the definition given above, we have *b* and *c* both control dependent on *a*, but *a* is not control dependent on itself.
  - However, when we view *the program trace*, we see that a later instance of *a* should be considered to be control dependent on the most recent instance of *a*
  - The static program dependence graph is intended to be the summary of all dependences collected from all possible dynamic dependence graphs. Hence the static dependence graph is supposed to have *a* being control dependent on *a* itself
  - We need to augment the definition as follows.



Control dependences: A formal definition

- Node *a* is control-dependent on node *b* iff

  (1) *a* does not ***strictly*** post dominate *b* (otherwise, no matter how control flows from *b* to *exit*, *a* will be executed)

  (2) A path exists from *b* to *a* such that every node on the path (excluding *b*) must be post dominated by *a*

  NOTE: The addition of word "strictly" will allow *a* to be control dependent on itself in the case of a loop, because *a* never strictly post dominate itself, thus satisfying (1)

Post Dominator Tree for Determining Control Dependences

- The formal definition given above is not suitable as an algorithmic base for determining control dependences
  - Because it would require enumeration of all pairs of nodes in the control graph, whose size is quadratic of the number of nodes
  - We want to find an order to visit each node in the graph *exactly once* do compute control dependences
  - The postdominator tree (pdom tree) is good for this purpose
- We show an example of pdom tree.

- An efficient algorithm for determining all control dependences given a control flow graph is based on the concept of *post dominance frontier (PDF*).
- First we will define the *dominance frontier, DF*.
- The PDF is simply the DF of the reversed flow graph
- Computing DF and PDF does not need to examine all paths, instead, it only examines the successors of a node of interest

Dominance Frontiers

- Definition (Dominance Frontiers):
  - The dominance frontier of x is the set of nodes that are not strictly dominated by x but have some predecessor being dominated by x.
  - Mathematically, the set is defined as

    DF(x) = { y | ($\exists\, z \in Predecessor\ (y)$ such that x dominates z) & $x\ does\ not\ strictly\ dominate\ y$ }
- The intuition: x "*almost*" dominates those nodes in DF(x)
- DF(x) contains the nearest *merging point* reached from x.

Post-dominance Frontiers

- The post-dominance frontier of *x* is the set of nodes that are not strictly post-dominated by *x* but have some successor being post-dominate d by *x*.
- Mathematically, we have

  PDF(x) = { y | ($\exists z \in Succ\ (y)$ such that x post-dominates z) and x $does\ not\ strictly\ post-dominate\ y$ }

- The intuition: x "*almost*" post-dominates those nodes in PDF(x)
- PDF(x) contains the nearest "*diverging points*" that lead to *x*

Theorem: *y* belongs to PDF(*x*) iff *x* is control dependent on *y*

- This theorem is based on the following **lemma**:
  - *x* post-dominates some successor of *y* iff a non-null path *p* exists from y to *x* such that *x* post-dominates every node in p, except y.
- This lemma tells us for Condition (2) there is no need to examine all paths from *y* to *x* in order to compute control dependences.
- It is sufficient to examine all successors of *y* and their post dominance relationship with *x*

Proof of the lemma

- ➔Suppose (y,z) is an edge and *x* post-dominates *z,* then pick any path from *z* to *exit,* it must contain *x. Every node in this path must be post-dominated by x. (*Otherwise, there would be an escape path from *z* to *exit* not containing *x.)*
- ← Suppose a non-null path *p* exists from *y* to *x* such that *x* post-dominates every node in *p*, except *y*, then take the first link (y,z) in this path. Obviously *x* post-dominates *z*.

- This lemma is from the paper titled
  - “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph” by Cytron, Ferrante , Rosen, Wegman, and Zadeck, TOPLAS 13(4), 1991
- The same paper adopts **a new definition for control dependences based on the PDF theorem**
- The original definition of control dependence is made in the following paper
  - The Program Dependence Graph and Its Use in Optimization, by Ferrante, Ottenstein, and Warren, TOPLAS 9(3), 1987

Computing DF(x)

- We first examine whether any edge (x,y) exists such that x does not strictly dominate y.
  - If so, by definition, y belongs to DF(x)
  - We denote the set of all such y by $DF_{local}(x)$
- Next, suppose x **immediately** dominates a set of nodes. For each of these nodes, z, we check each member y in DF(z).
  - If x does not **strictly** dominate y, then y belongs to DF(x)
  - We denote the set of all such y by $DF_{up}(x)$
  - Note: this definition deviates from the original paper but we believe it is more convenient

Mathematically

- Definition: DF_{local} (x) = {y ∈ Succ(x) | x does not strictly dominate y}
- Definition: $DF_{up}(x)$ = $\cup_{x\ idom\ z}\{\ y \mid y\ \in DF\ (z)$ and x $does\ not\ s$trictly dominate y}
- *We claim*
  - *DF(x)* = DF_{local} (x) ∪ $DF_{up}(x)$

Algorithm to compute DF(x)

- The above lemmas give rise to the algorithm for computing DF:
- For each x in the bottom-up traversal of the dominator tree do
  - DF(x) = ∅
  - Step 1: For each y in Succ(x) do /* local */

    if x is not *immediate dominator* of y then
    - DF(x) ← DF(x) ∪ $\{y\}$
  - Step 2: For each z that x immediately dominates, do
    - For each y ∈ $DF(z)$ do /* up */
      - If x is not *immediate dominator* of y then DF(x) ← DF(x) ∪ {y}

Proving the correctness of algorithm

- ***Lemma***: $Step\ 1\ of\ the\ algorithm\ computes\ DF_{local}(x)$

[proof] Let (x,y) be an edge, x ≠ y, and x dom y. Then x must be the immediate dominator of y.

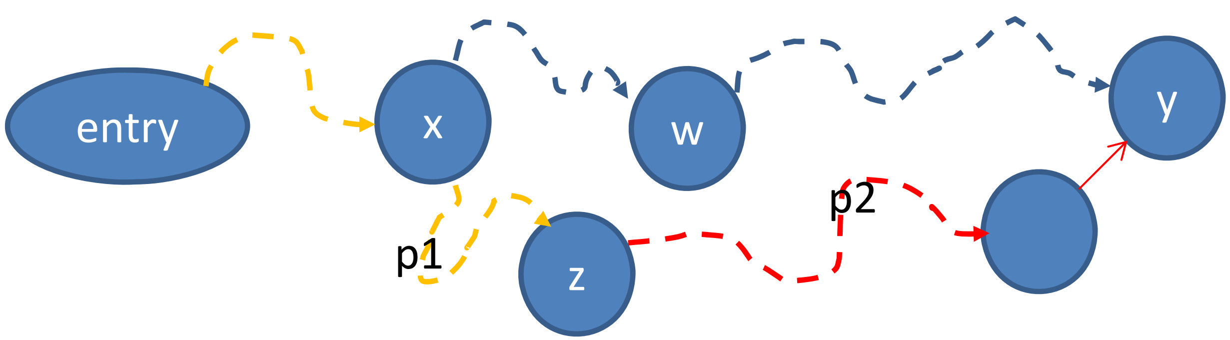
- ***[Implication]*** No need to search the dominator chain to establish that x does not dominate y in step 1
- ***Lemma:*** *Step 2 of the algorithm computes $DF_{up}(x)$*

[proof] → Every node y in $DF_{up}(x)$ will be added in Step 2, because if x does not strictly dominate y, then x does not idom y.

← See next page

Continue the proof

- ←Every node ***y*** added in Step 2 must belong to DF_{up}(x). Otherwise, suppose ***y*** is in DF(z) such that *x* idom *z*, *x* does not idom y but x strictly dominates ***y*** (i.e. ***y*** does not belong to DFup(x)). There must be another node **w** such that x idom **w,** and **w** strictly dominates **y.**
- Now this is impossible, because since ***y*** is in DF(z), *z* must dominate a predecessor of ***y, which is not w. Hence there is a path, p_2, from z to y containing no w.***
- Since x idom z, **w** cannot dominate z. Hence there exist a path p_1 from entry to z that does not contain **w.** Connecting p_1 and p_2, we find a path from entry to ***y*** containing no **w**, which contracts **"**w **strictly dominates** y"**.**



Proof that $DF(x) \subseteq DF_{local}(x) \cup DF_{up}(x)$

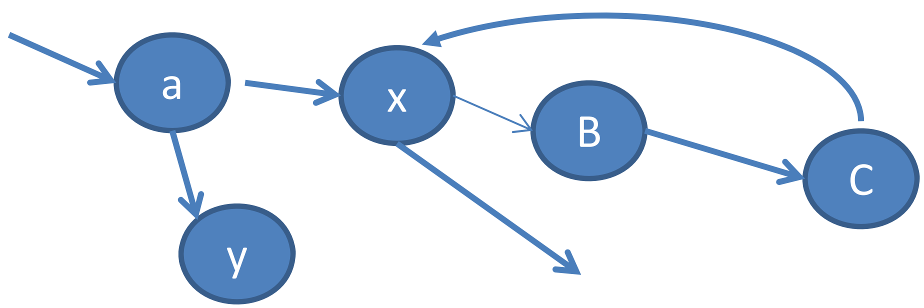
- If $y \in DF(x)$, then by definition *x* dominates a predecessor *z* of y. We want to prove that y is in either $DF_{local}(x)$ or $DFup(x)$.
- Case 1: $y = x = z$. (x,x) is an edge and $x \in DF_{local}(x)$.
- Case 2: $y \neq x = z$, then (x,y) and x does not strictly dominate y. Hence, $y \in DF_{local}(x)$
- Case 3: $y = x \neq z$. Since x=y strictly dominates z, z cannot strictly dominate x=y (why?). Hence $y \in DF_{local}(z)$.
- Case 4: $y \neq x \neq z$, then z does not strictly dominate y (otherwise x strictly dominates y, a contradiction). Hence, $y \in DF_{local}(z)$.
- In both Cases 3 and 4, there is a dominance chain between x and z in the dom tree. The nodes in this chain will appear in every path from x to z. No node in this chain strictly dominates y (otherwise we have x strictly dominates y). Since $y \in DF_{local}(z)$, through this chain, we have $y \in DF_{up}(idom(z))$. $y \in DF_{up}(idom(idom(z))$, …, $y \in DF_{up}(x)$.

Algorithm to compute PDF(x)

- A direct translation of the algorithm for computing DF(x) yields an algorithm for PDF(x)
- For each x in the bottom-up traversal of the post-dominator tree do
  - PDF(x) = Ø
  - Step 1: For each y in Predecessor(x) do /* local */

    if x is not *immediate post-dominator* of y then
    - PDF(x) ← PDF(x) ∪ $\{y\}$
  - Step 2: For each z that x immediately post-dominates, do
    - For each y ∈ $PDF(z)$ do /* up */
      - If x is not *immediate post-dominator* of y then PDF(x) ← PDF(x) ∪ {y}

- It is possible for a node x to be control dependent on more than one branch nodes
- A simple example is the loop header, x, being control dependent on itself and on another branch, a, that is “nearest” to the loop header, as in the following graph



- Examples that have no loops also exist, in which a node is control dependent on more than one node. We will present one in this lecture.

The program dependence graph

- In the program dependence graph, each node represents an operation in the program, and each edge represents a dependence.
- Often the kind of dependence (flow, anti-, output, control) is marked on the edge
- One can choose the granularity of the PDG, depending on the pur-
- pose of the analysis:

  • It can be fine-grained, such that a node represents an ALU operation, a load or a store, a branch instruction

  • It can also be coarse-grained, such that a node represents a function invocation.

  • It can also be of a granule in between:

  -- program statements

  -- compound statements, e.g. loops

  -- basic blocks