Data Dependence

**Definition**
- Data dependences are constraints on the order in which statements may be executed.

We say statement $s_2$ depends on $s_1$:
- **Flow (true) dependence**: $s_1$ writes memory that $s_2$ later reads (RAW)
- **Anti-dependence**: $s_1$ reads memory that $s_2$ later writes (WAR)
- **Output dependences**: $s_1$ writes memory that $s_2$ later writes (WAW)
- **Input dependences**: $s_1$ reads memory that $s_2$ later reads (RAR)

**True dependences**
- Flow dependences represent actual flow of data.

**False dependences**
- Anti- and output dependences reflect reuse of memory, not actual data flow; can often be eliminated.

Example

```
s_1: a = b;
s_2: b = c + d;
s_3: e = a + d;
s_4: b = 3;
s_5: f = b * 2;
```

DU Chains

**Definition**
- Du chains link each def to its uses.

Example

```
s_1: a = b;
s_2: b = c + d;
s_3: e = a + d;
s_4: b = 3;
s_5: f = b * 2;
```

Representing Data Dependences

**Implicitly**
- Using variable defs and uses
- Pros: simple
- Cons: hides data dependence (analyses must find this info)

**Def-use chains (du chains)**
- Link each def to its uses
- Pros: explicit; therefore fast
- Cons: must be computed and updated, space consuming

**Alternate representations**
- e.g., Static single assignment form (SSA), dependence flow graphs (DFG), value dependence graphs (VDG)
UD Chains

**Definition**
- UD chains link each use to its def's

**Example**

```
s_1  a = b;
s_2  b = c + d;
s_3  e = a + d;
s_4  b = 3;
s_5  f = b * 2;
```

Role of Alternate Program Representations

**Advantage**
- Allow analyses and transformations to be simpler & more efficient/effective

**Disadvantage**
- May not be "executable" (requires extra translations to and from)
- May be expensive (in terms of time or space)

**Process**

- Original Code (RTL) → SSA Code1 → SSA Code2 → SSA Code3 → Optimized Code (RTL)

Static Single Assignment (SSA) Form

**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**

```
v := ...
... := ... v ...
v := ...
... := ... v ...
v := ...
... := ... v ...
```

**What do we do when there's control flow?**

SSA and Control Flow

**Problem**
- A use may be reached by several definitions
**SSA and Control Flow (cont)**

Merging Definitions

- $\phi$-functions merge multiple reaching definitions

**Example**

![Diagram of SSA and Control Flow](image)

**Another Example**

```
1 v := 1
2 v := v+1
```

**SSA vs. ud/du Chains**

SSA form is more constrained

Advantages of SSA

- More compact
- Some analyses become simpler when each use has only one def
- Value merging is explicit
- Easier to update and manipulate?

Furthermore

- Eliminates false dependences (simplifying context)

```
for (i=0; i<n; i++)
    A[i] = i;
for (i=0; i<n; i++)
    print(foo(i));
```

**SSA vs. ud/du Chains (cont)**

Worst case du-chains?

```
switch (c1) {
  case 1:  x = 1; break;
  case 2:  x = 2; break;
  case 3:  x = 3; break;
}
x_4 = \phi(x_1, x_2, x_3)
switch (c2) {
  case 1:  y_1 = x; break;
  case 2:  y_2 = x; break;
  case 3:  y_3 = x; break;
  case 4:  y_4 = x; break;
}
```

$m$ defs and $n$ uses leads to $m \times n$ du-chains.
**Transformation to SSA Form**

Two steps
- Insert $\phi$-functions
- Rename variables

**Where Do We Place $\phi$-Functions?**

Basic Rule
- If two distinct (non-null) paths $x \rightarrow z$ and $y \rightarrow z$ converge at node $z$, and nodes $x$ and $y$ contain definitions of variable $v$, then a $\phi$-function for $v$ is inserted at $z$

$$
\begin{align*}
&x : v_1 := \ldots \\
&y : v_2 := \ldots \\
&z : v_3 := \phi(v_1, v_2) \\
&\ldots \ldots
\end{align*}
$$

**Approaches to Placing $\phi$-Functions**

**Minimal**
- As few as possible subject to the basic rule

**Briggs-Minimal**
- Same as minimal, except $v$ must be live across some edge of the CFG

**Pruned**
- Same as minimal, except dead $\phi$-functions are not inserted

What’s the difference between Briggs Minimal and Pruned SSA?

**Briggs Minimal vs. Pruned**

Briggs Minimal will add a $\phi$ function because $v$ is live across the blue edge, but Pruned SSA will not because the $\phi$ function is dead

Neither Briggs Minimal nor Pruned SSA will place a $\phi$ function in this case because $v$ is not live across any CFG edge

Why would we ever use Briggs Minimal instead of Pruned SSA?
Concepts

Data dependences
- Three kinds of data dependences
- du-chains
Alternate representations
SSA form
Conversion to SSA form
- $\phi$-function placement

Next Time

Assignments
- HW1 due
Lecture
- SSA continued