Lecture 11:

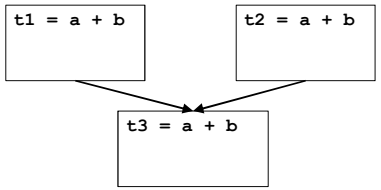
Partial Redundancy Elimination

- Global code motion optimization
  - Remove partially redundant expressions
  - Loop invariant code motion
  - Can be extended to do Strength Reduction
- No loop analysis needed
- Bidirectional flow problem

[ALSU 9.5-9.5.2]

Redundancy

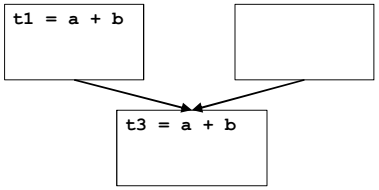
- A Common Subexpression is a Redundant Computation



- Occurrence of expression E at P is **redundant** if E is available there:
  - E is evaluated along every path to P, with no operands redefined since.
- Redundant expression can be eliminated

Partial Redundancy

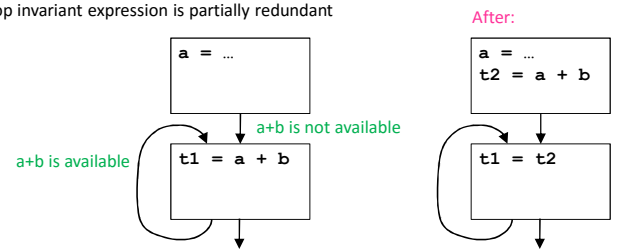
- Partially Redundant Computation



- Occurrence of expression E at P is **partially redundant** if E is partially available there:
  - E is evaluated along at least one path to P, with no operands redefined since.
- Partially redundant expression can be eliminated if we can insert computations to make it fully redundant.

Loop Invariants are Partial Redundancies

- Loop invariant expression is partially redundant



- As before, partially redundant computation can be eliminated if we insert computations to make it fully redundant.
- Remaining copies can be eliminated through copy propagation or more complex analysis of partially redundant assignments.

Partial Redundancy Elimination

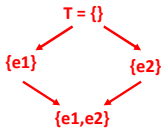
- **The Method:**
  1. Insert Computations to make partially redundant expression(s) fully redundant.
  2. Eliminate redundant expression(s).
- **Issues [Outline of Lecture]:**
  1. What expression occurrences are candidates for elimination?
  2. Where can we safely insert computations?
  3. Where do we want to insert them?
- For this lecture, we assume one expression of interest, a+b.
  - In practice, with some restrictions, can do many expressions in parallel.

Which Occurrences Might Be Eliminated?

- In CSE,
  - E is **available** at P if it is previously evaluated along **every** path to P, with no subsequent redefinitions of operands.
  - If so, we can eliminate computation at P.
- In PRE,
  - E is **partially available** at P if it is previously evaluated along **at least one** path to P, with no subsequent redefinitions of operands.
  - If so, we might be able to eliminate computation at P, if we can insert computations to make it fully redundant.
- Occurrences of E where E is partially available are candidates for elimination.

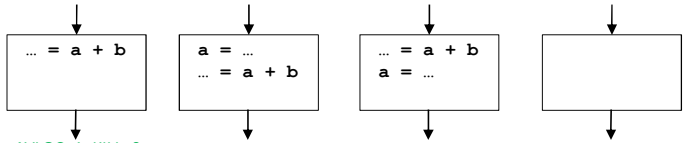
Finding Partially Available Expressions

- **Forward flow problem**
  - **Lattice = { 0, 1 }^n, meet is union (∪) (elementwise max)**
    Top = 0^n (= not PAVAIL), entry = 0^n, init = 0^n
    - PAVOUT[b] = (PAVIN[b] – KILL[b]) ∪ AVLOC[b]
    - PAVIN[b] = $\begin{cases} 0^n & b = entry \\ \bigcup_{p \in preds(b)} \text{PAVOUT}[p] & otherwise \end{cases}$



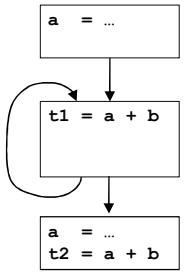
Meet Operator: Union

- **For a block,**
  - Expression is **locally available (AVLOC)** if computed & downwards exposed.
  - Expression is killed (**KILL**) if any assignments to operands.



Partial Availability Example

- **For expression a+b**



PAVOUT[entry] = 0

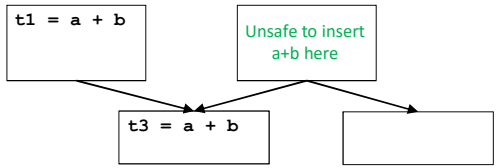
a = ...	KILL = 1 AVLOC = 0	PAVIN = 0 PAVOUT = 0
t1 = a + b	KILL = 0 AVLOC = 1	PAVIN = 0 1 (2nd iteration) PAVOUT = 0 1
a = ... t2 = a + b	KILL = 1 AVLOC = 1	PAVIN = 1 PAVOUT = 0 1

PAVOUT[b] = (PAVIN[b] – KILL[b]) ∪ AVLOC[b]

- **Occurrence in loop is partially redundant (PAVIN=1)**

Where Can We Insert Computations?

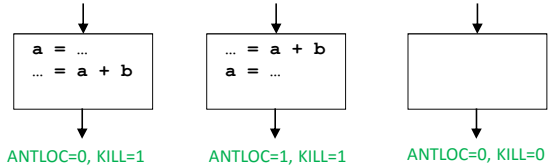
- **Safety: never introduce a new expression along any path.**



  - Insertion could introduce exception, change program behavior.
  - If we can add a new basic block, can insert safely in most cases.
  - Solution: insert expression only where it is anticipated.
- **Performance: never increase the # of computations on any path.**
  - Under simple model, guarantees program won't get worse.
  - Reality: might increase register lifetimes, add copies, lose.

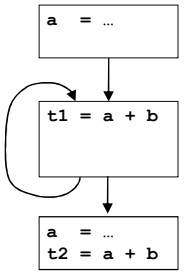
Finding Anticipated Expressions

- **Backward flow problem**
  - **Lattice = { 0, 1 }, meet is intersection (∩), top = 1 (ANT), exit = 0, init = 1**
    - ANTIN[i] = ANTLOC[i] ∪ (ANTOUT[i] - KILL[i])
    - ANTOUT[i] = $\begin{cases} 0 & i = exit \\ \bigcap_{s \in succ(i)} \text{ANTIN}[s] & otherwise \end{cases}$
- **For a block,**
  - Expression **locally anticipated (ANTLOC)** if defined & upwards exposed



Anticipation Example

- **For expression a+b** **ANTIN[i] = ANTLOC[i] ∪ (ANTOUT[i] - KILL[i])**



KILL = 1 ANTIN = 1 0
ANTLOC = 0 ANTOUT = 1

KILL = 0 ANTIN = 1
ANTLOC = 1 ANTOUT = 0

KILL = 1 ANTIN = 1 0
ANTLOC = 0 ANTOUT = 0

ANTIN[exit] = 0

- **Expression is anticipated at end of first block (ANTOUT=1)**
- **Computation may be safely inserted there**

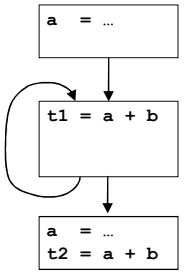
Where Do We Want to Insert Computations?

- **Morel-Renvoise and variants: "Placement Possible"**
  - Dataflow analysis shows where to insert:
    - PPIN = "Placement possible at entry of block or before."
    - PPOUT = "Placement possible at exit of block or before."
  - Insert at earliest place where PPIN = 1.
  - Only place at end of blocks,
    - PPIN really means "Placement possible or not necessary in each predecessor block."
  - Don't need to insert where expression is already available.
    - INSERT[i] = PPOUT[i] ∩ (¬PPIN[i] ∪ KILL[i]) ∩ ¬AVOUT[i]

      Can put it here — Can't move it back any further — Not already available
  - Remove (upwards-exposed) computations where PPIN=1.
    - DELETE[i] = PPIN[i] ∩ ANTLOC[i]

      Moved earlier — Used locally here

Where Do We Want to Insert? Example



PPIN = 0
PPOUT = 1

Insert here

PPIN = 1
PPOUT = 0

Delete here

PPIN = 0
PPOUT = 0

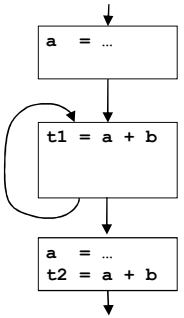
Formulating the Problem

- **PPOUT: we want to place at output of this block only if**
  - we want to place at entry of all successors (correctness & performance)
- **PPIN: we want to place at input of this block only if (all of):**
  - we have a local computation to place, or a placement at the end of this block which we can move up
  - we want to move computation to output of all predecessors where expression is not already available (don't insert at input)
  - we can gain something by placing it here (PAVIN)
- **Forward or Backward?**
  - **BOTH!**
- **Problem is *bidirectional*, but lattice {0, 1} is finite, so**
  - as long as transfer functions are monotone, it converges.

Computing "Placement Possible"

- **PPOUT: we want to place at output of this block only if**
  - we want to place at entry of all successors
    - PPOUT[i] = $\bigcap_{s \in succ(i)}$ PPIN[s]
- **PPIN: we want to place at start of this block only if (all of):**
  - we have a local computation to place, or a placement at the end of this block which we can move up
  - we want to move computation to output of all predecessors where expression is not already available (don't insert at input)
  - we gain something by moving it up (PAVIN heuristic)
    - PPIN[i] = (ANTLOC[i] $\cup$ (PPOUT[i] – KILL[i])) $\cap \bigcap_{p \in preds(i)}$ (PPOUT[p] $\cup$ AVOUT[p]) $\cap$ PAVIN[i]

"Placement Possible" Example 1



PPOUT[entry] = 0

KILL = 1 PAVIN = 0
AVLOC = 0 PAVOUT = 0
ANTLOC = 0 AVOUT = 0
PPIN = 0
PPOUT = 1

Insert here

KILL = 0 PAVIN = 1
AVLOC = 1 PAVOUT = 1
ANTLOC = 1 AVOUT = 1
PPIN = 1
PPOUT = 0

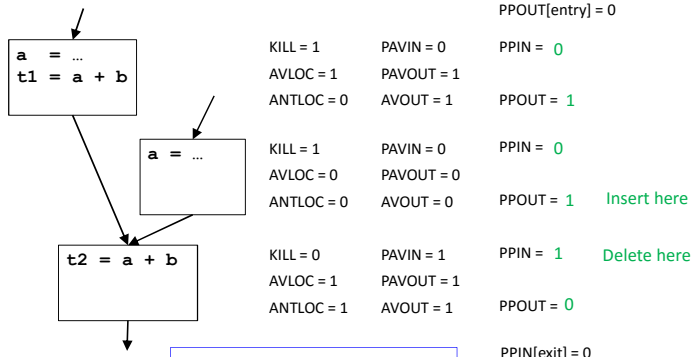
Delete here

KILL = 1 PAVIN = 1
AVLOC = 1 PAVOUT = 1
ANTLOC = 0 AVOUT = 1
PPIN = 0
PPOUT = 0

PPIN[exit] = 0

(ANTLOC[i] $\cup$ (PPOUT[i] – KILL[i])) $\cap \bigcap_{p \in preds(i)}$ (PPOUT[p] $\cup$ AVOUT[p]) $\cap$ PAVIN[i]

"Placement Possible" Example 2



PPOUT[entry] = 0

KILL = 1, AVLOC = 1, ANTLOC = 0 | PAVIN = 0, PAVOUT = 1, AVOUT = 1 | PPIN = 0, PPOUT = 1

KILL = 1, AVLOC = 0, ANTLOC = 0 | PAVIN = 0, PAVOUT = 0, AVOUT = 0 | PPIN = 0, PPOUT = 1 — Insert here

KILL = 0, AVLOC = 1, ANTLOC = 1 | PAVIN = 1, PAVOUT = 1, AVOUT = 1 | PPIN = 1, PPOUT = 0 — Delete here

PPIN[exit] = 0

$$(ANTLOC[i] \cup (PPOUT[i] - KILL[i])) \cap \bigcap_{p \in preds(i)} (PPOUT[p] \cup AVOUT[p]) \cap PAVIN[i]$$

"Placement Possible" Correctness

- **Convergence of analysis: transfer functions are monotone**
- **Safety: Insert only if anticipated**

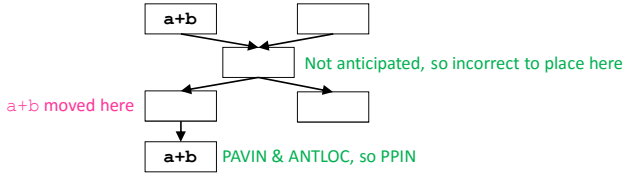
$$PPIN[i] \subseteq (PPOUT[i] - KILL[i]) \cup ANTLOC[i]$$

$$PPOUT[i] = \bigcap_{s \in succ(i)} PPIN[s]$$

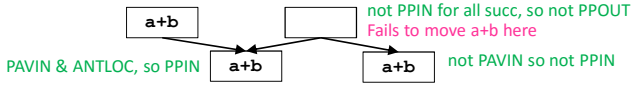
  - INSERT $\subseteq$ PPOUT $\subseteq$ ANTOUT, so insertion is safe
- **Performance: never increase the # of computations on any path**
  - DELETE = PPIN $\cap$ ANTLOC
  - On every path from an INSERT, there is a DELETE
  - The number of computations on a path does not increase

Morel-Renvoise Limitations

- **Movement usefulness tied to PAVIN heuristic**
  - Makes some useless moves, might increase register lifetimes:



  - Doesn't find some eliminations:



- **Bidirectional data flow difficult to compute**

Friday's Class

- Lazy Code Motion [ALSU 9.5.3-9.5.6]