

## Lecture 10

# Partial Redundancy Elimination

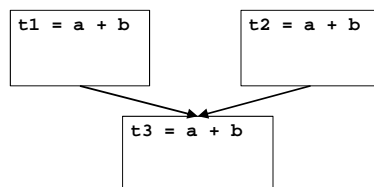
- Global code motion optimization
  - Remove partially redundant expressions
  - Loop invariant code motion
  - Can be extended to do Strength Reduction
- No loop analysis needed
- Bidirectional flow problem

## References

1. E. Morel and C. Rennoise, "Global Optimization by Suppression of Partial Redundancies," *CACM* 22 (2), Feb. 1979, pp. 96-103.
2. Knoop, Rüthing, Steffen, "Lazy Code Motion," PLDI 92.
3. F. Chow, "A Portable Machine-Independent Global Optimizer--Design and Measurements. Stanford CSL memo 83-254.
4. Dhamdhere, Rosen, Zadeck, "How to Analyze Large Programs Efficiently and Informatively," PLDI 92.
5. K. Drechsler, M. Stadel, "A Solution to a Problem with Morel and Rennoise's 'Global Optimization by Suppression of Partial Redundancies,'" *ACM TOPLAS* 10 (4), Oct. 1988, pp. 635-640.
6. D. Dhamdhere, "Practical Adaptation of the Global Optimization Algorithm of Morel and Rennoise," *ACM TOPLAS* 13 (2), April 1991.
7. D. Dhamdhere, "A Fast Algorithm for Code Movement Optimisation," *SIGPLAN* Not. 23 (10), 1988, pp. 172-180.
8. S. Joshi, D. Dhamdhere, "A composite hoisting --- strength reduction transformation for global program optimisation," *International Journal of Computer Mathematics*, 11 (1982), pp. 21-41, 111-126.

## Redundancy

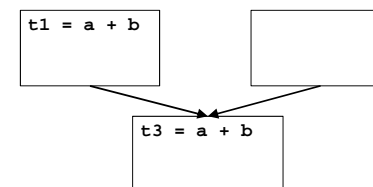
- A Common Subexpression is a Redundant Computation



- 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

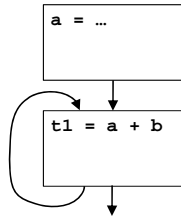
- Partially Redundant Computation



- 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

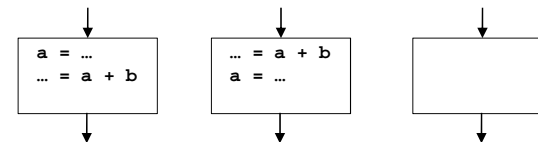
- The Method:**
  - Insert Computations to make partially redundant expression(s) fully redundant.
  - Eliminate redundant expression(s).
- Issues [Outline of Lecture]:**
  - What expression occurrences are candidates for elimination?
  - Where can we safely insert computations?
  - Where do we want 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.

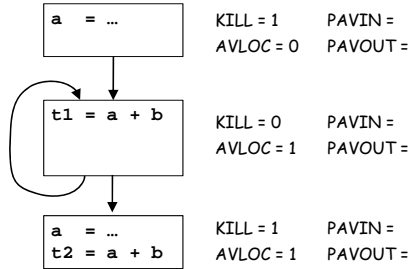
## Finding Partially Available Expressions

- Forward flow problem**
  - Lattice** =  $\{0, 1\}$ , **meet** is union ( $\cup$ ), **Top** = 0 (not PAVAIL), **entry** = 0
  - $PAVOUT[i] = (PAVIN[i] - KILL[i]) \cup AVLOC[i]$
  - $PAVIN[i] = \begin{cases} 0 & i = \text{entry} \\ \cup_{p \in \text{preds}(i)} PAVOUT[p] & \text{otherwise} \end{cases}$
- For a block,**
  - Expression is **locally available (AVLOC)** if downwards exposed.
  - Expression is killed (**KILL**) if any assignments to operands.



### Partial Availability Example

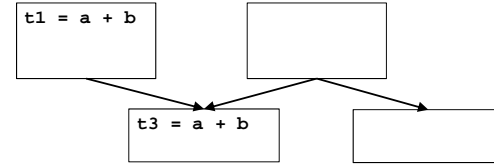
- For expression  $a+b$ .



- Occurrence in loop is partially redundant.

### 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

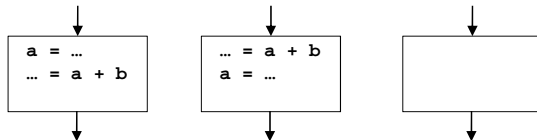
- Backward flow problem

- Lattice = { 0, 1 }, meet is intersect ( $\cap$ ), top = 1 (ANT), exit = 0

- $ANTIN[i] = ANTLOC[i] \cup (ANTOUT[i] - KILL[i])$
- $ANTOUT[i] = \begin{cases} 0 & i = exit \\ \cap_{s \in succ(i)} ANTIN[s] & otherwise \end{cases}$

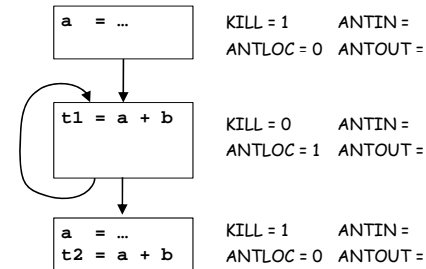
- For a block,

- Expression **locally anticipated (ANTLOC)** if upwards exposed.



### Anticipation Example

- For expression  $a+b$ .

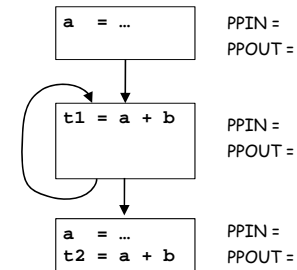


- Expression is anticipated at end of first block.
- Computation may be safely inserted there.

## 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 PP = 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.
    - $INSERT[i] = PPOUT[i] \cap (\neg PPIN[i] \cup KILL[i]) \cap \neg AVOUT[i]$
  - Remove (upwards-exposed) computations where  $PPIN=1$ .
    - $DELETE[i] = PPIN[i] \cap ANTLOC[i]$

## Where Do We Want to Insert? Example



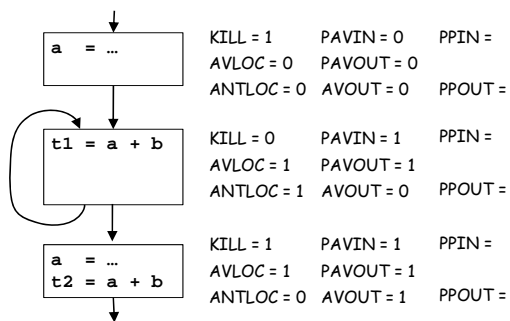
## Formulating the Problem

- **PPOUT: we want to place at output of this block only if**
  - we want to place at entry of all successors
- **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.

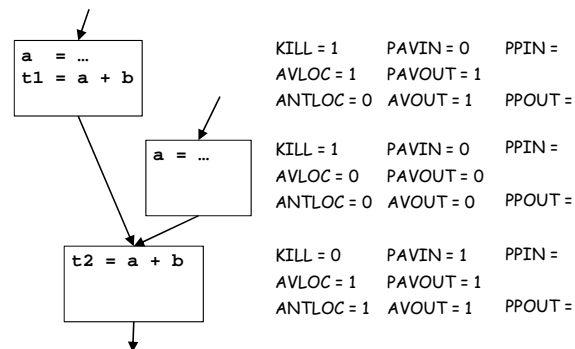
## Computing "Placement Possible"

- **PPOUT: we want to place at output of this block only if**
  - we want to place at entry of all successors
    - $PPOUT[i] = \begin{cases} 0 & i = exit \\ \bigcap_{s \in succ(i)} PPIN[s] & otherwise \end{cases}$
- **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)
    - $PPIN[i] = \begin{cases} 0 & i = exit \\ ((ANTLOC[i] \cup (PPOUT[i] - KILL[i])) \cap \bigcap_{p \in preds(i)} (PPOUT[p] \cup AVOUT[p])) \cap PAVIN[i] & otherwise \end{cases}$

### "Placement Possible" Example 1



### "Placement Possible" Example 2



### "Placement Possible" Correctness

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

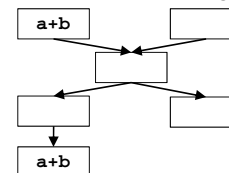
$$PPIN[i] \subseteq (PPOUT[i] - KILL[i]) \cup ANTLOC[i]$$

$$PPOUT[i] = \begin{cases} 0 & i = exit \\ \bigcap_{s \in succ(i)} PPIN[s] & otherwise \end{cases}$$

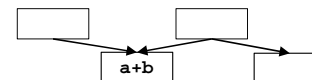
- INSERT  $\subseteq$  PPOUT  $\subseteq$  ANTOUT, so insertion is safe.
- **Performance**: never increase the # of computations on any path
  - DELETE = PPIN  $\cap$  ANTLOC
  - On every path from an INSERT, there is a DELETE.
  - The number of computations on a path does not increase.

### Morel-Rennoise 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.**

## Related Work

- **Don't need heuristic**
  - Dhamdhere, Drechsler-Stadel, Knoop, et.al.
  - use restricted flow graph or allow edge placements.
- **Data flow can be separated into unidirectional passes**
  - Dhamdhere, Knoop, et. al.
- **Improvement still tied to accuracy of computational model**
  - Assumes performance depends only on the number of computations along any path.
  - Ignores resource constraint issues: register allocation, etc.
  - Knoop, et.al. give "earliest" and "latest" placement algorithms which begin to address this.
- **Further issues:**
  - more than one expression at once, strength reduction, redundant assignments, redundant stores.