**15-745**

**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



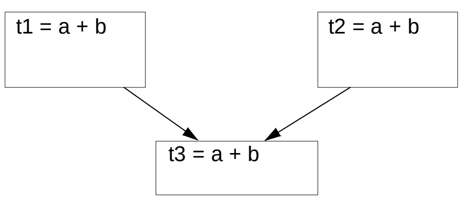
## References

1. E. Morel and C. Renvoise, “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 Renvoise’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 Renvoise,” 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

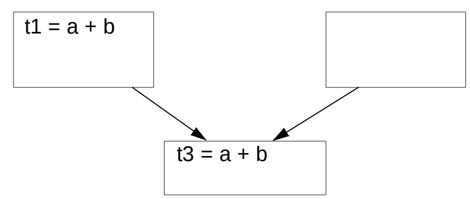



- Occurrence of expression E at P is **redundant** if E is available there:
  - E is evaluated along every path to P, with no operands redefined since.
- Redundant expression can be eliminated



## Partial Redundancy

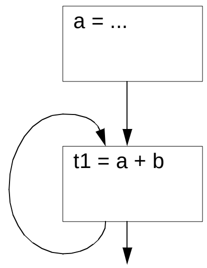
- Partially Redundant Computation




- 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:**
  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.



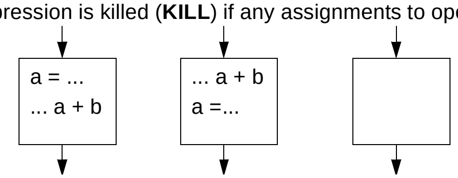
# Finding Partially Available Expressions

- **Forward flow problem**
- **Lattice = { 0, 1 }, meet is union ( ∪ ), top = 0 (not PAVAIL),entry = 0**

$PAVOUT[i] =$

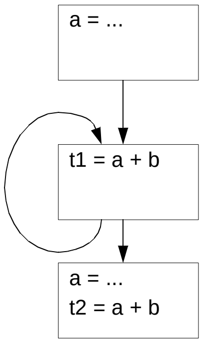
$PAVIN[i] =$

- **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.**



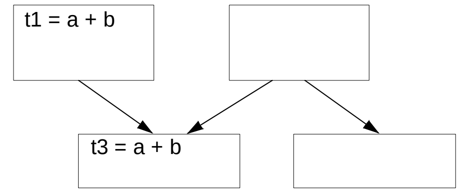

| | |
|---|---|
| KILL = 1 | PAVIN= |
| AVLOC=0 | PAVOUT= |
| KILL = 0 | PAVIN= |
| AVLOC=1 | PAVOUT= |
| KILL = 1 | PAVIN= |
| AVLOC=1 | PAVOUT= |

- **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 number of computations on any path.**
  - Under simple model, guarantees program won't get worse.
  - Reality: might increase register lifetimes, add copies, lose.



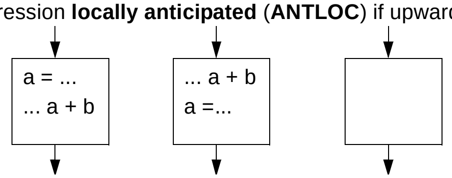
# Finding Anticipated Expressions

- **Backward flow problem**
- **Lattice = { 0, 1 }, meet is intersect ( $\cap$ ), top = 1 (PANT), exit = 0**

$PANTIN[i] =$

$PANTOUT[i] =$

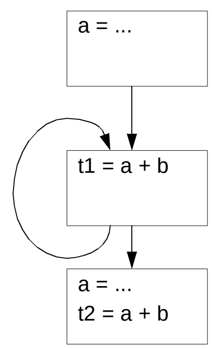
- **For a block,**
  - Expression **locally anticipated** (**ANTLOC**) if upwards exposed.






# Anticipation Example

- **For expression a+b.**




| | |
|---|---|
| KILL = 1 | ANTIN= |
| ANTLOC=0 | ANTOUT= |
| KILL = 0 | ANTIN= |
| ANTLOC=1 | ANTOUT= |
| KILL = 1 | ANTIN= |
| ANTLOC=0 | ANTOUT= |

- **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 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.

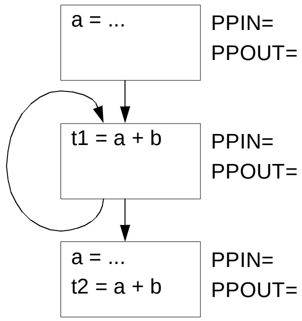
$INSERT[i] =$

  - Remove [upwards-exposed] computations where PPIN=1.

$DELETE[i] =$



## 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”

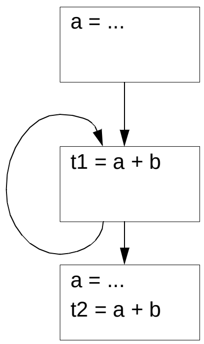
- **PPOUT : we want to place at output of this block only if**
  - we want to place at entry of all successors

$PPOUT[i] =$

- **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)

$PPIN[i] =$

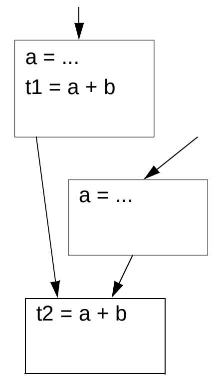
# “Placement Possible” Example 1




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



# “Placement Possible” Example 2




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



# “Placement Possible” Correctness

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

$PPIN[i] =$

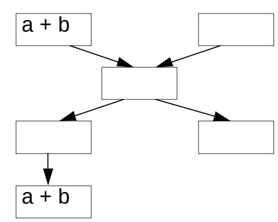
$PPOUT[i] =$

  - INSERT $\subseteq$ PPOUT $\subseteq$ ANTOUT, so insertion is safe.
- **Performance: Never increase the number 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.

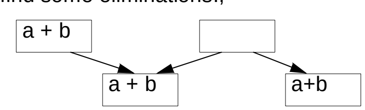


# Morel-Renvoise Limitations

- **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.**

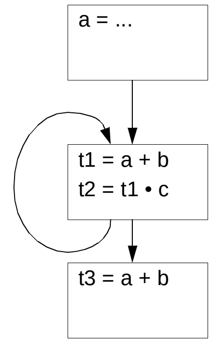
## Related Work

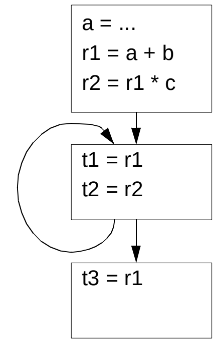
- **Don’t need heuristic**
  - Dhamdhere, Drechsler-Stadel, Knoop,et.al.
  - use restricted flow graph or allow edge placements.
- **Data flow can be separated into unidirectional passes**
  - Dhamdhere, Knoop, et. al.
- **Improvement still tied to accuracy of computational model**
  - Assumes performance depends only on the number of computa-tions 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.**



## Eliminating Complex Expressions

- **Expression (a+b)•c:**



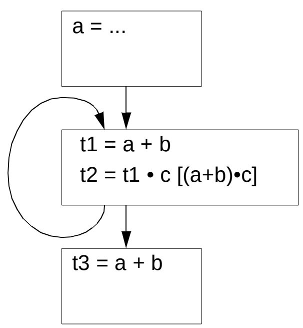




- **How can we do this?**
  - Consider 1 expression at a time, from top to bottom. - laborious.
  - Eliminate temporaries, build explicit complex expressions.



## Eliminating Complex Expressions 2

- **If we know actual computed expression, can do sub/expr in parallel:**



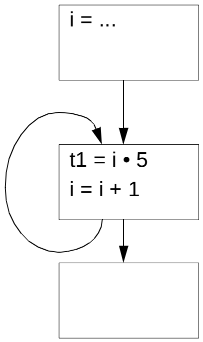

  - Only global operand assignments KILL the expression.
  - Restriction on placement: Additional expr occurrences never cause computation to be placed later in flow graph.



## Strength Reduction (Joshi-Dhamdhere 82)

- **Suppose the expression x = i • k is available.**
  - Assignment i = i + 1 kills it, but recomputing x is trivial: x = x + k
- **Distinguish between fast and slow computations:**
  - “one-unit” definition: x = x + k
  - “Q-unit” definition: x = i • k
- **One Q-unit definition is worth many one-unit definitions.**
  - Consider “killing” instruction which allows simple recomputation to be transparent to Q-unit computations:
  - i = i + c KILLs i + 3 but is X-Transparent to i • k.
  - i = x + y kills i • k as well (XKILL)

## Strength Reduction Example




| | | |
|---|---|---|
| XKILL = 1 | XPAVIN=0 | XPPIN= |
| XAVLOC=0 | XPAVOUT=0 | |
| ANTLOC=0 | XAVOUT=0 | XPPOUT= |

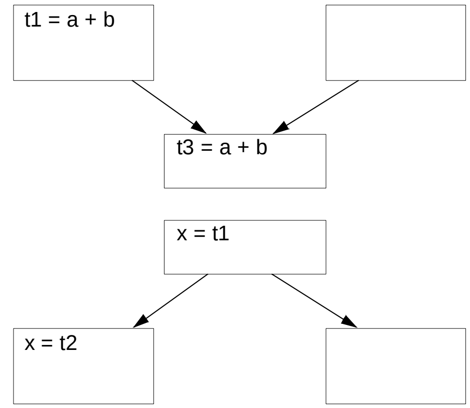
| | | |
|---|---|---|
| XKILL = 0 | XPAVIN=1 | XPPIN= |
| XAVLOC=1 | XPAVOUT=1 | |
| ANTLOC=1 | XAVOUT=1 | XPPOUT= |

- **Two placement computations - Q-unit, one-unit insertion**



## Store Redundancy

- **Dual problem with computation redundancy**:

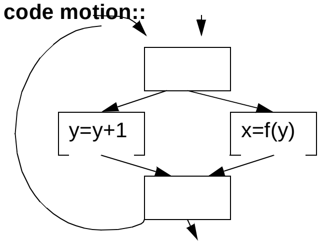



- **First store partially redundant.**



## Profiling feedback

- **Speculative code motion::**




- **Program quality = expected number of total computations, with expectation guided by a profiling run of program.**
- **Label flow graph edges with number of times taken.**
- **“Computationally Optimal” solution places computations on edges in a Min-Cut(killnodes,usenodes).**