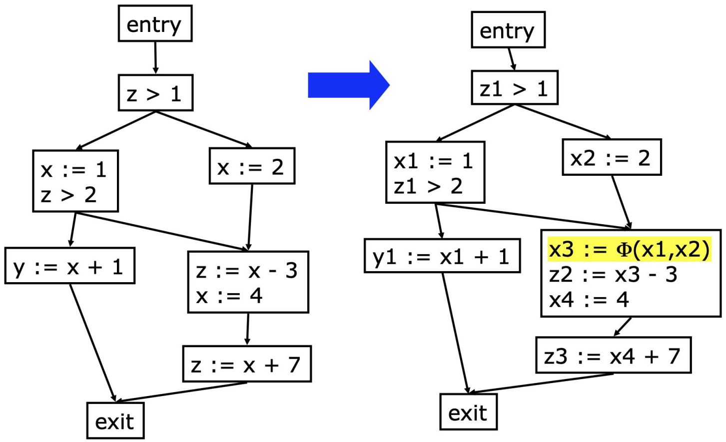
# Partial Redundancy Elimination in SSA Form

Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow

Presented by Group 19:
Peter Zhong, Zhengjie Xu, Zhongqian Duan, and Katsumi Ibaraki

# Recap on SSA and PRE

- Static Single-Assignment (SSA) Form
  - Each assignment to a variable is given a unique name
  - All uses reached by that assignment are renamed

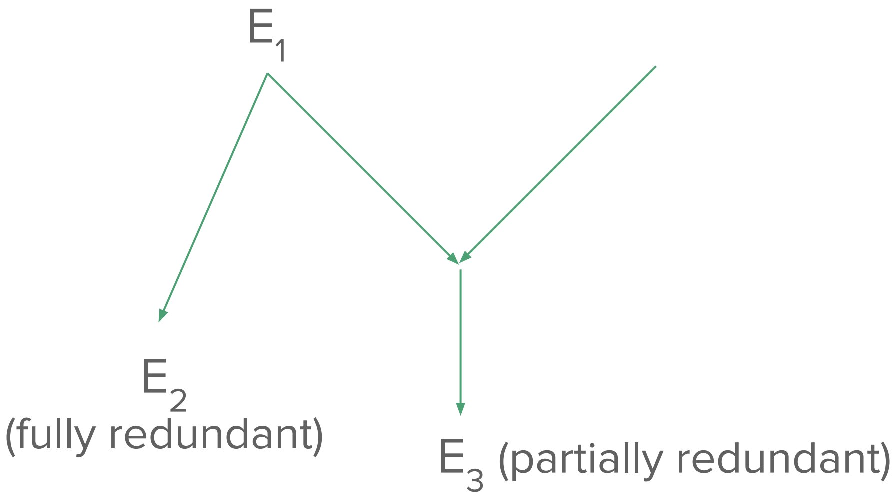


# Recap on SSA and PRE

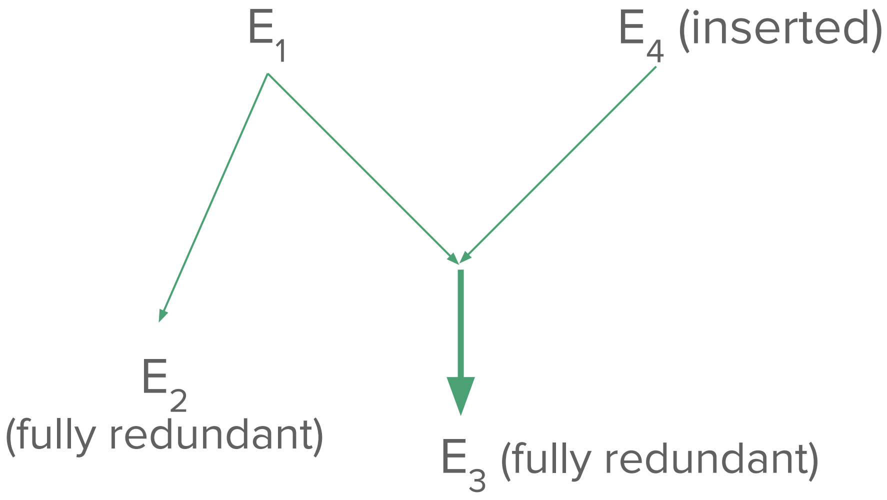
- Partial Redundancy Elimination (PRE)
  - Eliminate expressions that are redundant on some but not necessarily all paths
  - Partially redundant expression → fully redundant
    - Insert the partially redundant expression on the paths that do not already compute it

# Recap on SSA and PRE

- Partial Redundancy Elimination (PRE)
  - Eliminate expressions that are redundant on some but not necessarily all paths




(a) before PRE



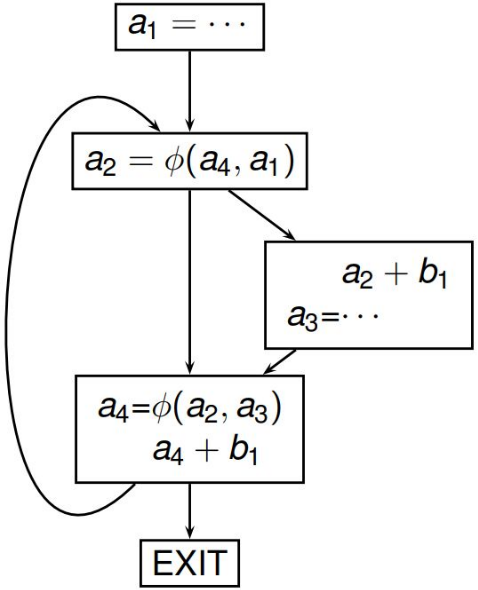

(b) after PRE

# SSAPRE Algorithm

- Assumptions
  - Input is a program in SSA form
  - Prior computation of the dominator tree (DT) and iterated dominance frontiers (DF+)
  - Each φ assignment has the property that its left-hand side and all of its operands are versions of the same original program variable
  - The live ranges of different versions of the same original program variable do not overlap

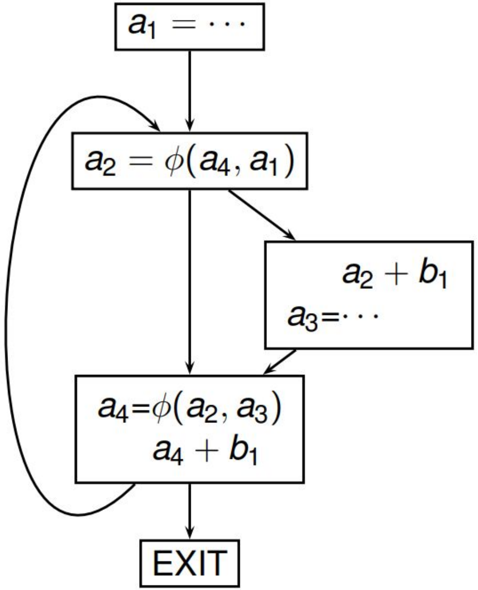
# SSAPRE Algorithm

- Step 1: The *Φ-Insertion* Step
  - Similar to SSA Phi insertion, but for expressions instead of variables
  - Identify all **lexically identical** expressions
    - Same base variable and same operand



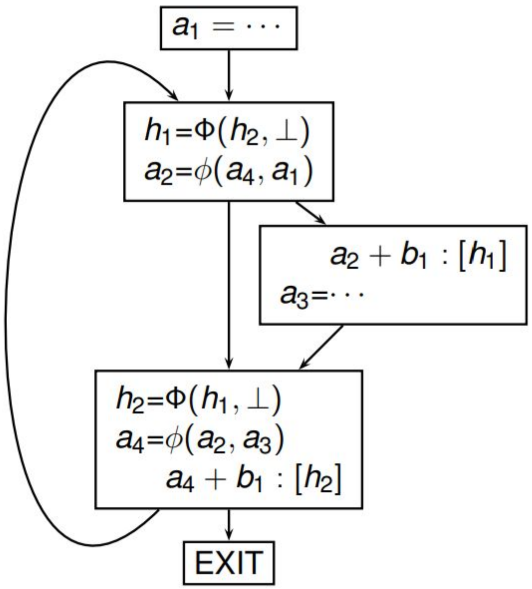
# SSAPRE Algorithm

- Step 1: The *Φ-Insertion* Step
  - Insert Phi nodes at
    - Iterated dominance frontier (IDF)
      - Same as SSA Phi insertion
    - When one variable of the expression is defined by a Phi node
      - An alteration of expression



# SSAPRE Algorithm

- Step 2: The *Rename* Step
  - Conducts a preorder traversal of the dominator tree, while maintaining both variable and expression stacks
  - Three types of expression occurrences:
    - Real occurrences
    - Φ nodes inserted in the Φ-Insertion step
    - Φ operands occurring at predecessor block ends

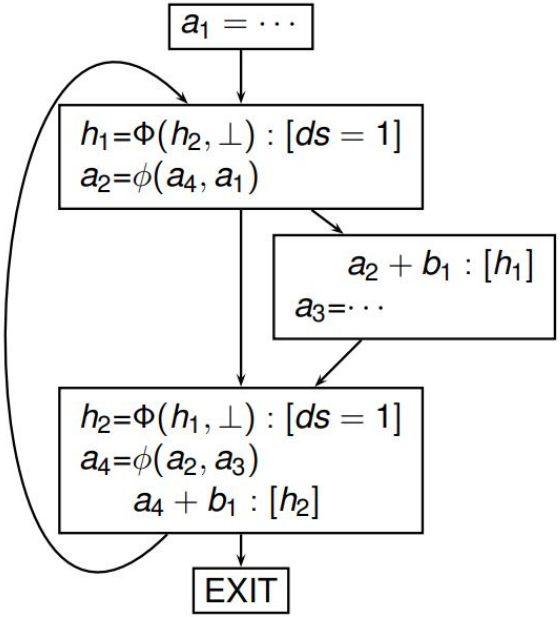


# SSAPRE Algorithm

- Step 3: The *DownSafety* Step
  - Insertions must be “down-safe”
  - A Φ computation is not down-safe if there is a path to EXIT from Φ along which the result of Φ is:
    - not used
    - used only as an operand of another Φ that itself is NOT down-safe

# SSAPRE Algorithm

- Step 3: The *DownSafety* Step
  - Begins at each Φ that is initially not marked down safe
  - Searches along upward edges, clearing the down safe flag for each Φ visited
  - *HasRealUse*: Real occurrence of an expression



# SSAPRE Algorithm

- Step 4: The *WillBeAvail* Step
  - The set of Φ where the expression must be available in any computationally optimal placement
  - Consist of two parts:
    - CanBeAvail
      - Φs for which E is either available or anticipable or both
    - Later
      - Φs that are CanBeAvail, but do not reach any real occurrence of E
  - WillBeAvail = CanBeAvail Λ ¬ Later

# SSAPRE Algorithm

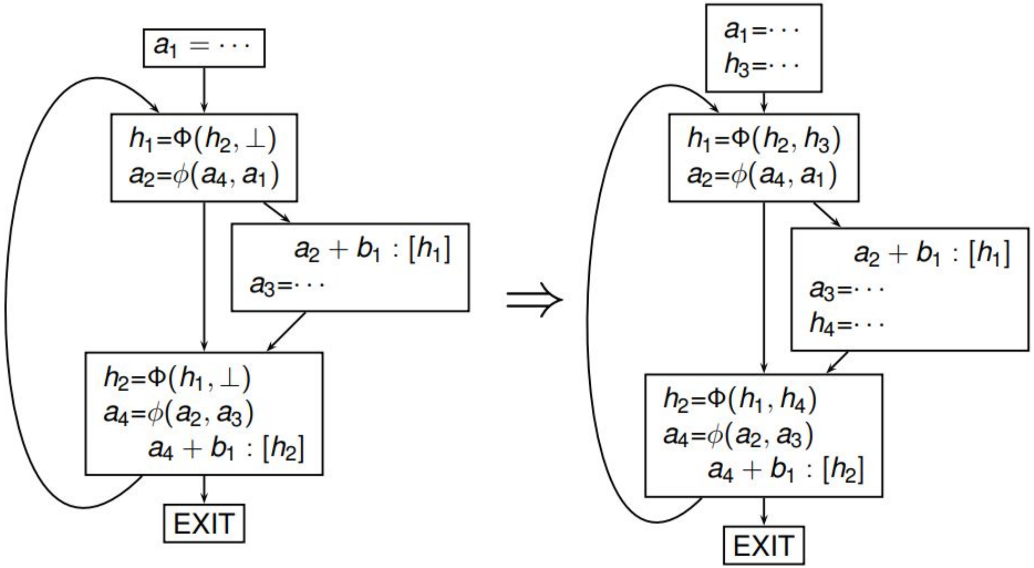
- *CanBeAvail*
  - Set Boundary Φs to be false
    - Not down-safe, and
    - At least one argument is ⊥
  - Propagate false value along the chain of def-use to other Φs
    - exclude edges along which HasRealUse is true
- *Later*
  - Initialize Later to true wherever CanBeAvail is true, otherwise false
  - Assign false for Φs with at least one operand with HasRealUse flag true
  - Propagate false value forward to other Φs

# SSAPRE Algorithm

- Step 5: The *Finalize* Step
  - Initializes AvailDef data structure.
  - Analyzes expressions in a control flow graph.
  - Updates and substitutes expression definitions.
  - Handles PHI nodes and operand traversals.
- Step 6: The *CodeMotion* Step
  - Iterates over pairs of expressions and instructions.
  - Handles variable or constant expressions by replacing instruction uses.
  - Processes and skips certain expressions based on conditions.
  - Computes substitutions for expressions and handles different cases.

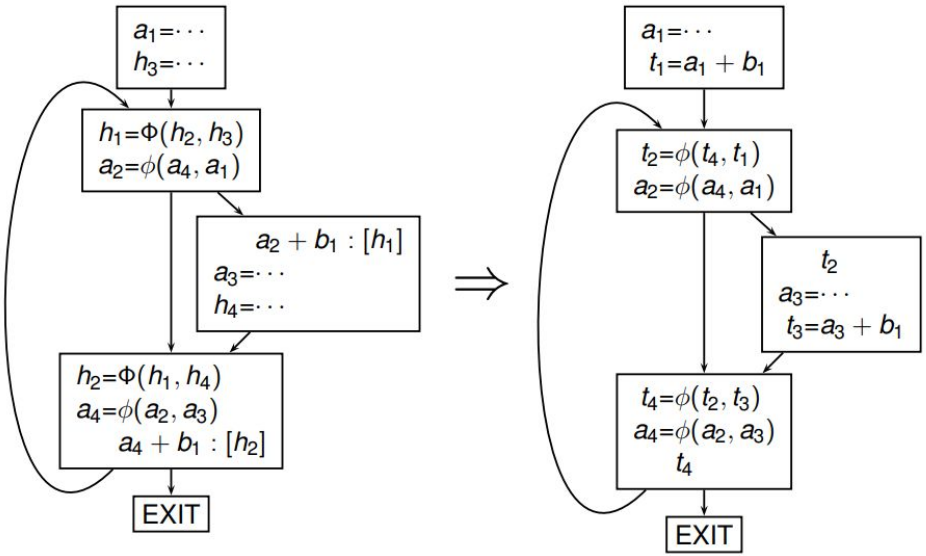
# SSAPRE Algorithm

- Step 5: The *Finalize* Step



# SSAPRE Algorithm

- Step 6: The *CodeMotion* Step



# Analysis

- Time complexity: **O(n(E + V))**
  - E and V: number of edges and vertices in SSA graph
  - Step 2-6 are all linear w.r.t (E + V)
  - Phi Insertion is normally O(V^2) because of IDF
    - But there are linear algorithms
  - Bit-vector PRE algorithms have cubic complexity

# Performance

- Compared against bit-vector based PRE
  - Program runtime: no noticeable difference
  - Compile time: Varies

| SPECint95 Benchmarks | go | m88ksim | gcc | compress | li | ijpeg | perl | vortex |
|---|---|---|---|---|---|---|---|---|
| Bit-vector PRE (T1) | 116900 | 4850 | 886360 | 100 | 12950 | 10340 | 98840 | 62950 |
| SSAPRE (T2) | 151260 | 4440 | 339160 | 60 | 5090 | 11200 | 34970 | 53000 |
| Ratio T2/T1 | 1.293 | 0.915 | 0.382 | 0.600 | 0.393 | 1.083 | 0.353 | 0.841 |

| SPECfp95 Benchmarks | tomcatv | swim | su2cor | hydro2d | mgrid | applu | turb3d | apsi | fpppp | wave5 |
|---|---|---|---|---|---|---|---|---|---|---|
| Bit-vector PRE (T1) | 40 | 170 | 500 | 7080 | 500 | 5060 | 2420 | 37930 | 1450 | 94150 |
| SSAPRE (T2) | 60 | 400 | 700 | 8780 | 1400 | 9450 | 5000 | 93960 | 1980 | 85800 |
| Ratio T2/T1 | 1.500 | 2.352 | 1.399 | 1.240 | 2.799 | 1.867 | 2.066 | 2.477 | 1.365 | 0.911 |

Table 2: Time (in msec.) spent in Partial Redundancy Elimination in compiling SPECint95 and SPECfp95

# Performance

- Analysing performance results
  - Larger procedures benefit more from SSAPRE
    - Sparse FRG smaller than CFG
  - Prototype implementation, needs further tuning
  - Algorithmic complexity

# Future Work

- Further investigation wide compile time difference
- Improve SSA graph construction through characterization
- Extending SSA dataflow characterization to other classical optimization techniques
  - Code hoisting, load/store redundancies

# Conclusion/Commentary

- SSAPRE takes advantage of SSA form to present a sparse approach to PRE
- Using SSA to solve dataflow problem related to expressions
- Good algorithmic complexity compared to bit-vector based PRE algorithms