Value Based Speculative Partial Redundancy Elimination With Edge Profiling Information on LLVM

Yuanli Zhu, Yihao Huang, Jing Zhu, Yujian Liu

Content

- **Introduction**
- Algorithm
- Implementation
- Evaluation and demo

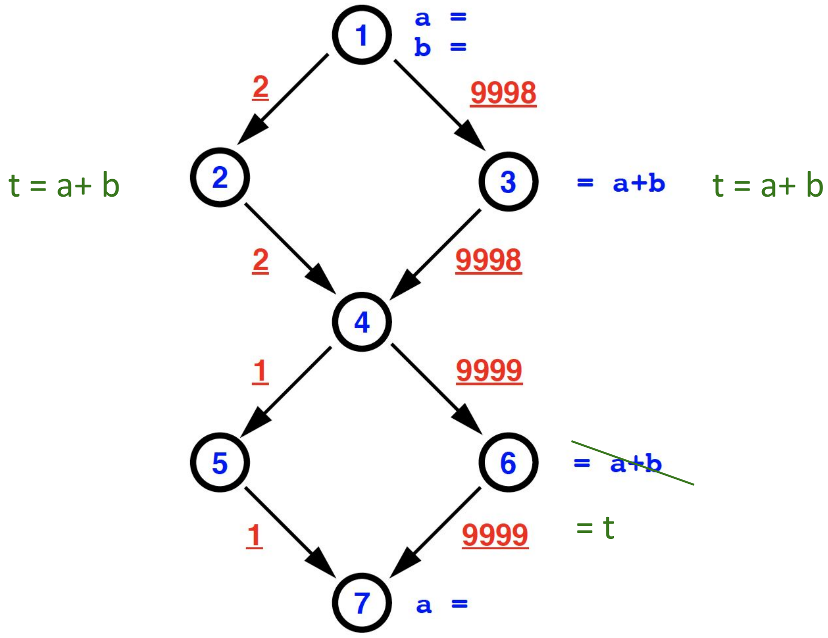
Introduction: Concept overview

Partial Redundancy Elimination (PRE):
- An optimization technique that eliminates the partial redundancy of some paths in a program

Value-based PRE:
- Will be able to optimize for expressions that have the same computation but different operands, e.g. c=a, r1 = a+b, r2 = c+b
- Optimize by keep track of the values instead of the operands

Speculation-based PRE:
- use runtime information to do more aggressive PRE

Introduction: SPRE



Introduction: Contribution of our work

Combined the value-based PRE and speculation-based PRE, and designed a new Speculation-based Global Value Numbering Partial Redundancy Elimination Algorithm

Implemented in an LLVM Pass and are based on the SSA form instead of lexically equivalent expressions

Evaluated our algorithm using two simple case studies to show the correctness of our algorithm and test its runtime on a subset of the LLVM SingleSource Benchmarks

Content

- Introduction
- **Algorithm**
- Implementation
- Evaluation and demo

Algorithm - Overview

1. Match expressions to value numbers and vice versa.

2. Build sets for future usages.

3. Apply MC-PRE to construct ISTMAP.

4. Add a new block to insert the expressions and replace old uses of them.

5. Conduct dead code elimination.

```
Data: OrigF (input IR), PF (Profiling Data)
Result: OptF (Optimized IR after SPGVNPRE)
VN, VNR = ValueNumbering(OrigF);
PANT_IN, AVAI_OUT = BuildSets(OrigF);
number = 0;
ISTMAP = {};
while number < VNR.size() do
    essentialEdges = findEssentialEdges(PANT_IN,
     AVAI_OUT, number);
    RFG = ReducedFlowGraph(essentialEdges, PF);
    CutEdges = minCut(RFG);
    for each edge in CutEdges do
        ISTMAP[edge].add(number);
    end
    number++;
end
tmpF = InsertAndReplace(origF, AVAI_OUT,
 PANT_IN, ISTMAP, VNR);
OptF = clean(tmpF);
```

Algorithm 1: Overall algorithm of SPGVNPRE.

Algorithm - Overview

1. Match expressions to value numbers and vice versa.

2. Build sets for future usages.

3. Apply MC-PRE to construct ISTMAP.

4. Add a new block to insert the expressions and replace old uses of them.

5. Conduct dead code elimination.

```
Data: OrigF (input IR), PF (Profiling Data)
Result: OptF (Optimized IR after SPGVNPRE)
VN, VNR = ValueNumbering(OrigF);
PANT_IN, AVAI_OUT = BuildSets(OrigF);
number = 0;
ISTMAP = {};
while number < VNR.size() do
    essentialEdges = findEssentialEdges(PANT_IN,
     AVAI_OUT, number);
    RFG = ReducedFlowGraph(essentialEdges, PF);
    CutEdges = minCut(RFG);
    for each edge in CutEdges do
        ISTMAP[edge].add(number);
    end
    number++;
end
tmpF = InsertAndReplace(origF, AVAI_OUT,
 PANT_IN, ISTMAP, VNR);
OptF = clean(tmpF);
```
Algorithm 1: Overall algorithm of SPGVNPRE.

Algorithm - Overview

1. Match expressions to value numbers and vice versa.

2. Build sets for future usages.

3. Apply MC-PRE to construct ISTMAP.

4. Add a new block to insert the expressions and replace old uses of them.

5. Conduct dead code elimination.

```
Data: OrigF (input IR), PF (Profiling Data)
Result: OptF (Optimized IR after SPGVNPRE)
VN, VNR = ValueNumbering(OrigF);
PANT_IN, AVAI_OUT = BuildSets(OrigF);
number = 0;
ISTMAP = {};
while number < VNR.size() do
    essentialEdges = findEssentialEdges(PANT_IN,
     AVAI_OUT, number);
    RFG = ReducedFlowGraph(essentialEdges, PF);
    CutEdges = minCut(RFG);
    for each edge in CutEdges do
        ISTMAP[edge].add(number);
    end
    number++;
end
tmpF = InsertAndReplace(origF, AVAI_OUT,
 PANT_IN, ISTMAP, VNR);
OptF = clean(tmpF);
```

Algorithm 1: Overall algorithm of SPGVNPRE.

Algorithm - Overview

1. Match expressions to value numbers and vice versa.

2. Build sets for future usages.

3. Apply MC-PRE to construct ISTMAP.

4. Add a new block to insert the expressions and replace old uses of them.

5. Conduct dead code elimination.

```
Data: OrigF (input IR), PF (Profiling Data)
Result: OptF (Optimized IR after SPGVNPRE)
VN, VNR = ValueNumbering(OrigF);
PANT_IN, AVAI_OUT = BuildSets(OrigF);
number = 0;
ISTMAP = {};
while number < VNR.size() do
    essentialEdges = findEssentialEdges(PANT_IN,
      AVAI_OUT, number);
    RFG = ReducedFlowGraph(essentialEdges, PF);
    CutEdges = minCut(RFG);
    for each edge in CutEdges do
        ISTMAP[edge].add(number);
    end
    number++;
end
tmpF = InsertAndReplace(origF, AVAI_OUT,
  PANT_IN, ISTMAP, VNR);
OptF = clean(tmpF);
```

Algorithm 1: Overall algorithm of SPGVNPRE.

Algorithm - Overview

1. Match expressions to value numbers and vice versa.

2. Build sets for future usages.

3. Apply MC-PRE to construct ISTMAP.

4. Add a new block to insert the expressions and replace old uses of them.

5. Conduct dead code elimination.

Data: OrigF (input IR), PF (Profiling Data)
Result: OptF (Optimized IR after SPGVNPRE)

```
VN, VNR = ValueNumbering(OrigF);
PANT_IN, AVAI_OUT = BuildSets(OrigF);
number = 0;
ISTMAP = {};
while number < VNR.size() do
    essentialEdges = findEssentialEdges(PANT_IN,
      AVAI_OUT, number);
    RFG = ReducedFlowGraph(essentialEdges, PF);
    CutEdges = minCut(RFG);
    for each edge in CutEdges do
        ISTMAP[edge].add(number);
    end
    number++;
end
tmpF = InsertAndReplace(origF, AVAI_OUT,
  PANT_IN, ISTMAP, VNR);
OptF = clean(tmpF);
```

Algorithm 1: Overall algorithm of SPGVNPRE.

Algorithm - Overview

1. Match expressions to value numbers and vice versa.

2. Build sets for future usages.

3. Apply MC-PRE to construct ISTMAP.

4. Add a new block to insert the expressions and replace old uses of them.

5. Conduct dead code elimination.

Data: OrigF (input IR), PF (Profiling Data)
Result: OptF (Optimized IR after SPGVNPRE)

```
VN, VNR = ValueNumbering(OrigF);
PANT_IN, AVAI_OUT = BuildSets(OrigF);
number = 0;
ISTMAP = {};
while number < VNR.size() do
    essentialEdges = findEssentialEdges(PANT_IN,
     AVAI_OUT, number);
    RFG = ReducedFlowGraph(essentialEdges, PF);
    CutEdges = minCut(RFG);
    for each edge in CutEdges do
        ISTMAP[edge].add(number);
    end
    number++;
end
tmpF = InsertAndReplace(origF, AVAI_OUT,
 PANT_IN, ISTMAP, VNR);
OptF = clean(tmpF);
```

Algorithm 1: Overall algorithm of SPGVNPRE.

Algorithm - Value Numbering

- Maintain two hash tables, VN and VNR, mapping a value number to a list of expressions that are actually the same and vice versa.
- For example, if *v1* contains *r1* and *r2*, and *v3* contains *r3*, then *r3* + *r1* and *r3* + *r2* should be the same.
    - i.e., VN contains the mapping [1] -> [r*3* + *r1, r3* + *r2*]

Algorithm - Build Sets

- Conduct a standard forward availability analysis

$$\begin{aligned}\text{AVAIL\_IN}(b) &= \text{AVAIL\_OUT}[\text{dom}(b)] \\ \text{AVAIL\_OUT}(b) &= \text{AVAIL\_IN}(b) \cup \text{PHI\_GEN}(b) \\ &\quad \cup \text{TMP\_GEN}(b)\end{aligned}$$

Algorithm - Build Sets

- Conduct a backward partial anticipability analysis

$$\text{PANT\_OUT}(b) = \begin{cases} \bigvee_{m \in succ(n)} \text{PANT\_IN}(m) & \text{if } |\text{succ}(b)| > 1 \\ \text{phi\_translate}(A[\text{succ}(b)], b, \text{succ}(b)) & \\ & \text{if } |\text{succ}(b)| = 1 \end{cases}$$

$$\text{PANT\_IN}(b) = \text{PANT\_OUT}(b) \cup \text{EXP\_GEN}(b) \cap (\neg \text{TMP\_GEN}(b))$$

Algorithm - Build Sets

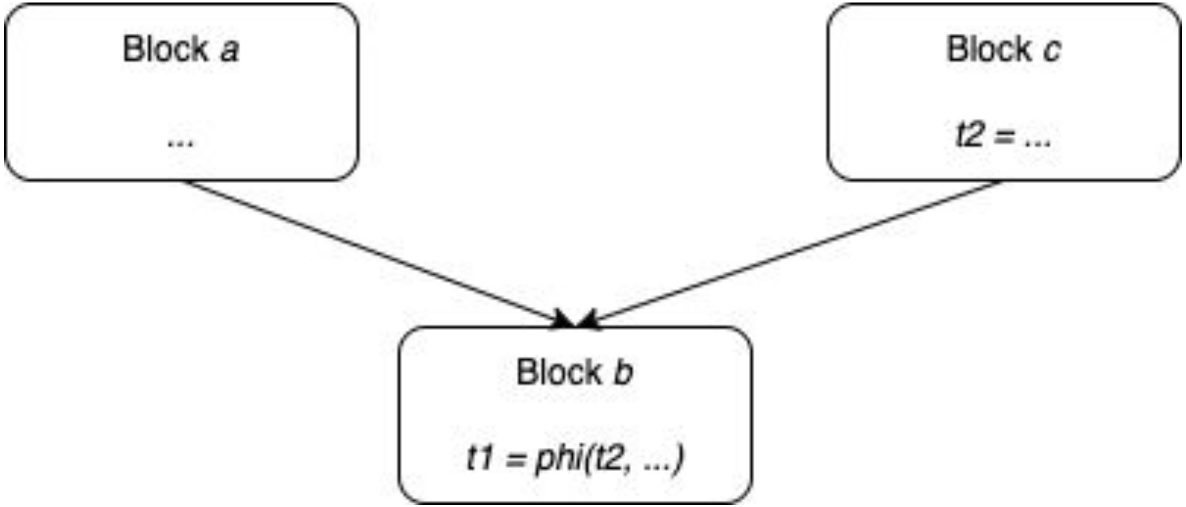
- Conduct a backward partial anticipability analysis

$$
\mathrm{PANT\_OUT}(b) = \begin{cases} \bigvee_{m \in succ(n)} \mathrm{PANT\_IN}(m) & \text{if } |\mathrm{succ}(b)| > 1 \\ \boxed{\mathrm{phi\_translate}}(A[\mathrm{succ}(b)], b, \mathrm{succ}(b)) & \\ & \text{if } |\mathrm{succ}(b)| = 1 \end{cases}
$$

$$
\mathrm{PANT\_IN}(b) = \mathrm{PANT\_OUT}(b) \cup \mathrm{EXP\_GEN}(b) \cap (\neg \mathrm{TMP\_GEN}(b))
$$

Algorithm - Build Sets

- Conduct a backward partial anticipability analysis
  - phi_translate



Algorithm- Find the Optimal Insertion Map

Covered in our paper presentation

ISTMAP = {};
while *number* < *VNR.size()* **do**
  essentialEdges = findEssentialEdges(PANT_IN, AVAI_OUT, number);
  RFG = ReducedFlowGraph(essentialEdges, PF);
  CutEdges = minCut(RFG);
  for *each edge in CutEdges* **do**
    ISTMAP[edge].add(number);
  end
  number++;
end

Algorithm- Find the Optimal Insertion Map

1. Find Essential Edges

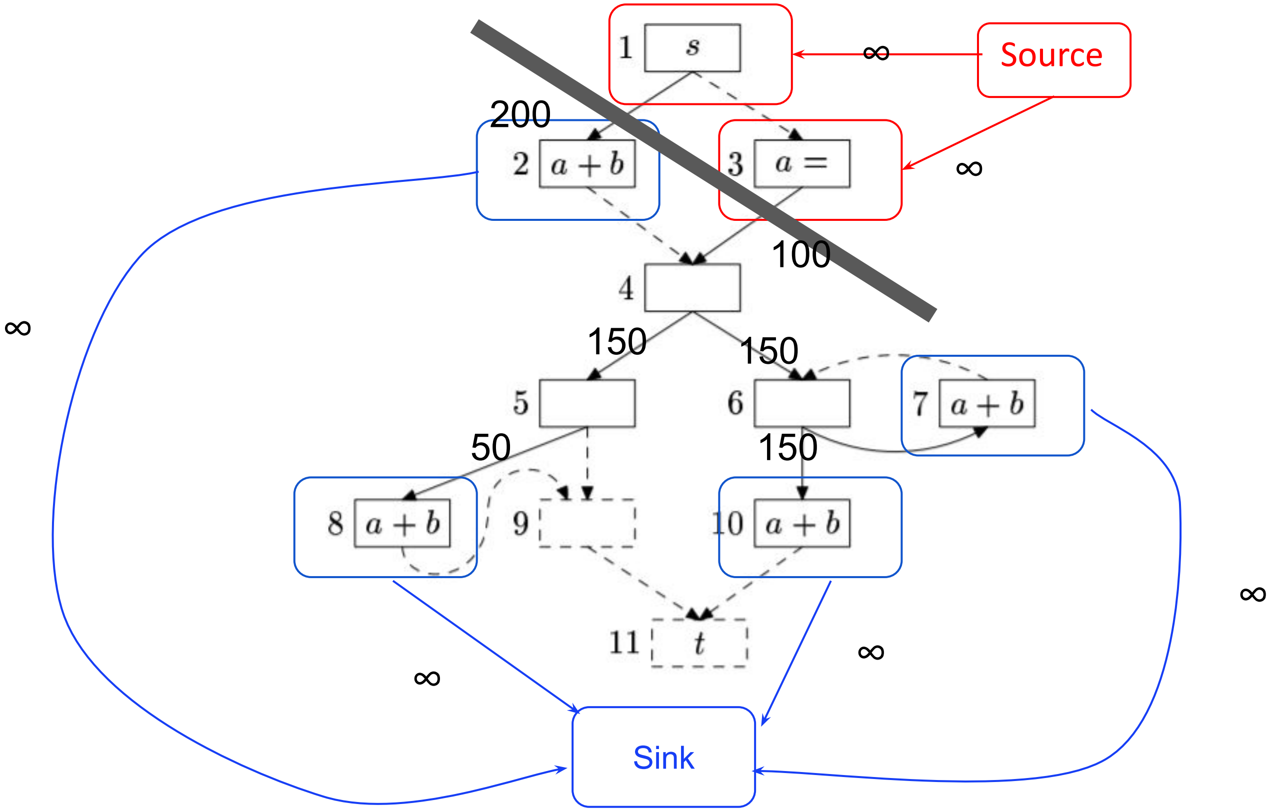
   Edge (u,v) is essential for n if n is in PANT_IN[v] but not in AVAI_OUT[u]

2. Construct a single-source, single-sink reduced flow graph
   - Only keep edges and nodes are essential
   - Weight of each edge is the execution times
   - Combine sources into one, sinks into one
   - Assign weights of new edges as infinity

3. Apply Minimum Cut

   We use Ford-Fulkerson Algorithm.

Algorithm- Find the Optimal Insertion Map



Algorithm- Insert and Replace

1. Create new blocks at cutting edge and insert new Definitions
2. Insert Phi Node at the dominance frontier
3. Replace use of old definitions

Insert Phi Nodes

SSA Step 1 - Phi Node Insertion

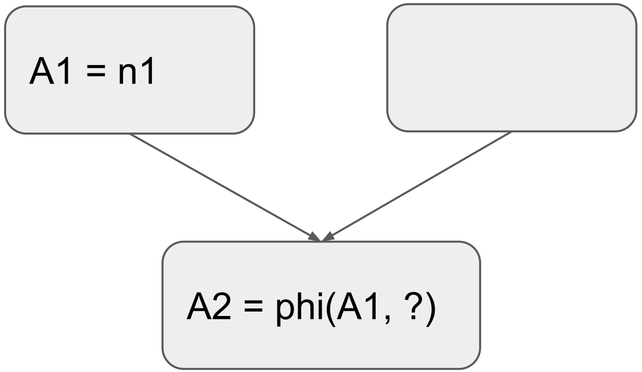
- Compute dominance frontiers
- Find global names (aka virtual registers)
  - Global if name live on entry to some block
  - For each name, build a list of blocks that define it
- Insert Phi nodes
  - For each global name n
    - For each BB b in which n is defined
      - For each BB d in b's dominance frontier
        - Insert a Phi node for n in d
        - Add d to n's list of defining BBs

Insert Phi Nodes

```
for each value number n do
    for each Instruction i in newVNmap[n] do
        b = basic block of i;
        for each basic block d in b's dominance
         frontier do
            if n is in PANT_IN[d] then
                Insert a Phi node for n in d;
                Add d to newVNmap[n];
            end
        end
    end
end
```

Difference:

1. Value Number instead of Variable
2. Phi node of n is only inserted if n is partial anticipated



Replace use of old definitions

SSA Step 2 – Renaming Variables

- Use an array of stacks, one stack per global variable (VR)
- Algorithm sketch
  - For each BB b in a preorder traversal of the dominator tree
    - Generate unique names for each Phi node
    - Rewrite each operation in the BB
      - Uses of global name: current name from stack
      - Defs of global name: create and push new name
    - Fill in Phi node parameters of successor blocks
    - Recurse on b's children in the dominator tree
    - <on exit from b> pop names generated in b from stacks

Replace use of old definitions

Difference:

1. One stack per value number instead of per global variable
2. Differentiate original Phi node and inserted Phi node
   - Inserted Phi: add incoming new definition
   - Original Phi: replace old definition with new one

```
Initialize stackMap with one stack per value number;
for each basic block b in a preorder tranversal of the
 dominator tree do
    for each instruction i in b do
        if i is not phi and its operand op has VN[p]
         = n then
            replace op with current definition from
             stackMap[n];
        end
        if i is a definitions and newVNmap[i] = n
         then
            push the definition on stackMap[n]
        end
    end
    for each successor s of b do
        for each phi p of s do
            if p is in newVNmap then
                Fill in phi node parameters with
                 stackMap[newVNmap[p]];
            else
                if p has operand op with VN[op] = n
                 then
                    replace op with current definition
                     from stackMap[n];
                end
            end
        end
    end
    Recurse on b's children in the dominator tree;
    On exit from b, pop definitions generated in b
     from stackMap;
end
```

Content

- Introduction
- Algorithm
- **Implementation**
- Evaluation and demo

Implementation

Five passes:

1. mem2reg (LLVM), make value numbering easier
2. Profiling
3. SGVNPRE
    - Built on gvnpre (LLVM)
    - minCut referenced GeeksforGeeks
4. dce (LLVM), dead code elimination
5. Merge blocks, result from splitting edges during insertion

```
Data: OrigF (input IR), PF (Profiling Data)
Result: OptF (Optimized IR after SPGVNPRE)
VN, VNR = ValueNumbering(OrigF);
PANT_IN, AVAI_OUT = BuildSets(OrigF);
number = 0;
ISTMAP = {};
while number < VNR.size() do
    essentialEdges = findEssentialEdges(PANT_IN,
     AVAI_OUT, number);
    RFG = ReducedFlowGraph(essentialEdges, PF);
    CutEdges = minCut(RFG);
    for each edge in CutEdges do
        ISTMAP[edge].add(number);
    end
    number++;
end
tmpF = InsertAndReplace(origF, AVAI_OUT,
 PANT_IN, ISTMAP, VNR);
OptF = clean(tmpF);
```

Algorithm 1: Overall algorithm of SPGVNPRE.

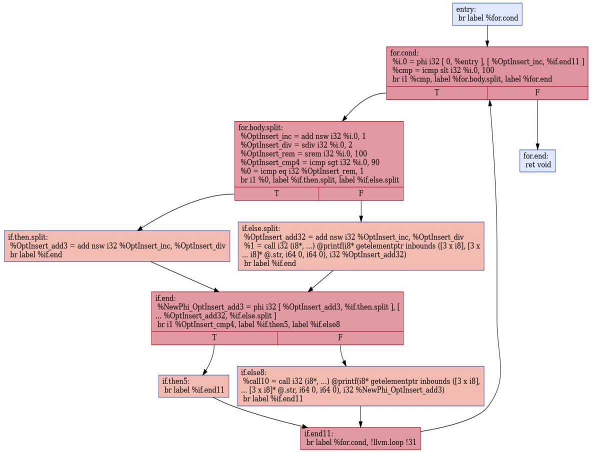
Content

- Introduction
- Algorithm
- Implementation
- **Evaluation and demo**

Case study: classic

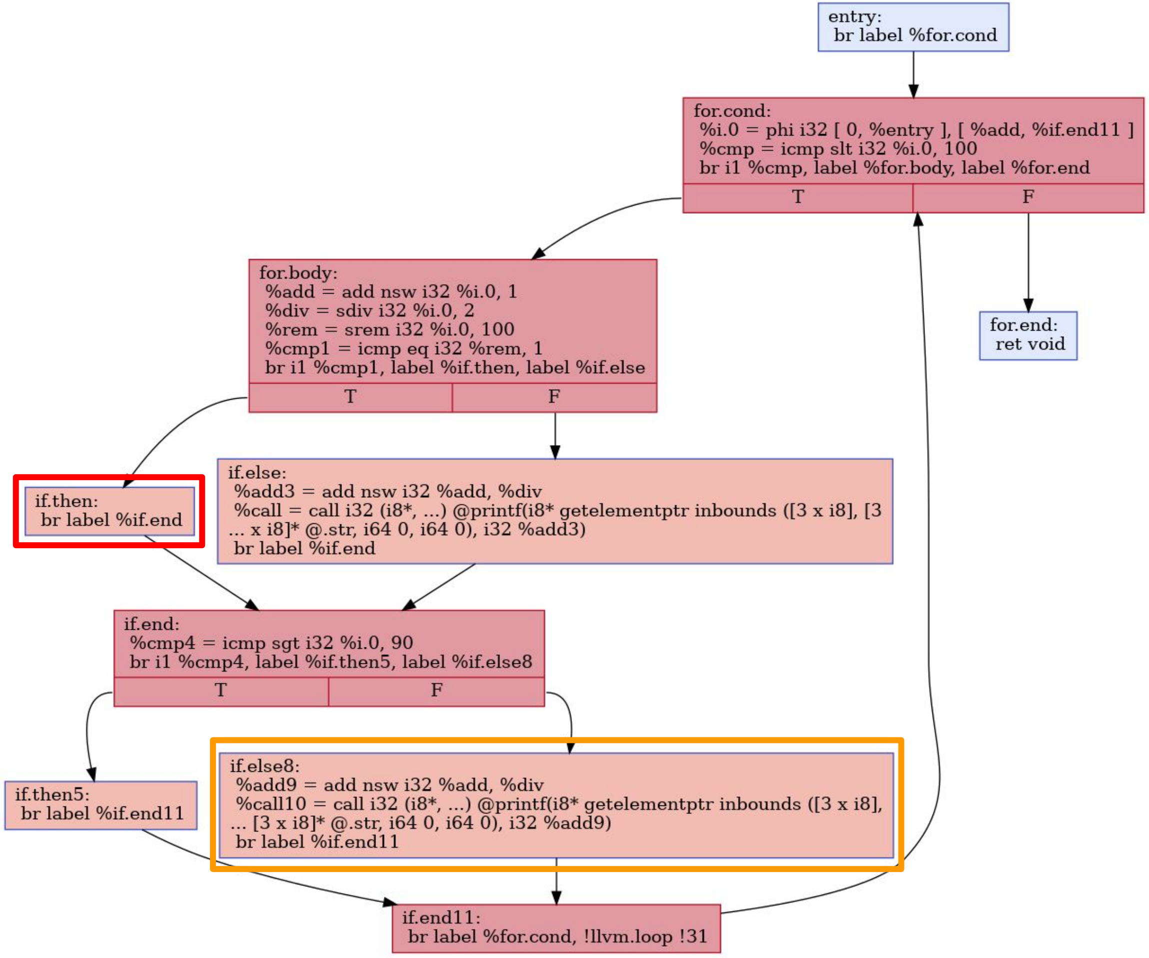
```c
void classic(){
    for(int i=0; i<100; i++){
        int a = i+1;
        int b = i/2;

        if(i%100==1){
            int c = a+1;
        }
        else{
            printf("%d", a+b);
        }

        if(i>90){
            int c = a+1;
        }
        else{
            printf("%d", a+b);
        }
    }
}
```

Case study: classic



CFG for 'classic' function

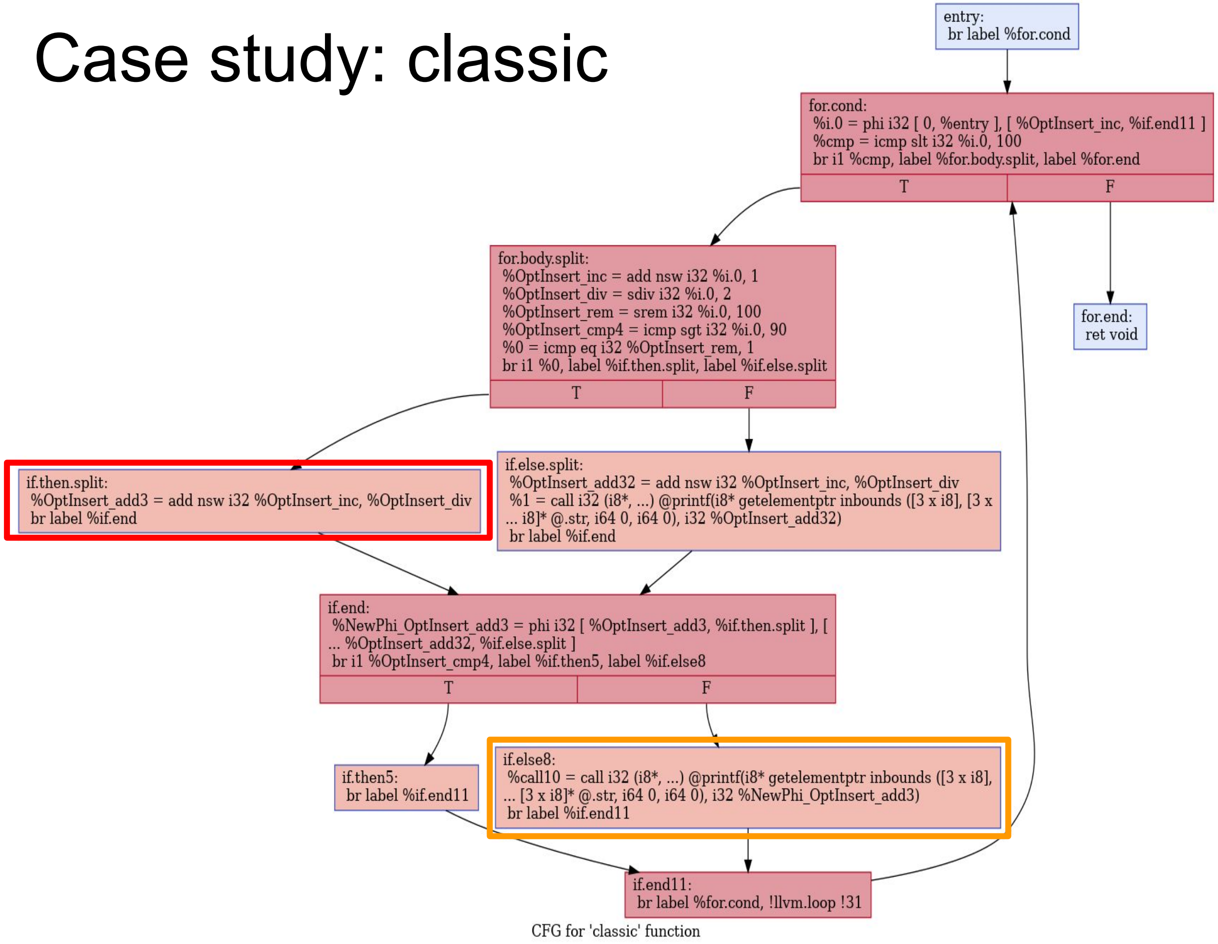
SGVNPRE



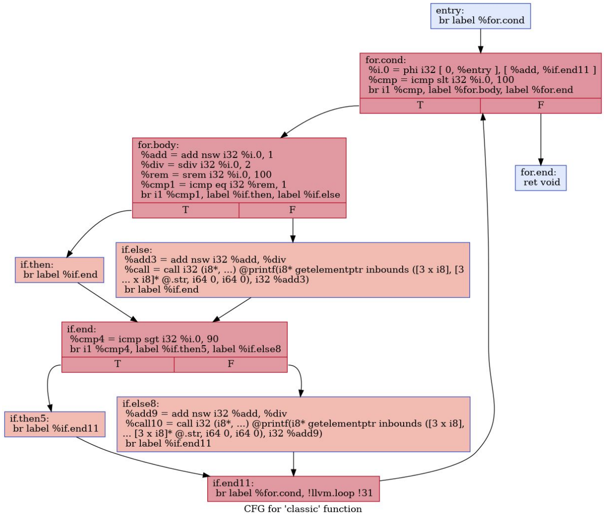
CFG for 'classic' function

GVNPRE

Case study: classic



SGVNPRE

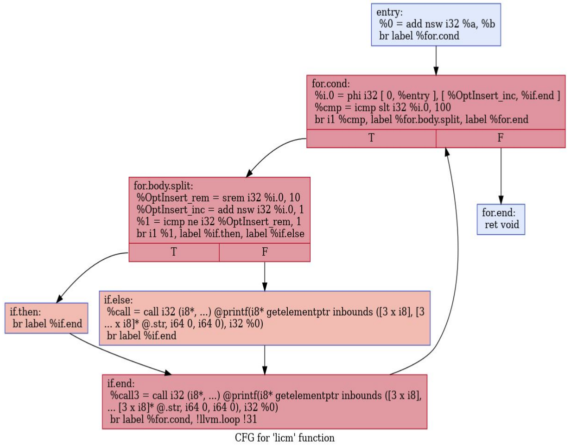


GVNPRE

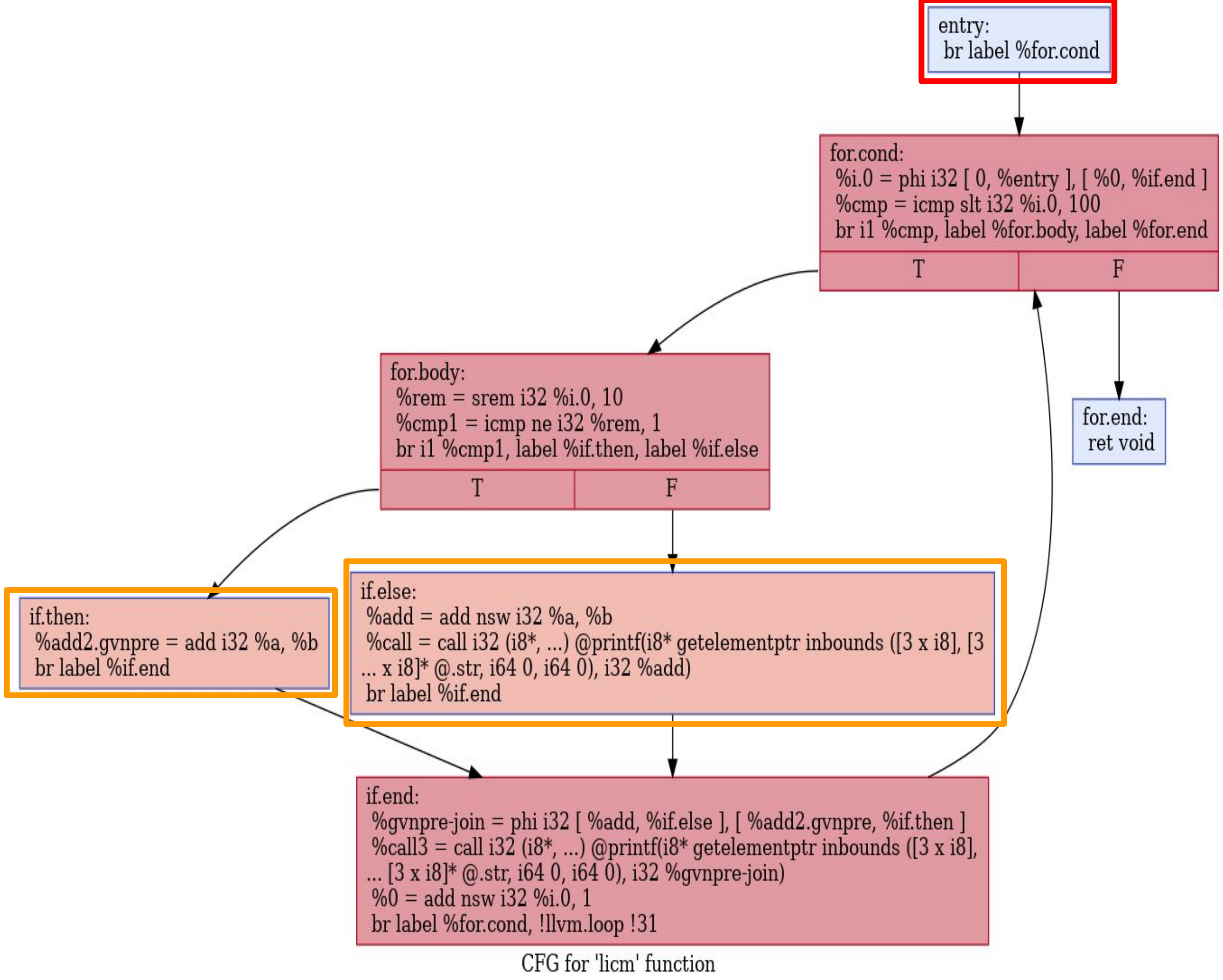
Case study: licm

```
void licm(int a, int b){
    for(int i=0; i<100; i++){
        if(i%10!=1){
            int c = a;
        }
        else{
            int  k = a+b;
            printf("%d", k);
        }

        printf("%d", a+b);

    }

}
```

Case study: licm

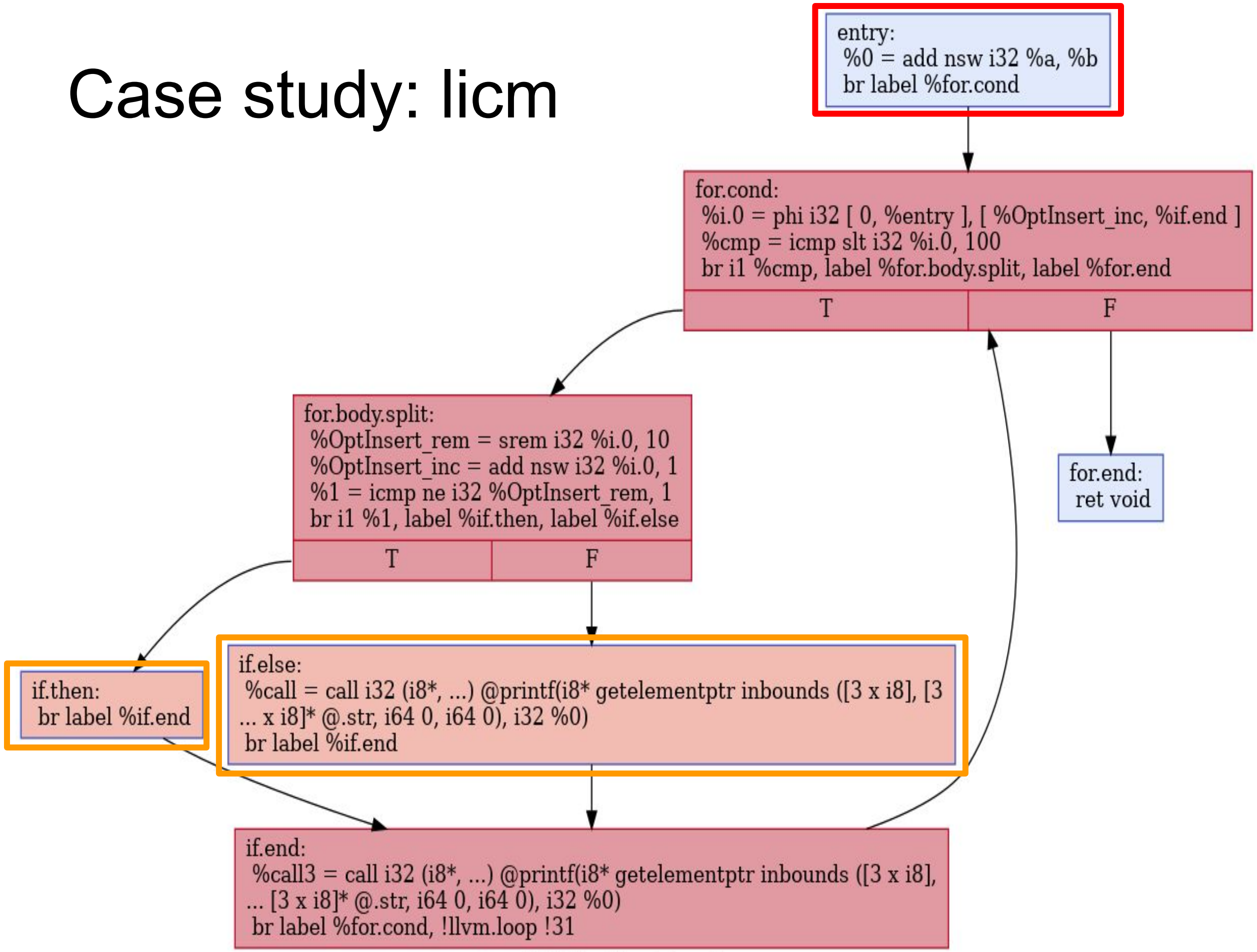


SGVNPRE



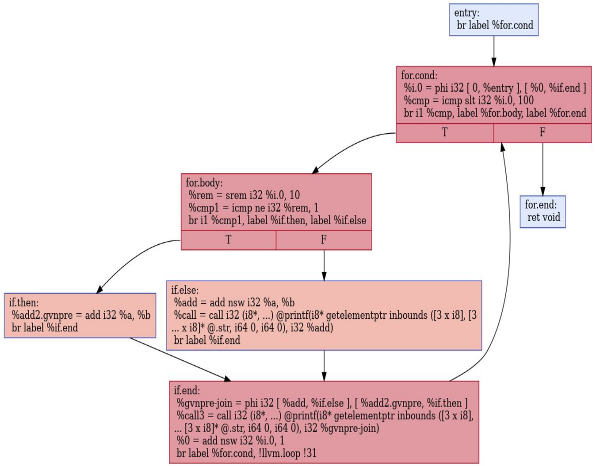
GVNPRE

Case study: licm



CFG for 'licm' function

SGVNPRE

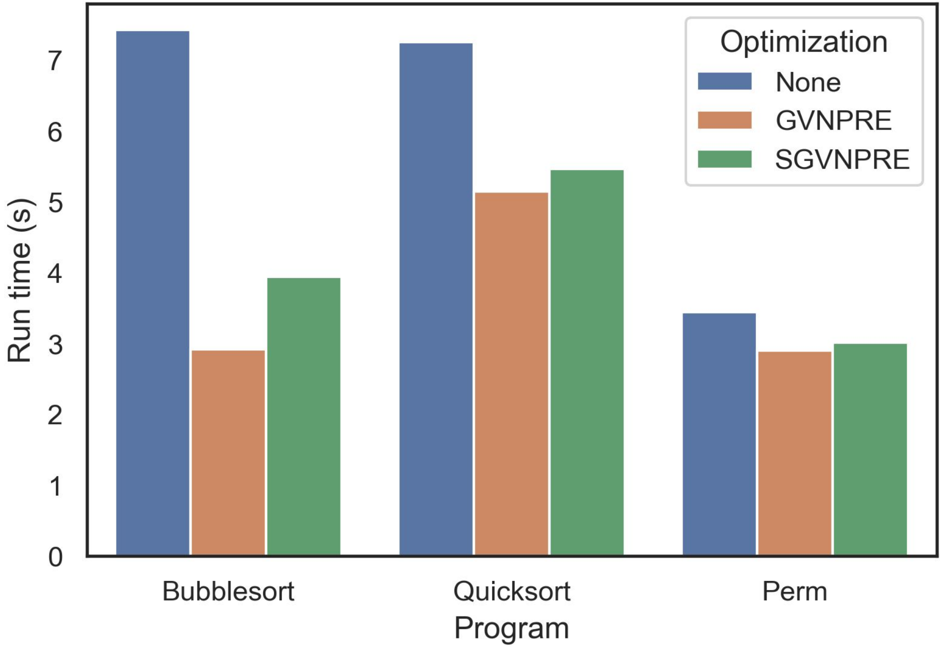


CFG for 'licm' function

GVNPRE

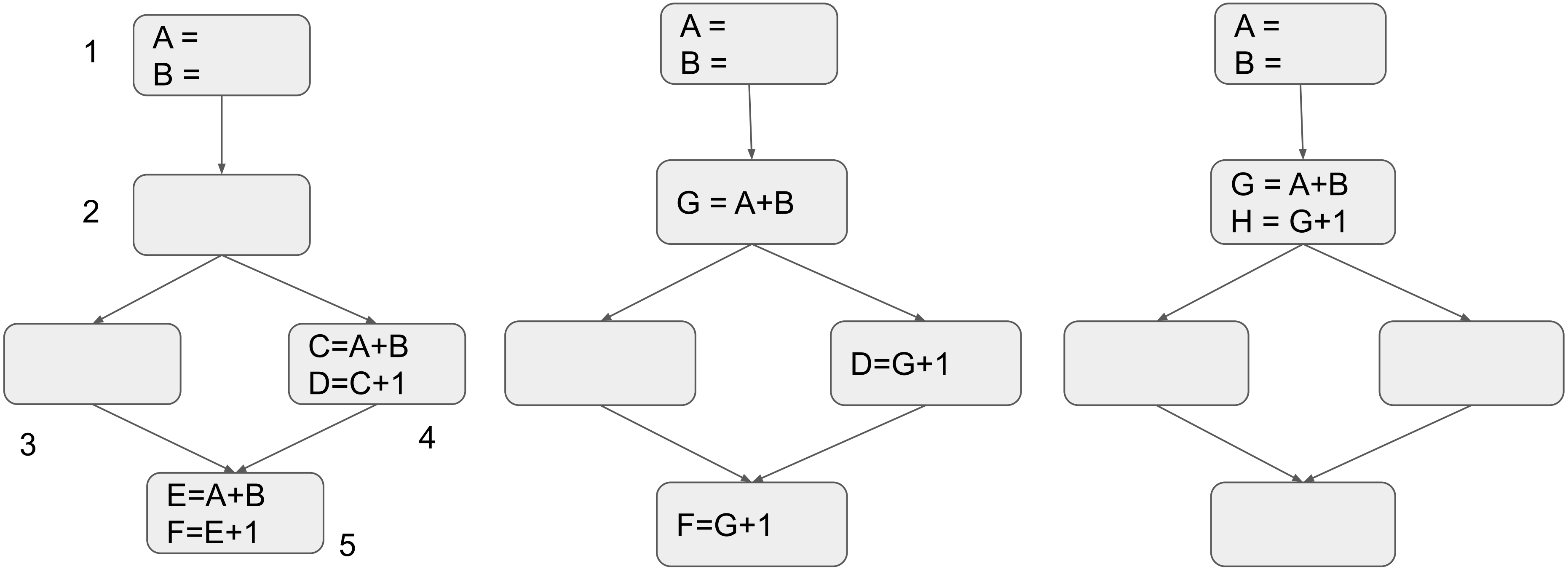
Benchmark evaluation

- LLVM SingleSource Benchmark:



Reason for why slower than LLVM GVNPRE

more rounds of SGVNPRE pass maybe needed



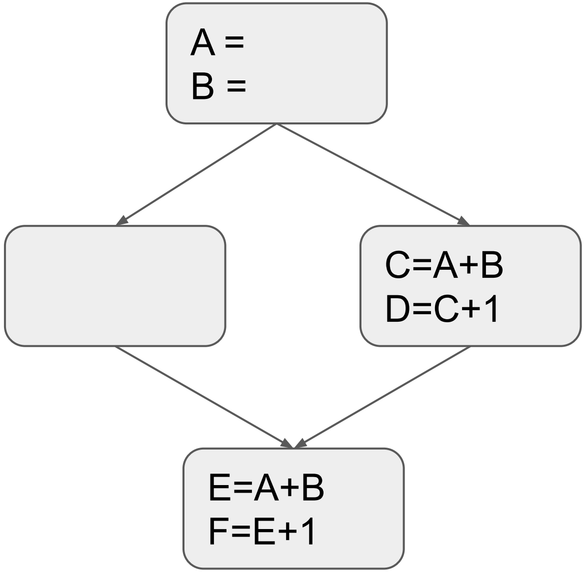
Before SGVNPRE

After 1st SGVNPRE

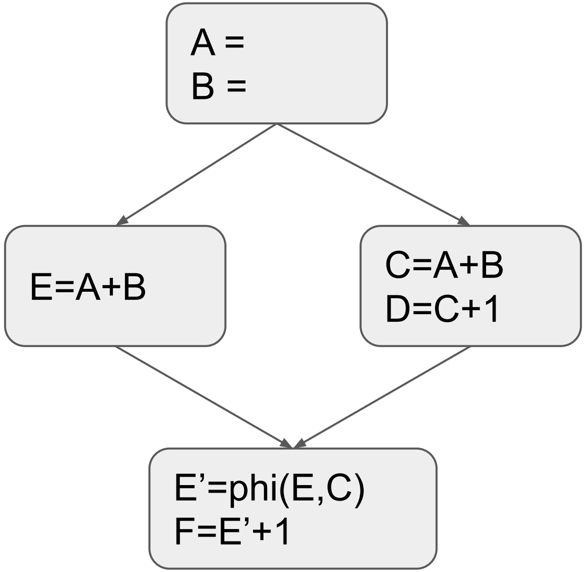
After 2nd SGVNPRE

Reason for why slower than LLVM GVNPRE

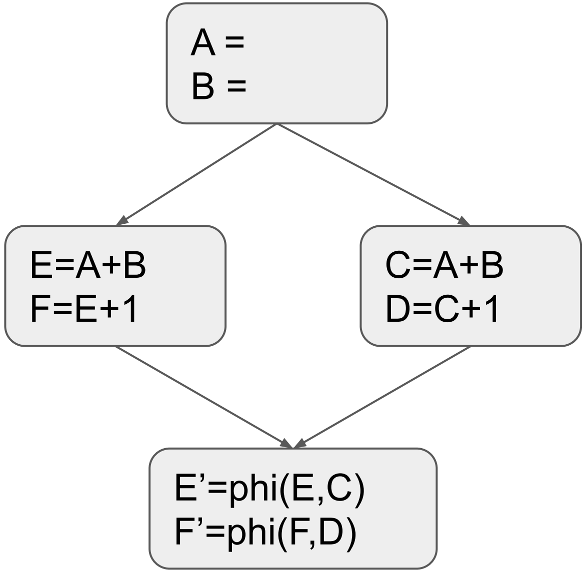
Reason 2: Phi node infinite blocking



Before SGVNPRE



After SGVNPRE



After LLVM gvnpre pass

Questions?