Value-Based Spectulative Partial Redundancy Elimination with Edge Profiling Information on LLVM

Yuanli Zhu, Yihao Huang, Jing Zhu, Yujian Liu

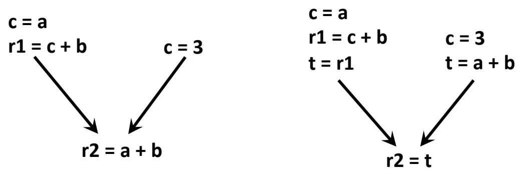
***Abstract*— Partial redundancy elimination (PRE) is a method that removes partial redundancies on some paths by hoisting the computation up on other paths. The transformation makes the computation fully redundant and therefore can be eliminated. Speculation-based PRE (SPRE) further uses profiling information to do more aggressive partial redundancy elimination. In this project, we implemented value-based speculative PRE (SGVNPRE) based on edge-profiling information. Compared with the main reference paper [2], which is solely based on lexical equivalence, we care about value equivalence and incorporated value-based PRE in our algorithm. We also make our algorithm work based on the SSA form, which are not implemented in the original paper. We further compare our result with the global value based PRE (GVNPRE) [9] to see if there is any speedup in terms of runtime. To the best of our knowledge, this comparison is not done in any published paper. We evaluate our algorithm on simple self-designed testcases as well as a subset of LLVM SingleSource Benchmarks. Results showed that for the limited testcases we tried, SGVNPRE is slower than GVNPRE and we provided potential explanations and solutions for this.**

I. INTRODUCTION

A. Motivation

Partial redundancy elimination (PRE) is a widely used optimization technique for removing redundant computation in the program. Existing work has explored different variants of PRE including SPRE and GVN-PRE [2], [9]. [2] proposed leveraging edge profiling information to efficiently find the optimal PRE. However, their algorithm is based on lexically equivalent expressions and not applicable to redundant computations that operate on the same values. Moreover, they did not compare their algorithm with other SPRE methods, leaving the effectiveness of their algorithm unclear. [9] presented a way to perform PRE on semantically equivalent values, but they only searched for fully redundant computations and missed many possible optimization opportunities. For example, the algorithm cannot completely eliminate loop invariant code [7], which is potentially a very important feature of PRE. This project aims to perform more aggressive partial redundancy elimination by taking advantage of edge profiling information, which guarantees the computation times of each value is optimal [2]. We call it Speculation-based GVNPRE (SGVNPRE).

Our SGVNPRE algorithm is evaluated on two carefully designed testcases as well as a subset of LLVM SingleSource Benchmarks. Results showed that, our SGVNPRE is fatser than not doing PRE, but slower than GVNPRE pass built in LLVM and we provided two potential solutions for this problem.



(a) before optimization (b) after optimization

Fig. 1: Example of a missed opportunity for expression-based PRE.

B. Related Work

1) Partial Redundancy Elimination: Partial redundancy elimination (PRE) is an optimization technique that removes the partial redundancies along some paths of the program. The main idea of this technique is to insert the same computation on irredundant paths and make it fully redundant. Existing methods can then be applied to remove redundancies. PRE has become an important component in global optimizers[1], [3].

2) Value-Based Partial Redundancy Elimination: Classic PRE searches for lexically equivalent expressions (i.e., expressions with the same operands). However, many expressions perform the same underlying computation despite they have different operands. For example, in Figure 1a, the computation $r1 = c + b$ and $r2 = a + b$ are redundant, but classic expression-based PRE can not find such redundancy. GVNPRE is therefore proposed to tackle this limitation [9]. Concretely, it keeps track of the underlying value of an expression and finds redundancies using values. As a result, in Figure 1b, GVNPRE removes the partial redundancy.

3) Speculation-Based Partial Redundancy Elimination: Another limitation of classic PRE is that it requires the insertion to not increase the total computation, limiting insertion points to be fully anticipated for the computation. This conservative searching thus misses some optimization opportunities [5]. A simple example of a missed opportunity for classic PRE is shown in 2. In the figure, the underlined numbers written alongside the control flow edges represent frequencies of execution. If the expression $a + b$ were to be inserted into block 2 (or were to be inserted on the edges 1-2 or 2-4), the computation of $a + b$ in block 6 would become fully redundant and could be deleted. However, none of the classic PRE formulations performs such a code transformation to be best of our knowledge [5]. There are two reasons for this. First, classic PRE algorithms do not have the runtime information (e.g. execution frequency information in this case). Without the actual runtime information, it's possible that the insertion of a new computation on an edge might increase the total number of evaluations of the expression. Most optimizers ended up not taking such risks. The second reason is that an optimizer must be careful not to introduce side-effects into a program that did not exhibit them before. If we were to insert the expression a/b in block 3 of the flowgraph of Figure 2 and if division by zero could cause an error interrupt on the target computer, then there is a risk that we have just caused the program to fail. But if we have their execution frequencies available and if we know that that expression cannot cause an exception, we can perform code transformations missed by classic PRE methods. And this is the central idea of Speculation-based Partial Redundancy Elimination (SPRE).

There are mainly two ways of doing SPRE, the first one is to use the edge profiling information to determine the profitability of using speculation [2], [5], the second one is to use path profiling information[4]. Path profiling information is more expensive that the edge profiling information.

C. Challenge

This project is built on the top of GVNPRE pass in LLVM. We need to read and understand the code more than two thousand lines. In addition, there is little source code we can reuse except global value numbering part and available expression set calculation. For the remaining part, we implement MC-PRE [2] to get calculation place for each value. Related code cannot be found online to the best of our knowledge. In addition, we also need to modify the algorithm to make it work in our project. First, the original algorithm only cares about lexical equivalence instead of value equivalence. Second, we need to design an algorithm to make it work in SSA form. We get some ideas from SSAPRE algorithms, which performs partial redundancy elimination based entirely on SSA form [7]. Thirdly, the algorithm only cares about the optimal computation times of a single value instead of all expressions in the program. Finally, there are many assumptions in the paper that is impractical in the real world and we are required to fill in the details by ourselves.

II. ALGORITHM

The overall algorithm of SGVNPRE is presented in algorithm 1. First, for each expression, we assign it a number and keep it in a value numbering table (VN and VNR). VN can map from expression to number and VNR can map from number to an array of expression. The expressions with the same value will be assigned to the same number. We then need to calculate the partial anticipated expression set at the entry point (PANT_IN) and available expression set (AVAI_OUT) at the exit block of each block.

ISTMAP maps from each edge to a set of value numbers that will be inserted on this edge. To construct this map, we apply MC-PRE [2] to each value number. It is done by first finding essential edges according to PANT_IN and AVAI_OUT and then build a single-source, single-sink reduced flow graph according to essential edges. Finally, we perform a minimum cut to the flow graph and get the point where expression can be computed in least times.

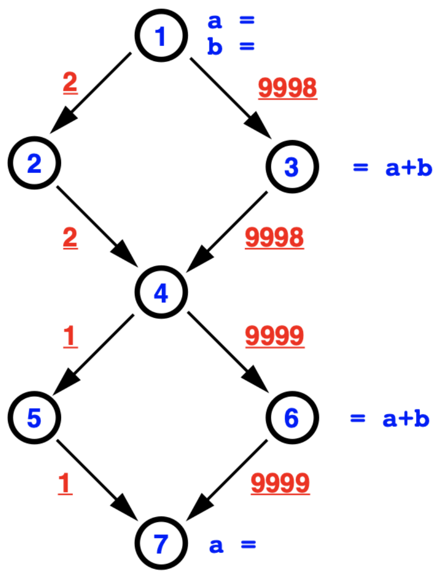


Fig. 2: Example of a missed opportunity for classic PRE [5]

Next, at each insertion point, we add a new block on that edge and insert the expression of the value number we got from ISTMAP. Then, we replace old use of these values with the new ones we inserted. In this step, we also need to add additional phi nodes and keep SSA form valid. Lastly, **clean()** function do dead code elimination to eliminate the old definitions of the values and also eliminate some edges whose source block has only one successor and destination block has only one predecessor. We need to do it because we split too many edges in last step.

```
Data: OrigF (input IR), PF (Profiling Data)
Result: OptF (Optimized IR after SGVNPRE)
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 SGVNPRE.

Following parts talk about the detailed algorithm of some main functions.

A. Value Numbering

First, we follow [9] on mapping expressions to values using a value table (a hash table) VN. Each value then contains a set of expressions that are evaluated to be the same. We use the value of each operand in the expression to define the expression's value. For example, if v_1 contains r_1 and r_2, and v_3 contains r_3, then we do not need to consider r_3+r_1 and r_3+r_2 separately because both of them can be represented as v_3+v_1. Note that this mapping function is flexible and can be as smart as possible. For example, it can consider the algebraic properties of operations, so that if v_1 contains 1, v_2 contains 2, v_4 contains v_1+v_3 for some v_3, and v_5 contains v_4+1, then v_5 should also contains v_2+v_3.

B. Build Sets

In this step, we will construct several sets that are necessary to construct a reduced graph, from which we will then derive the single-source, single-sink graph. The algorithm is modified from [9]. We first define three gen sets for any basic block b, EXP_GEN(b) for expressions that appear in the right hand sides of an instruction in b, PHI_GEN(b) for temporaries that are defined by a phi node in b, and TMP_GEN(b) for temporaries that are defined by non-phi instructions in b. To obtain AVAI_OUT, we conduct a standard forward availability analysis as the following:

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

[9] builds anticipation set for each block but here we change it to partial anticipation. An expression e is partial anticipable at a program point p if e will be computed along at least one path from p to p_{end}, while qn expression e is anticipable at a program point p if e will be computed along all paths from p to p_{end}, and no variable in e is redefined until its computation. This is one of the main differences between PRE and SPRE. PRE will only consider all paths in case inserting unexecuted instructions which is the reason why optimization of a+b is missed in figure 2.

We could find out PANT_IN by conducting a backward partial anticipability analysis as the following:

$$\begin{aligned}\text{PANT\_OUT}(b) &= \begin{cases}\bigvee_{m\in succ(n)}\text{PANT\_IN}(m) & \text{if } |\text{succ}(b)|>1\\[1em] \text{phi\_translate}(A[\text{succ}(b)],b,\text{succ}(b)) & \\ \qquad\qquad\qquad\qquad \text{if } |\text{succ}(b)|=1\end{cases}\\ \text{PANT\_IN}(b) &= \text{PANT\_OUT}(b)\cup\text{EXP\_GEN}(b)\\ &\quad \cap(\neg\,\text{TMP\_GEN}(b))\end{aligned}$$

where **phi_translate** is a procedure that translates the value into the legal value for the predecessor block given a value, value's parent block and the predecessor of its parent. For example, if t1 is anticipated by block b, and block b has a phi which defines t1 and has t2 as an operand from block c, then t2, rather than t1, is anticipated at the end of block c, and it has a different value. In addition, **phi_translate** is done recursively to each operand of the value to be translated.

C. Find the Optimal Insertion Map

This step is to build the optimal insertion Map ISTMAP which returns which value numbers are needed given an edge. This is done by iterating all value number in VNR, and for each value number, we perform MC-PRE [2] as shown below.

1) Find Essential Edges: We first find the set of essential edges for insertion. The essentiality of an edge depends on two definitions: insertion redundant and insertion useless. An edge (u,v) from basic block u to basic block v is insertion redundant for a value V if V is available at the exit of u. An edge (u,v) is insertion useless for a value V if V is not anticipated at the entry of v. An edge is then essential if it is neither insertion redundant nor insertion useless. Formally, edge (u,v) is essential for value number n if n is in PANT_IN[v] but not in AVAI_OUT[u].

2) Construct a Single-Source, Single-Sink Reduced Flow Graph: Given the essential edges, we construct a reduced flow graph where we remove all essential edges and nodes that do not have an essential incident edge. The weight of the edge is the execution times of that edge according to the profiling data. The reduced graph usually contains multiple source nodes and sink nodes, we then insert a pseudo entry and exit node to convert the graph into a single-source, single-sink graph G_{st}. The inserted associating edges will have weight ∞.

3) Apply Minimum-Cut: Finally, with the graph G_{st}, the optimal insertion set I_π will be a minimal cut on the graph. For the proof of the correctness and optimality of I_π, please refer to the original paper [2]. For our implementation, we choose Ford-Fulkerson Algorithm [6].

D. Insert and Replace Expressions

1) Insert New Definition: From last step, we get ISTMAP from which we know what values should be inserted at each edge. For each value of each edge, we need to choose appropriate expression of the value number and insert the expression at that edge. Noticed that there may be multiple expressions for each value number in VNR but not all expressions can be used. If the expression uses any definition which is not available at that point, the expression cannot be chosen. Luckily, according to definition, it guarantees that at least one expression uses definitions that are all available. Recall that we will only insert value on an essential edge, which means the value is in PANT_IN of the destination block but not in AVAI_OUT of the source block. In addition, if the value is in PANT_IN of the destination block, it means an expression with this value is partial anticipated and the definition it uses is not killed according to the way we build PANT_IN. Therefore, we will at least find one expression can be safely computed for that value. We store all the expressions into new value numbering map **newVNmap**, which maps from the value number to an array of expressions we newly created and inserted.

2) Insert Phi Node: This step is done with the idea of SSA construction mentioned in the lecture [8]. We modify it slightly as shown in algorithm 2. Noticed that here we do not put any phi node at the dominance frontier but only the block that anticipates the value, because the original algorithm assumes there is an initial value for each variable so phi node will always be complete, which is not our case. There may not be a same value from another branch, which makes phi node invalid. However, if the value is anticipated, then minimum cut algorithm guarantees all paths from the entry to the block will have a newly inserted value, so phi node is guaranteed to be complete.

```
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
```
Algorithm 2: Inserting phi node for inserted definition.

3) Replacing variables: For each value, we have got all its new definitions in newVNmap. Now, all old definitions of those values are useless and we need to replace the places where they are used. Algorithm 3 shows how to choose appropriate definition to replace the old use with the same value number, which is also a modification version of the renaming algorithm on the lecture slide [8]. The first difference is that we maintain one stack per value number instead of per variable and we also do not need to create new name for a definition since the newly inserted definitions are already in SSA form.

Secondly, since not all phi nodes are inserted during last step and some exist in OrigF, we need to deal with original phi nodes as well. When those phi nodes use old definitions that can be replaced, it should be processed similar to the new inserted phi nodes instead of treating it as a use of a definition like other operations. However, unlike newly inserted phi nodes, instead of adding incoming definition, we replace the 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
```
Algorithm 3: Replacing old definitions with new ones.

III. IMPLEMENTATION

We implement and test our algorithms on LLVM with five passes. The first pass is -mem2reg, which promotes memory references to register references. We perform this pass because if variables are allocated in the memory and loaded while using them, then we cannot safely guarantee whether a variable is still the same when it is loaded from memory. Since we need global value numbering to identify as many same values as possible, we need -mem2reg pass.

The second pass is data profiling pass, which provides execution frequency of each block and probability of each branch.

The third pass is -sgvnpre pass, which is the core of our algorithm. It is implemented based on the source code of llvm 2.6 -gvnpre pass, which is an implementation of [9]. However, it is a very old version of llvm so some operators in IR, such as fmul, are not supported to do global value numbering and will report error, which limits our test cases. The value numbering table constructing and available out sets building is almost not changed but we changed the anticipation sets building to partial anticipation sets building. The rest of the algorithm is almost completely implemented by ourselves. We referenced the code in GeeksforGeeks of the Ford-Fulkerson Algorithm.

The forth and fifth passes are used for **clean()** function. The forth pass is the dead code elimination pass, which we directly call -dce transform pass built in LLVM. The fifth pass is block merging pass to merge consecutive blocks withe edges whose source block has only one successor and destination block has only one predecessor. It is also a straightforward pass implemented by ourselves.

In the following evaluation part, when measuring runtime of LLVM gvnpre pass, it also needs to first go through mem2reg pass and finally dce and block merging pass to ensure fairness.

IV. EVALUATION

A. *Case study for two simple testcases*

The main goal of this part is to evaluate the correctness of our Speculation-based GVNPRE in two simple testcases. We carefully designed two testcases where the original GVNPRE will miss the optimization opportunities but Speculation-based GVNPRE should be able to optimize.

The first test case is as follows. This test case is similar to the CFG in Fig.2.

```
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);
        }

    }

}
```

The corresponding CFG is shown in Fig. 3. The left part of the figure is the CFG for SGVNPRE and the right part of the figure is the CFG for GVNPRE. The red box indicates the expression that is inserted specially for SGVNPRE compared with the original GVNPRE. While the orange box indicates the expression hoisted by SGVNPRE while it still exists in the original GVNPRE.

We can see that in the SGVNPRE, we are able to hoist the $a+b$ expression in the second right branch by adding the $a+b$ expression in the top left branch. And it is beneficial since there is only 1 out of 100 chance that you will execute the top left branch while there are 90 out of 100 chance you will execute the second right branch.

The second test case is as follows.

```
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);

    }

}
```

The corresponding CFG is shown in Fig. 4. The left part of the figure is the CFG for SGVNPRE and the right part of the figure is the CFG for GVNPRE. The red box indicates the expression that is inserted specially for SGVNPRE compared with the original GVNPRE. While the orange box indicates the expression hoisted by SGVNPRE while it still exists in the original GVNPRE.

We can see that in the SGVNPRE, we are able to hoist the $a+b$ expression that is originally in the right branch to the entry block while in the case of GVNPRE, it does not do this. It's clear that this optimization is beneficial because you need to compute $a+b$ at the end of the basic block no matter which branch you take.

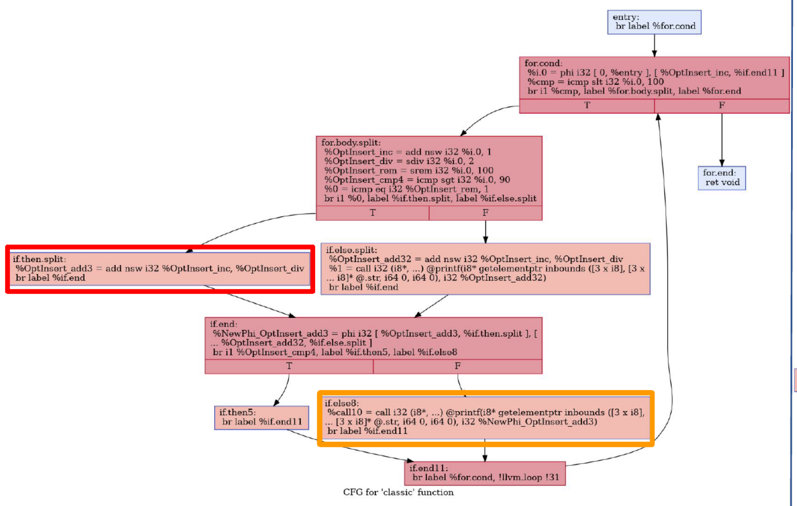
B. *LLVM SingleSource Benchmark*

We further evaluate our algorithm on the LLVM SingleSource Benchmarks for real-life runtime performances. Due to the limit of time and some bugs in our program, we choose 3 testcases from the LLVM SingleSource benchmark for evaluation: bubblesort, quicksort, permutation. The testcases we choose includes the sorting and permutation algorithms that compilers tend to run a lot in real life as well as hard backtracking problems that are computationally expensive.

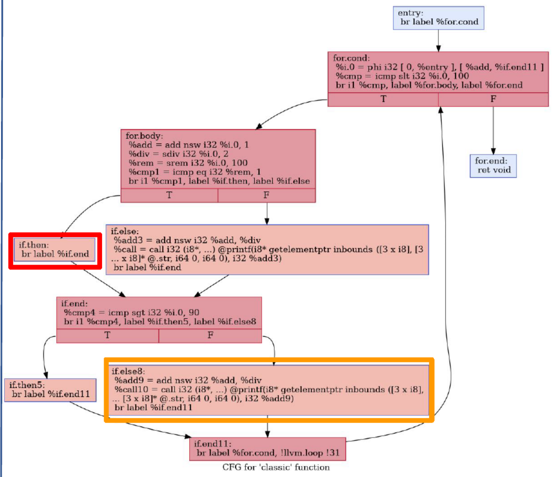
We compare SGVNPRE optimized program with both non-optimized program and LLVM -gvn -enable-pre optimized program. They all go through reg2mem pass, dead code elimination pass and merge block pass to ensure fairness.The result is shown in Fig. 5.

We can see that SGVNPRE optimized program is faster than non-optimized program as expected. However, it is slower than GVNPRE implemented in LLVM. After comparison of the control flow graph, we found that SGVNPRE can only deal with instructions that their operands are not optimized in SGVNPRE. A simple example is shown in Fig. 6. Although value table is able to recognize C and E, D and F has the same value and both pairs are partial redundant, in the first round of SGVNPRE, it will only eliminate the partial redundancy of C and E. The reason is that when we build our PANT_IN set, we use a backward analysis that if a definition A depends on a definition B, the anticipation of A will be killed at B. It results in the computation of A will not exist on the essential edge so it will not be optimized until the second round optimization of SGVNPRE. We implemented our algorithm of building PANT_IN based on MC-PRE [2], which focuses on optimizing the execution times of one computation without moving around other instructions.

Case study: classic


```
entry:
 br label %for.cond

for.cond:
 %i.0 = phi i32 [ 0, %entry ], [ %OptInsert_inc, %if.end11 ]
 %cmp = icmp slt i32 %i.0, 100
 br i1 %cmp, label %for.body.split, label %for.end
 T | F

for.end:
 ret void

for.body.split:
 %OptInsert_inc = add nsw i32 %i.0, 1
 %OptInsert_div = sdiv i32 %i.0, 2
 %OptInsert_rem = srem i32 %i.0, 100
 %OptInsert_cmp4 = icmp sgt i32 %i.0, 90
 %0 = icmp eq i32 %OptInsert_rem, 1
 br i1 %0, label %if.then.split, label %if.else.split
 T | F

if.then.split:
 %OptInsert_add3 = add nsw i32 %OptInsert_inc, %OptInsert_div
 br label %if.end

if.else.split:
 %OptInsert_add32 = add nsw i32 %OptInsert_inc, %OptInsert_div
 %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x
... i8]* @.str, i64 0, i64 0), i32 %OptInsert_add32)
 br label %if.end

if.end:
 %NewPhi_OptInsert_add3 = phi i32 [ %OptInsert_add3, %if.then.split ], [
... %OptInsert_add32, %if.else.split ]
 br i1 %OptInsert_cmp4, label %if.then5, label %if.else8
 T | F

if.then5:
 br label %if.end11

if.else8:
 %call10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
... [3 x i8]* @.str, i64 0, i64 0), i32 %NewPhi_OptInsert_add3)
 br label %if.end11

if.end11:
 br label %for.cond, !llvm.loop !31
```
CFG for 'classic' function

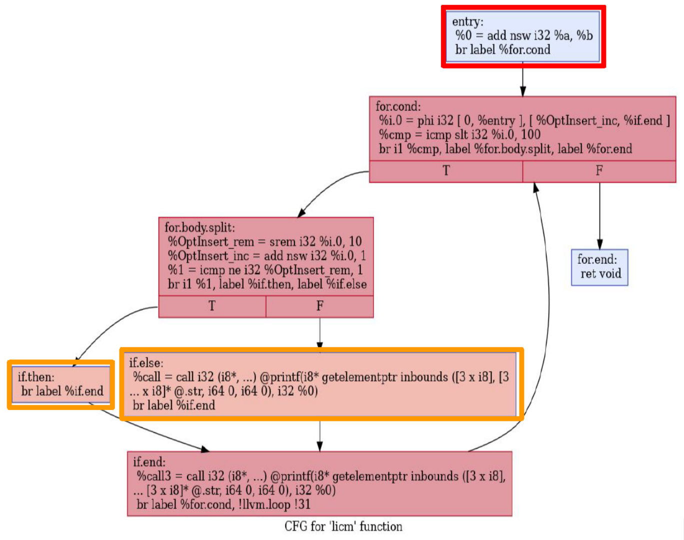
SPGVNPRE


```
entry:
 br label %for.cond

for.cond:
 %i.0 = phi i32 [ 0, %entry ], [ %add, %if.end11 ]
 %cmp = icmp slt i32 %i.0, 100
 br i1 %cmp, label %for.body, label %for.end
 T | F

for.end:
 ret void

for.body:
 %add = add nsw i32 %i.0, 1
 %div = sdiv i32 %i.0, 2
 %rem = srem i32 %i.0, 100
 %cmp1 = icmp eq i32 %rem, 1
 br i1 %cmp1, label %if.then, label %if.else
 T | F

if.then:
 br label %if.end

if.else:
 %add3 = add nsw i32 %add, %div
 %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3
... x i8]* @.str, i64 0, i64 0), i32 %add3)
 br label %if.end

if.end:
 %cmp4 = icmp sgt i32 %i.0, 90
 br i1 %cmp4, label %if.then5, label %if.else8
 T | F

if.then5:
 br label %if.end11

if.else8:
 %add9 = add nsw i32 %add, %div
 %call10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
... [3 x i8]* @.str, i64 0, i64 0), i32 %add9)
 br label %if.end11

if.end11:
 br label %for.cond, !llvm.loop !31
```
CFG for 'classic' function

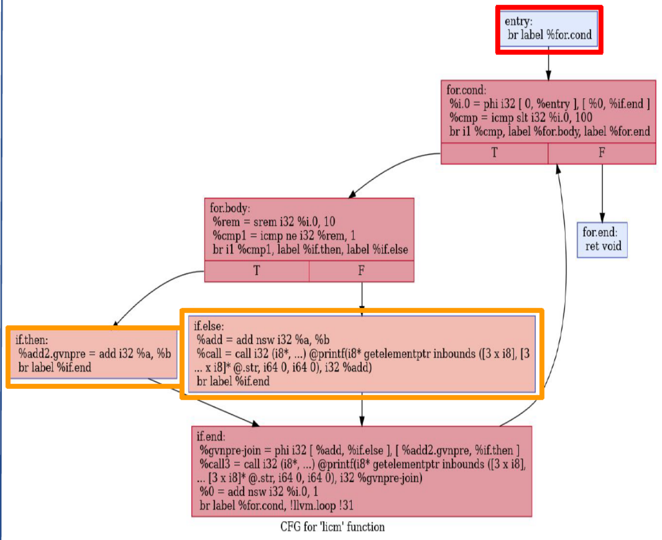
GVNPRE

Fig. 3: CFGs for SGVNPRE and GVNPRE for the first testcase

Case study: licm


```
entry:
 %0 = add nsw i32 %a, %b
 br label %for.cond

for.cond:
 %i.0 = phi i32 [ 0, %entry ], [ %OptInsert_inc, %if.end ]
 %cmp = icmp slt i32 %i.0, 100
 br i1 %cmp, label %for.body.split, label %for.end
 T | F

for.end:
 ret void

for.body.split:
 %OptInsert_rem = srem i32 %i.0, 10
 %OptInsert_inc = add nsw i32 %i.0, 1
 %1 = icmp ne i32 %OptInsert_rem, 1
 br i1 %1, label %if.then, label %if.else
 T | F

if.then:
 br label %if.end

if.else:
 %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3
... x i8]* @.str, i64 0, i64 0), i32 %0)
 br label %if.end

if.end:
 %call3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
... [3 x i8]* @.str, i64 0, i64 0), i32 %0)
 br label %for.cond, !llvm.loop !31
```
CFG for 'licm' function

SPGVNPRE


```
entry:
 br label %for.cond

for.cond:
 %i.0 = phi i32 [ 0, %entry ], [ %0, %if.end ]
 %cmp = icmp slt i32 %i.0, 100
 br i1 %cmp, label %for.body, label %for.end
 T | F

for.end:
 ret void

for.body:
 %rem = srem i32 %i.0, 10
 %cmp1 = icmp ne i32 %rem, 1
 br i1 %cmp1, label %if.then, label %if.else
 T | F

if.then:
 %add2.gvnpre = add i32 %a, %b
 br label %if.end

if.else:
 %add = add nsw i32 %a, %b
 %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3
... x i8]* @.str, i64 0, i64 0), i32 %add)
 br label %if.end

if.end:
 %gvnpre-join = phi i32 [ %add, %if.else ], [ %add2.gvnpre, %if.then ]
 %call3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
... [3 x i8]* @.str, i64 0, i64 0), i32 %gvnpre-join)
 %0 = add nsw i32 %i.0, 1
 br label %for.cond, !llvm.loop !31
```
CFG for 'licm' function

GVNPRE

Fig. 4: CFGs for SGVNPRE and GVNPRE for the second testcase

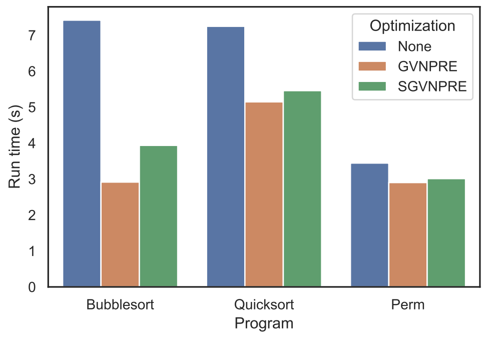


Fig. 5: Run time of different optimization methods on SingleSource Benchmark.

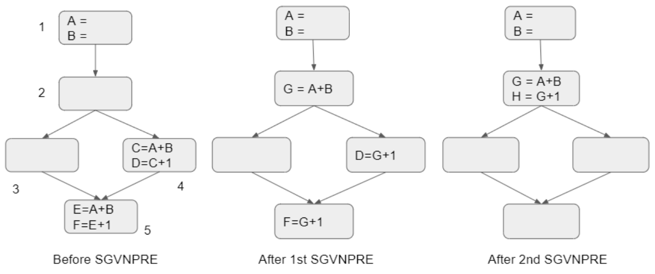


Fig. 6: Example of SGVNPRE fails to optimize an expression computation.

This problem cannot simply be addressed by running multiple times of SGVNPRE. One important reason is the phi node blocking issue as an example shown in Fig. 7. The example is similar to Fig. 6 but it does not have an edge to be cut at the entry. Then, according to minimum cut algorithm, it will insert A+B at the left branch instead of at the entry. In this case, there is a phi node E' at the merging point. However, since F will always depend on E' and E' cannot be moved, the partial redundancy of F can never be eliminated no matter how many times we run SGVNPRE. However, the GVNPRE pass in LLVM is able to eliminate this problem. The original SPRE algorithm [2] we referenced is not designed for SSA form so the algorithm needs to be improved further for this issue.

There are two possible solutions to this problem. One straight forward idea is to think of an algorithm to address the phi node blocking issue and run the pass multiple times until no change happens. However, it could not be as easy as it sounds because we will always compute the optimal position for all value numbers and insert them at the right place even they are already at the optimal position. Therefore, how to judge whether a change has happened remains a problem. The second way is to think of a better way to build PANT_IN set. The LLVM GVNPRE pass utilizes this idea. It does not run pass multiple times but only one time and successfully solving the dependency problem. However, it builds its full anticipation set in a way that even the operand is defined, the expression is not killed and still can be anticipated. However, their insertion algorithm is also different because of this reason.

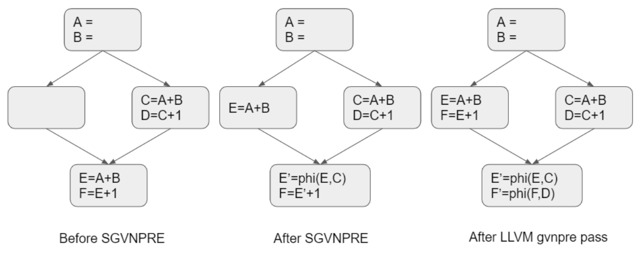


Fig. 7: Phi node blocking issue of SGVNPRE.

V. CONCLUSION

In this project, we proposed and implemented a new Speculation-based Global Value Numbering Partial Redundancy Elimination(SGVNPRE) method for code optimization. Our methods are implemented in the form of LLVM passes and are based on the SSA forms that LLVM relies upon. We carefully designed two testcases as case studies to show the correctness of our algorithm. Comparing our method with respect to the original GVNPRE, our method inserts extra expressions on the infrequently executed branch and hoists expressions that are executed frequency in the frequent branches for both testcases. Evaluating our method on the LLVM SingleSource benchmark shows that our SGVNPRE is slower than the original GVNPRE pass in LLVM in general and the two potential solutions for this are running multiple passes with phi node solving and building better partial anticipation sets.

REFERENCES

[1] Preston Briggs and Keith D Cooper. Effective partial redundancy elimination. *ACM SIGPLAN Notices*, 29(6):159–170, 1994.

[2] Qiong Cai and Jingling Xue. Optimal and efficient speculation-based partial redundancy elimination. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 91–102. IEEE, 2003.

[3] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[4] Rajiv Gupta, David A Berson, and Jesse Zhixi Fang. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No. 98CB36225)*, pages 230–239. IEEE, 1998.

[5] R Nigel Horspool and HC Ho. Partial redundancy elimination driven by a cost-benefit analysis. In *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*, pages 111–118. IEEE, 1997.

[6] L.R.Ford JR and D.R.Fulkerson. Maximal flow through a network. *Canadian Journall of Mathematics*, 8:399–404, 1955.

[7] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in ssa form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):627–676, 1999.

[8] T. Harvey P. Briggs, K. Cooper and L. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 22(8):859–891, 1998.

[9] Thomas VanDrunen and Antony L Hosking. Value-based partial redundancy elimination. In *International Conference on Compiler Construction*, pages 167–184. Springer, 2004.