Google Summer of Code 2024 Report

Reviving NewGVN

Manuel Brito

1 Introduction

Global Value Numbering (GVN) is an optimization technique that identifies and eliminates redundant code. This is achieved by assigning value numbers to each instruction, ensuring that instructions with the same value number are treated as equivalent.

```
int f(int x, int y) {
  // VN(x) = VN0, VN(y) = VN1
  int a, b, c, d;
  a = x - y; // VN0 - VN1 = VN2
  b = x + x; // VN0 + VN0 = VN3
  if (x == y) {
    // inside the if VN(y) = VN(x) = VN0
    // VN(a) = 0
    c = y + a; // VN0 - 0 =  VN0
    d = c * 2; // VN0 * 2 =  VN3
  } else {
    d = x - y; // VN0 - VN1 = VN2
  }
  return d;
}
```

```
int f_opt(int x, int y) {
  int a, b, d;
  a = x - y;
  b = x + x;
  if (x == y) {
    d = b;
  } else {
    d = a;
  }
  return d;
}
```

Figure 1: Basic example of value numbering with a C function

There are two main approaches to GVN, categorized as pessimistic or optimistic, based on the set of initial assumptions [1, 2]. The pessimistic view is conservative, assuming that the entire program is reachable and that no expressions are redundant. In contrast, the optimistic view makes some unsafe assumptions, presuming only the entry basic block is reachable and that all values are redundant.

1.1 GVN in LLVM

LLVM's optimizer includes two implementations of global value numbering: `GVN`[1] and `NewGVN`.[2] `GVN` is a pessimistic hash-based algorithm that iterates through all basic blocks of the function using reverse post order (RPO) until a fixed point is reached. It is combined with partial redundancy elimination (PRE) in a naive manner, where the PRE algorithm is executed after the value numbering process reaches a fixed point. To distinguish this implementation from others in LLVM, we refer to it as `GVNPRE`.

In contrast, `NewGVN` is an optimistic algorithm based on algorithms developed by Gargi [2]. Introduced in LLVM in 2016, it aimed to address the limitations of `GVNPRE`, such as long compile times and the absence of value numbering for memory operations, among other issues.[3]

[1] https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/GVN.cpp
[2] https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/NewGVN.cpp
[3] https://lists.llvm.org/pipermail/llvm-dev/2016-November/107110.html

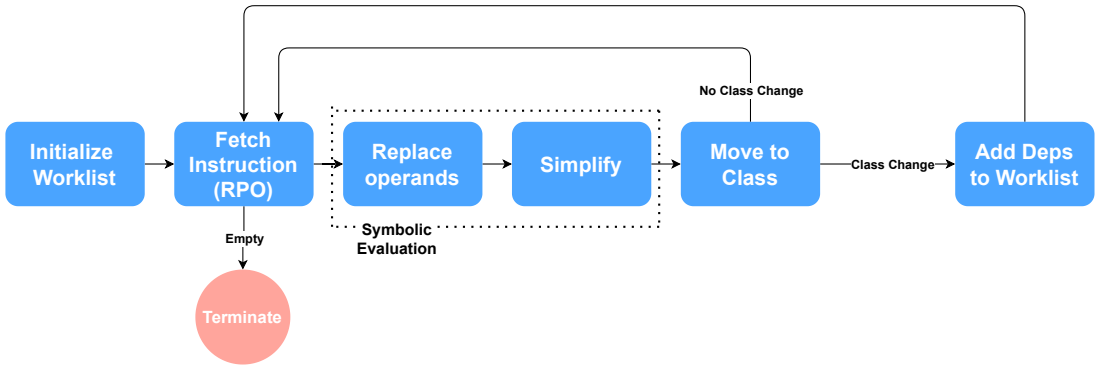

Figure 2: Overview of NewGVN's analysis phase

```
define i32 @optimistic(i32 %x, i32 %y){
entry:
    br label %loop

loop:
    %a = phi i32 [0, %entry], [%a.i, %loop]
    %b = phi i32 [0, %entry], [%b.i, %loop]
    %c = xor i32 %x, %a
    %a.i = sub i32 %x, %c
    %b.i = add i32 %b, 1
    %cmp = icmp ne i32 %b.i, %y
    br i1 %cmp,  label %loop,label %exit

exit:
    ret i32 %a
}
```

```
define i32 @optimistic(i32 %x, i32 %y) {
entry:
  br label %loop

loop:
  %b = phi i32 [ 0, %entry ], [ %b.i, %loop ]
  %b.i = add i32 %b, 1
  %cmp = icmp ne i32 %b.i, %y
  br i1 %cmp, label %loop, label %exit

exit:
  ret i32 0
}
```

Figure 3: Example where GVNPRE, as a pessimistic algorithm, fails to recognize that %a is a loop invariant. In contrast, NewGVN—shown on the right—employs an optimistic assumption and successfully identifies it.

An overview of NewGVN's execution is presented in Figure 2. It represents its state through a set of congruence classes, each identified by a defining expression. A fixed point is computed using a semi-naive algorithm where a worklist of instructions is maintained. As an optimistic algorithm, the worklist begins with instructions from the entry block only. Instructions are then retrieved from the worklist using RPO. Processing an instruction involves symbolically evaluating it to produce its defining expression, after which the instruction is moved to the corresponding congruence class. Finally, if changes occur in NewGVN's state, all dependencies of the instruction are added to the worklist.

1.2 Why NewGVN?

Both implementations utilize InstructionSimplify to conduct algebraic simplification. However, NewGVN represents a significantly more ambitious approach to value numbering. NewGVN integrates optimistic global value numbering with algebraic simplification and the elimination of unreachable code. Additionally, it exploits Static Single Information (SSI) [3] and Memory SSA (MSSA) [4] extensions to SSA for a more precise analysis.

In Figure 3, we illustrate a transformation achievable by NewGVN but not by GVNPRE. Recognizing that a is loop-invariant requires assuming it from the outset, a capability lacking in GVNPRE due to its pessimistic nature.

We have established that `NewGVN` has the potential to be both a more powerful form of value numbering for the LLVM optimizer. This raises the question: why is it not enabled by default? There are two main reasons. Firstly, `NewGVN` is not stable, as it suffers from numerous bugs, particularly concerning termination and performing unsound transformations. Secondly, there is a lack of partial redundancy elimination(PRE) in `NewGVN`. Having a PRE stage is key to produce performant code [1].

1.3 Contributions

In this project we made the following contributions: implementation of PRE in NewGVN, implementation of fully optimistic algebraic simplification and solved 4 open issues.

2 Implementing PRE in NewGVN

Partial-redundancy elimination (PRE) [5,6] is a whole-function optimization that aims to remove expressions that are redundant in some but not all execution paths. The expression is made fully redundant by inserting equivalent expressions in the paths that are missing. Unifying it with GVN gives better results. For example in figure 4, we can only eliminate the partial redundancy if we can prove that the expression `b - y - 42` is equivalent to `x + 42`. This can be proven by value numbering.

```
b = x + y;
if (...) {
    a = x + 42;
    ...
} else {
    ...
}
a = b - y + 42;
```

```
b = x + y;
if (...) {
    a = x + 42;
    ...
} else {
    a = x + 42;
    ...
}
```

Figure 4: Example of eliminating a value-based partial redundancy.

In the remainder of this chapter, we described how a PRE stage was implemented in the existing NewGVN codebase.

2.1 *Phi-Of-Ops* Transformation

The *Phi-Of-Ops* transformation aims to improve the precision of NewGVN by performing the following:

$$\phi(a_1, a_2) \ \mathbf{op} \ \phi(b_1, b_2) \Rightarrow \phi(a_1 \ \mathbf{op} \ b_1, a_2 \ \mathbf{op} \ b_2)$$

This allows it to discover equivalences hidden behind phi instructions. Given some phi-dependent expression e at some basic block b, we start by translating e to each of the predecessors of b. Translating an expression to a predecessor consists of, for each of its operands: if it is a phi instruction that resides in basic block b, then the operand is replaced by the phi operand incoming from the predecessor. Otherwise, recursively translate the operands until block b is dominated.

In NewGVN, the expressions are not recursively translated. Instead, if one of the operands of the original instruction depends on a phi in block b, then the transformation is aborted. This check is performed by the `OpIsSafeForPHIOfOps` function. This is to avoid having to materialize multiple instructions to account for the phi-translated subexpressions.

After translation, check if the translated expression is available in the predecessor. This can be achieved by inserting it as a temporary instruction in block b and computing its value number. Then we examine the corresponding congruence class to find a value that dominates the predecessor. If the expression is available in every predecessor, then we can insert a phi expression composed of the path-available values, making the expression fully available.

This process is ilustrated in Figure 5.

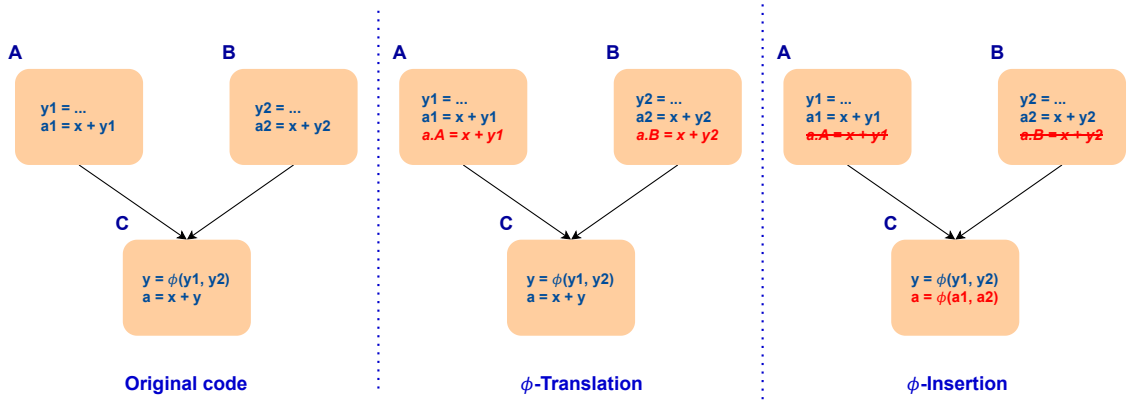

Figure 5: Performing the phi-of-ops transformation.

2.2 Generalizing *Phi-Of-Ops*

The *Phi-Of-Ops* transformation is essentially a special case of PRE, where the value always depends on a phi instruction and is available on all reaching paths. In order to generalize *Phi-Of-Ops* to PRE, we removed the restriction that the transformation only occurs for instructions that depend on phis. Also, we have to be able to insert instructions in predecessors where the value is not available. Instead of inserting, we simply use the temporary instructions used in the phi translation. It is important to note, however, that it is not always safe to use this temporary since it might be the case that not all of its operands are available in the predecessor.

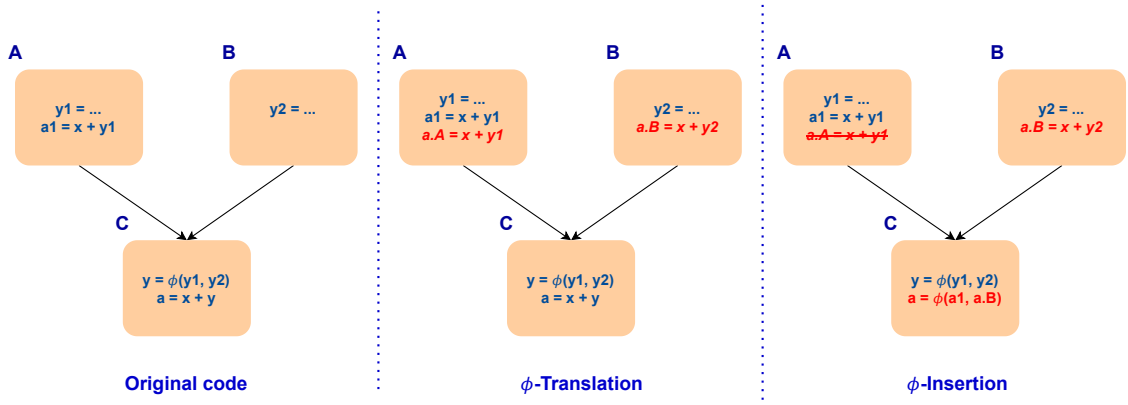

Figure 6: Performing the phi-of-ops transformation with an insertion. In predecessor B, the value is not available, so instead of removing the temporary instruction, we use it to make the value fully available.

To prevent increases in code size (ignoring phi instructions), PRE insertion is only performed if only one insertion is required to make the value fully available. `GVNPRE` uses the same approach.

Handling loads Performing this transformation for loads requires extra care due to MemorySSA. In LLVM, MemorySSA is not a native part of the IR; it is instead constructed on the side as an analysis. Originally, NewGVN did not perform *Phi-Of-Ops* for load instructions. To support this, during the phi translation, temporary memory uses are created for the temporary loads using the `MemorySSAUpdater`.

2.3 Loop Invariant Code Motion

Loop Invariant Code Motion (LICM) is a transformation that moves loop-invariant code out of loops. LICM is subsumed by PRE since the value is available from all backedges and the only predecessor where it is not available is in the loop pre-header.

In `GVNPRE`, LICM is only performed for loads, and it requires the `LoopInfo` analysis. In our implementation, LICM arises naturally from the interaction between the PRE algorithm described in the previous section and the optimistic assumption.

In essence, the PRE algorithm when combined with the optimistic assumption, allows us to speculate that the value in the first iteration of the loop will be the same as the value in any given iteration.

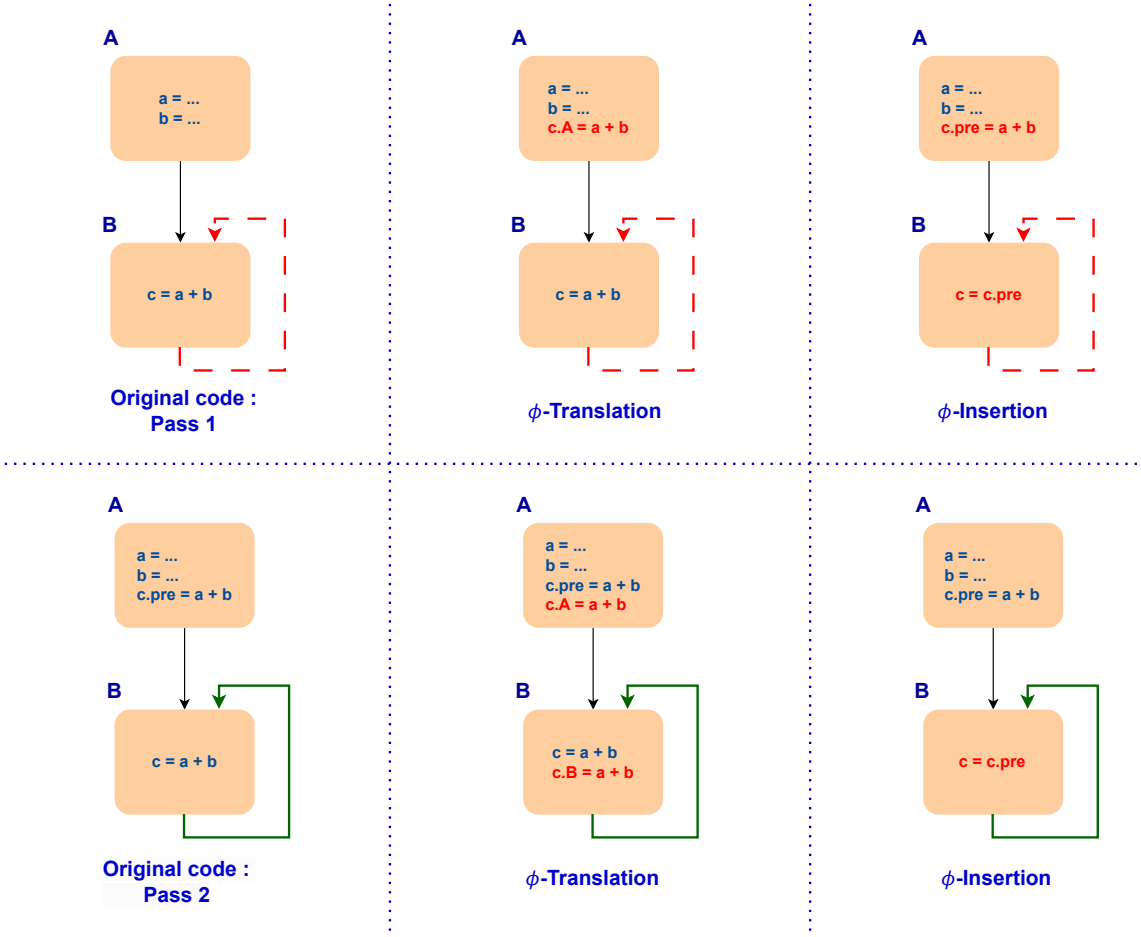

Figure 7: Performing LICM by using speculative PRE: In the first pass (top row), the backedge is assumed to be unreachable (●). In the second pass (bottom row), after discovering that the backedge is reachable (●), we have to corroborate the speculated PRE.

The process is ilustrated in Figure 7. When processing the instructions in block B for the first time, GVN does not know if the backedge is reachable. Since we are operating under an optimistic assumption, GVN will treat it as unreachable until proven otherwise. The first pass, where the backedge is assumed unreachable, is represented in the top row. When performing PRE for instruction `c`, we can skip the backedge. This leaves us with only the loop pre-header (block A) not having the value available, thus `c.pre` is inserted. In the second pass (bottom row), we know that the backedge is reachable and can no

longer ignore it. However, since the value is truly loop invariant, the inserted instruction `c.pre` will be equivalent to the phi-translated expressions, thus confirming the speculation performed in the previous pass.

2.4 Missing Features

In this section, we highlight some of the missing features in our implementation of PRE that are supported in `GVNPRE`.

Critical Edge Splitting In Figure 8 on the left, we have a partially redundant expression. In one of the execution paths, the expression is computed twice. In the middle, we have a naive application of PRE where the expression is moved to the predecessor block. However, this is problematic since we are now introducing redundant computation in other paths. The issue here is the edge highlighted in red. Its origin has multiple successors, and its target has multiple predecessors. This edge is known as a critical edge. To ensure that PRE does not introduce redundant computation, critical edges must be split. Splitting an edge consists of adding an intermediate basic block. The partially redundant expression is then moved to said basic block.

`GVNPRE` supports critical edge splitting, while our implementation conservatively bails when the required PRE insertion is over a critical edge.

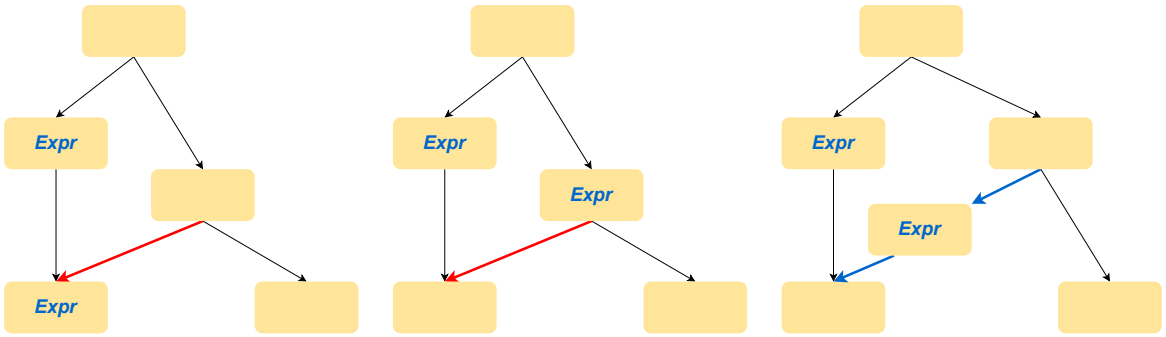

Figure 8: Example: PRE with critical edge splitting.

Load Coercion Load Coercion involves optimizing memory loads by utilizing previously loaded data to eliminate the need for new loads, even when the loads differ in type. This optimization applies when an earlier load encompasses all the memory required by a subsequent load. Note that this is not specific to PRE transformations but can also be used by the standard GVN algorithm.

For example, `load i64, ptr %p, align 4` subsumes `load i32, ptr %p, align 4`. To replace the smaller load, we have to coerce the bigger load to the correct type by truncating it. Figure 9 shows an example where, on the left, we have a partially redundant load that requires load coercion. On the right, the function after `GVNPRE`.

To support this in our implementation, we propose that all loads from the same pointer with the same MemorySSA operand, regardless of the loaded type, should be aggregated into an equivalence superclass (as illustrated in Figure 10).

Notice how a hierarchical structure arises, meaning a superclass made up of superclasses. For instance, to replace `load i32, ptr %p, align 4`, we could use any dominating load with the same memory and pointer, with at least the same size as `i32`. This means the load could have been an `i64` or even a load of a vector type such as `<2 x i16>`.

A similar mechanism exists in `NewGVN`, where loads and stores are placed in the same class. This allows it to perform dead store elimination and basic store-to-load forwarding.

Due to time constraints, we were not able to implement this feature.

```
define i32 @main(ptr %p, i1 %c) {
    br i1 %c, label %b1, label %b2
b1:
  %a.1 = load i64, ptr %p, align 4
  %u.1 = add i64 %a.1, %a.1
  br label %exit
b2:
  br label %exit

exit:
  %a = load i32, ptr %p, align 4
  ret i32 %a
}
```

```
define i32 @main(ptr %p, i1 %c) {
    br i1 %c, label %b1, label %b2
b1:
  %a.1 = load i64, ptr %p, align 4
  %u.1 = add i64 %a.1, %a.1
  %0 = trunc i64 %a.1 to i32
  br label %exit

b2:
  %a.pre = load i32, ptr %p, align 4
  br label %exit

exit:
  %a = phi i32 [ %a.pre, %b2 ], [ %0, %b1 ]
  ret i32 %a
}
```

Figure 9: Combining PRE with load coercion.

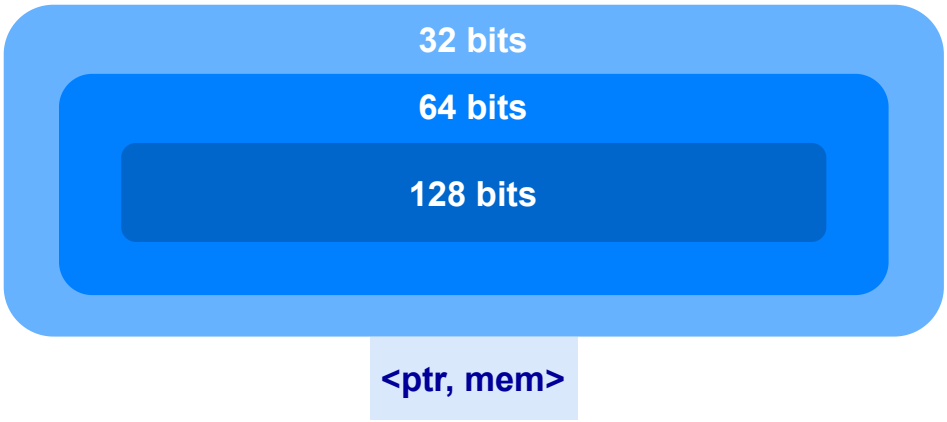

Figure 10: Load superclass.

3 Fully Optimistic Algebraic Simplification

Since `NewGVN` is an optimistic algorithm at any given point the congruence classes may not be sound. One consequence of this is that it is not allowed to eliminate redundancies during the analysis phase since these might be based on an incorrect assumption. `NewGVN` uses `InstructionSimplify` during the symbolic evaluation step to simplify expressions. Note, however, that `InstructionSimplify` operates over LLVM IR. For example, to simplify `%a = add i32 %x, 1`, it receives the opcode for the addition and the corresponding operands, which can be LLVM instructions, constants, or function arguments.

To cope with this, the defining expressions are `GVNExpressions` that mimic this structure. This is not problematic for the operands since, if we know that a given operand is equivalent to a constant, we can pass said constant to `InstructionSimplify`. The issue arises when simplification requires information from multiple levels of the expression tree. More specifically, if an operand of an operand is known to be equivalent to a constant, `NewGVN` is not able to pass this information to `InstructionSimplify`. In other words, beyond the first level of operands, there is a discrepancy between what `NewGVN` knows and what `InstructionSimplify` sees. This discrepancy results in fewer redundancies being discovered.

For instance, running `NewGVN` on the function in Figure 11 twice reveals additional equivalences that

a single run does not detect (refer to each output in Figure 12).[4] In the first NewGVN run, it discovers that %conv is equal to 1, implying that %bf.set is equal to or i32 1, %bf.clear, i.e., its defining expression. When performing symbolic evaluation on %bf.clear.1 = and i32 %bf.set, -131072, NewGVN passes or i32 %conv, %bf.clear as the representative for %bf.set, although in reality, we know that it is equivalent to or i32 1, %bf.clear. Consequently, InstructionSimplify is not able to simplify as much as it should.

```
define void @fn1(i1 %bc) {
entry:
    br label %for.cond

for.cond:
    %tmp = phi i1 [ 1, %entry ], [ 1, %for.cond ]
    %conv = zext i1 %tmp to i32
    %lv = load i32, i32* bitcast (i64* @a to i32*)
    %bf.clear = and i32 %lv, -131072
    %bf.set = or i32 %conv, %bf.clear
    %bf.clear.1 = and i32 %bf.set, -131072
    %bf.set.1 = or i32 1, %bf.clear.1
    br i1 %bc, label %for.cond, label %exit

exit: ; preds = %for.cond1
    store i32 %bf.set.1, i32* bitcast (i64* @a to i32*)
    ret void
}
```

Figure 11: Input program for which NewGVN does not reach a fixed point.

In the second NewGVN run, %conv has already been replaced with 1. Consequently, there is no discrepancy between the defining expression (what NewGVN knows) and the class leader (what InstructionSimplify sees). This results in the discovery of more redundancies. In this second run, NewGVN discovers that %bf.clear and %bf.clear.1 are equivalent and consequently that %bf.set and %bf.set.1 are also equivalent.

```
define void @fn1(i1 %bc) {
  br label %for.cond

for.cond:
  %lv = load i32, i32* @a, align 4
  %bf.clear = and i32 %lv, -131072
  %bf.set = or i32 1, %bf.clear
  %bf.clear.1 = and i32 %bf.set, -131072
  %bf.set.1 = or i32 1, %bf.clear.1
  br i1 %bc, label %for.cond, label %exit

exit:
  store i32 %bf.set.1, i32* @a, align 4
  ret void
}
```

```
define void @fn1(i1 %bc) {
  br label %for.cond

for.cond:
  %lv = load i32, i32* @a, align 4
  %bf.clear = and i32 %lv, -131072
  %bf.set = or i32 1, %bf.clear
  br i1 %bc, label %for.cond, label %exit

exit:
  store i32 %bf.set, i32* @a, align 4
  ret void
}
```

Figure 12: On the left, the output when running NewGVN once on the program from Figure 11. On the right, the output when running NewGVN a second time.

[4] https://github.com/llvm/llvm-project/issues/38031

On the other hand, since `GVNPRE` is pessimistic, the discovered redundancies are always sound. Therefore, unlike `NewGVN`, it is free to replace instructions as it discovers redundancies. Thus, the aforementioned issues do not occur. This means that even for acyclic code, where the optimistic assumption makes no difference, `GVNPRE` might be able to simplify more aggressively than `NewGVN`.

3.1 IR Checkpointing

To address this issue, we implemented a checkpointing mechanism for `NewGVN`, allowing us to update the IR on the fly with the optimistically discovered redundancies and then roll back the changes if necessary. To simplify the implementation, we changed the worklist fixpoint to a naive fixpoint, meaning the entire function is reprocessed in each pass. This is because tracking dependencies becomes much harder when arbitrary changes can be made to the function. This is an issue we wish to revisit in the future.

When an instruction changes from one congruence class to another, the uses of the instruction are replaced by the class leader (`UpdateIR`). However, before making the replacement, the old uses are stored (`SnapshotIR`). Finally, if the optimistic assumption was used, we must corroborate its findings by performing another pass. Before this additional pass, we have to roll back the changes made to the function (`RollbackIR`).

There are two cases where the optimistic assumption is applied: when evaluating phi instructions and when performing `Phi-Of-Ops`. If, during a given pass over the function, at least one use of the optimistic assumption leads to a simplified phi expression or a `Phi-Of-Ops`, then we consider that pass to be an optimistic pass.

Before presenting the full algorithm, we need the following definitions:

$$\begin{aligned} \mathit{Partition} &:: \mathit{Expression} \to \mathcal{P}(\mathit{Register}) \\ \mathit{Use} &:: \mathit{User} \times \mathit{Value} \times \mathit{Index} \\ \mathit{Snapshot} &:: \mathit{Instruction} \to \mathcal{P}(\mathit{Use}) \end{aligned}$$

A partition is a set of congruence classes, each identified by a unique expression—its defining expression. A use is represented by the instruction where it is used (User), the value used, and the operand index. The snapshot data structure maps instructions to their set of old uses before the optimistic simplification. The algorithm is presented in 3.1.

For simplicity, some of the procedures are not defined here; instead, an informal definition is provided. The functions `RPO` and `minRPO` sort a set of instructions and return a minimum, respectively, according to a reverse post-ordering of the CFG.

The procedure `ProcessOutgoingEdges` evaluates terminator instructions, updating the reachability of the CFG.

`ReplaceOps` takes an expression and replaces its operands with their known class leaders. In the case of phi expressions, if an edge is optimistically ignored, then `Optimistic` is set to true. `Simplify` is essentially LLVM's `InstructionSimplify`, returning either a constant or a subexpression of the original expression.

Finally the `MoveToClass` procedure moves the register defined at *inst* to the class identified by expression e'.

4 Bug Fixes and Improvements

In this section, we highlight some of the work upstreamed to LLVM, including bug fixes and other improvements.

4.1 Class Leader with Minimum RPO Number

Cyclic dependencies in `NewGVN` can result in miscompilation and termination issues. For example, in Figure 13, we show a function miscompiled by `NewGVN` due to a cyclic dependence. `NewGVN` deems `%xor`

```
1: procedure GVNCheckpointing(function F)
2:     Partition p, p'
3:     for all arg ∈ FunctionArgs do
4:         p'[arg] := arg                                  ▷ Singleton classes for arguments.
5:     end for
6:     p'[⊤] := {register | register ∈ F}                  ▷ Every instruction is optimistically equivalent.
7:     Optimistic := false
8:     Snapshot snap := ∅
9:     repeat
10:        p := p'
11:        if Optimistic then
12:            for all (inst, uses) ∈ snap do                                  ▷ RollbackIR
13:                for all use ∈ uses do
14:                    use.Value := inst
15:                end for
16:            end for
17:        end if
18:        for all inst ∈ RPO(F) do
19:            if inst is Terminator then
20:                ProcessOutgoingEdges(inst, p')
21:            else
22:                (e, Optimistic) := ReplaceOps(inst.expr, p')
23:                e' := Simplify(e)
24:                (e', Optimistic) := Phi-Of-Ops(e', p)
25:                if Optimistic then
26:                    snap[inst] := inst.uses                                 ▷ SnapshotIR
27:                end if
28:                p' := MoveToClass(inst, e')
29:                leader := minRPO(p'[e'])                                    ▷ UpdateIR
30:                for all use ∈ inst.uses do
31:                    use.Value := leader
32:                end for
33:            end if
34:        end for
35:    until p = p'
36:    return F
37: end procedure
38:
```

and %neg to be equivalent.

```
define void @main(i1 %c1, i32 %x) {
  br i1 %c1, label %L, label %end

L:
  %d.1 = phi i8 [ undef, %entry ], [ -1, %L ]
  %conv = sext i8 %d.1 to i32
  %xor = xor i32 %x, %conv
  %neg = xor i32 %xor, -1
  call void @foo(i32 %neg)
  br label %L

end:
  ret void
}
```

```
define void @main(i1 %c1, i32 %x) {
  br i1 %c1, label %L, label %end

L:
  %xor = xor i32 %x, -1
  call void @foo(i32 %xor)
  br label %L

end:
  ret void
}
```

Figure 13: NewGVN miscompilation due to cyclic dependency.

The cyclic dependence occurs because %neg depends on %xor as its operand, and %xor depends on %neg because it is the class leader for the expression xor i32 %x, -1. The defining expression for %neg is based on an incorrect optimistic assumption. When we process %xor, we use the stale leader %neg, which in turn leads to %neg evaluating to the same incorrect defining expression.

Pass	Instruction	Defining Expression	Note
1	%d.1	undef	Ignores unreachable backedge
	%conv	0	Simplify : refines undef to 0
	%xor	%x	Simplify
	%neg	xor i32 %x, -1	Replaces %xor with leader
2	%d.1	-1	Refines undef to -1
	%conv	-1	Constant propagation
	%xor	xor i32 %x, -1	Leader for expression is %neg
	%neg	xor i32 %x, -1	Replaces using stale leader

We fixed this by ensuring that the class leader is always the member with the lowest RPO number.[5] This guarantees that the class leader is processed before all other members, making the cyclic dependence impossible.

4.2 Fix caching for `OpIsSafeForPhiOfOps`

As previously discussed, `NewGVN` does not recursively translate expressions when performing *Phi-Of-Ops*. Instead, it uses the `OpIsSafeForPHIOfOps` function to check if the original instruction depends on a phi in the block where the *Phi-Of-Ops* is being performed. To avoid redundant code walking, the results are cached in `OpSafeForPHIOfOps`. However, the caching mechanism was unsound because safety is block-specific, meaning the cached results cannot generally be used for other blocks.

We fixed this by implementing a cache per block instead of a single one for the entire function.[6]

4.3 Relax Conditions When Checking Safety of Memory Accesses

Since we are operating with MemorySSA, we can treat memory versions as any other operand. When performing *Phi-Of-Ops*, we have to perform the same safety checks as for regular operands. For instance,

[5] https://github.com/llvm/llvm-project/pull/82110
[6] https://github.com/llvm/llvm-project/pull/98340

in Figure 14, the `Phi-Of-Ops` for `%n63` is not performed because it depends on a phi other than `%n59`—in this case, a phi of memory versions. To support this case, we would need to phi-translate `%n60`.

```
define void @function-in-loop(ptr %p) {
bb56:
  br label %bb57

bb57:                                             ; preds = %bb229, %bb56
; 2 = MemoryPhi({bb56,liveOnEntry},{bb229,1})
  %n59 = phi i1 [ false, %bb229 ], [ true, %bb56 ]
; MemoryUse(2)
  %n60 = load i8, ptr %p, align 1
  %n62 = icmp ne i8 %n60, 2
  %n63 = or i1 %n59, %n62
  br i1 %n63, label %bb229, label %bb237

bb229:                                            ; preds = %bb57
; 1 = MemoryDef(2)
  call void @f()
  br label %bb57

bb237:                                            ; preds = %bb57
  ret void
}
```

Figure 14: The loaded value depends on the loop iteration.

However, `NewGVN` does not perform the safety checks for memory operands as it does for regular operands; instead, it bails as soon as it encounters an operand that may read from memory. This prevents the application of *Phi-Of-Ops* in cases such as the one shown in Figure 15. In this case, the target of the *Phi-Of-Ops* depends on a load (`%n60`); however, the loaded value is independent of the predecessor. Therefore, it is safe to perform the transformation without additional phi-translation.

There is currently a pull request[7] under review to generalize `OpIsSafeForPHIOfOps` to also perform a code walk for memory operands. The main challenge is that MemorySSA is not a native part of the IR.

[7] https://github.com/llvm/llvm-project/pull/98609

```
define void @no-alias(ptr noalias %p, ptr noalias
↪  %q) {
bb56:
  br label %bb57

bb57:
; 2 = MemoryPhi({bb56,liveOnEntry},{bb229,1})
  %n59 = phi i1 [ false, %bb229 ], [ true, %bb56 ]
  %idx = phi i8 [ 0, %bb56 ], [ %inc, %bb229 ]
; MemoryUse(liveOnEntry)
  %n60 = load i8, ptr %p, align 1
  %n62 = icmp ne i8 %n60, 2
  %n63 = or i1 %n59, %n62
  br i1 %n63, label %bb229, label %bb237

bb229:
  %inc = add i8 %idx, 1
; 1 = MemoryDef(2)
  store i8 %inc, ptr %q, align 1
  br label %bb57

bb237:
  ret void
}
```

```
define void @no-alias(ptr noalias %p, ptr noalias %q) {
bb56:
  br label %bb57

bb57:
; 2 = MemoryPhi({bb56,liveOnEntry},{bb229,1})
  %phiofops = phi i1 [%n62, %bb229], [true, %bb56]
  %idx = phi i8 [ 0, %bb56 ], [ %inc, %bb229 ]
; MemoryUse(liveOnEntry)
  %n60 = load i8, ptr %p, align 1
  %n62 = icmp ne i8 %n60, 2
  br i1 %phiofops, label %bb229, label %bb237

bb229:
  %inc = add i8 %idx, 1
; 1 = MemoryDef(2)
  store i8 %inc, ptr %q, align 1
  br label %bb57

bb237:
  ret void
}
```

Figure 15: The loaded value is loop invariant.

5 Evaluation

In this section, we provide some benchmarks for the solutions presented in Sections 2 and 3. The benchmarking was carried out using the open-source, automated benchmarking tool Phoronix Test Suite.[8] The chosen C/C++ applications are listed in Table 1.

Source			
aircrack-ng-1.3.0	encode-flac-1.8.1	luajit-1.1.0	scimark2-1.3.2
botan-1.6.0	espeak-1.7.0	mafft-1.6.2	simdjson-2.0.1
compress-zstd-1.6.0	fftw-1.2.0	ngspice-1.0.0	sqlite-speedtest-1.0.1
crafty-1.4.5	john-the-ripper-1.8.0	quantlib-1.2.0	tjbench-1.2.0
draco-1.6.0	jpegxl-1.5.0	rnnoise-1.0.2	graphics-magick-2.1.0

Table 1: Selected profiles from Phoronix (the versions numbers are specific to Phoronix [9])

.

Each profile was compiled with `-O2` and tested under five different configurations for GVN: `GVNPRE` with default options (`baseline`), optimistic without algebraic simplification (`NoSimpl`), optimistic with algebraic simplification (`Simpl`), optimistic with algebraic simplification and full redundancy elimination (`FRE`), and optimistic with algebraic simplification and partial redundancy elimination (`PRE`).

Each profile has a series of tests, and their results are aggregated in Figure 16. We present the performance impact (%) relative to a baseline of `GVNPRE`. The test results are categorized into different classes: severe degradation (< -5), moderate degradation ($[-5, 2[$), slight degradation ($[-2, -0.5[$), noise ($[-0.5, 0.5[$), slight improvement ($[0.5, 2[$), moderate improvement ($[2, 5[$), and improvement (> 5).

[8] https://www.phoronix-test-suite.com/

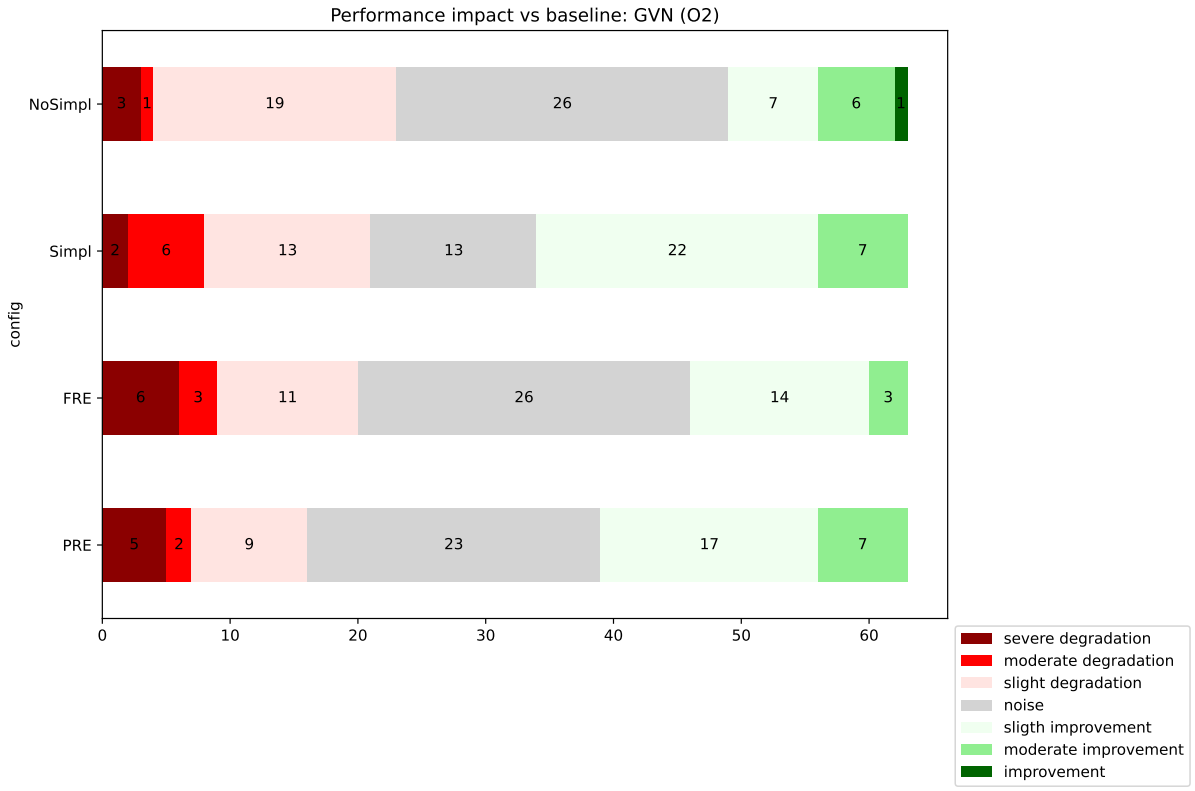

Figure 16: Performance impact different `NewGVN` configurations against `GVNPRE`.

5.1 Discussion

At first glance, the results suggest that implementing FRE and PRE in `NewGVN` does not yield significant benefits; in fact, there are cases of severe degradation for these configurations. However, it is important to note that the additional severe degradation is observed only in the jpegxl-1.5.0 profile. This is likely due to unexpected interactions between the code produced by our implementations and the rest of the pipeline, which requires further investigation.

Even when excluding the tests from the jpegxl-1.5.0 profile, some unexpected results remain. The configuration with the most cases of improvement was `Simpl`. This is likely because other passes do not handle the code generated by our implementation well. To address this, we need to identify where these negative interactions occur and either adjust the other passes to accommodate the code introduced by FRE and PRE or refine the heuristics for applying FRE and PRE.

Overall, the general trend when introducing FRE and PRE is to achieve the same or better performance, as evidenced by the decreasing number of degradation cases.

6 Conclusion and Future Work

During this project, we made significant progress toward replacing `GVNPRE` with `NewGVN`.

We successfully implemented a PRE algorithm in `NewGVN`. The key advantage of our approach is its integration with the existing framework. While the performance results are promising, the implementation still requires further development, particularly to address missing features such as load coercion and critical edge splitting. Additionally, improved heuristics and fine-tuning are needed to enhance interaction with the rest of the pipeline.

Furthermore, some longstanding issues have been fixed or are currently being addressed, making it easier for users to adopt `NewGVN` over `GVNPRE`.

We plan to continue this project to resolve the aforementioned issues and further improve `NewGVN`.

References

[1] C. Click and K. D. Cooper, "Combining analyses, combining optimizations," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 2, p. 181–196, mar 1995.

[2] K. Gargi, "A sparse algorithm for predicated global value numbering," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02, 2002.

[3] S. Ananian, "The static single information form," 2001.

[4] D. Novillo, "Memory ssa-a unified approach for sparsely representing memory operations," in *Proc of the GCC Developers' Summit*, 2007.

[5] E. Morel and C. Renvoise, "Global optimization by suppression of partial redundancies," *Commun. ACM*, vol. 22, no. 2, p. 96–103, 1979.

[6] S. Dasgupta and T. Gangwani, "Partial redundancy elimination using lazy code motion," *CoRR*, vol. abs/1905.08178, 2019.