TÉCNICO LISBOA

Optimistic Global Value Numbering for Compilers with Undefined Behavior

Manuel José Gomes Brito

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Nuno Claudino Pereira Lopes
Doctor Alina Sbîrlea

Examination Committee

Chairperson: Prof. Manuel Fernando Cabido Peres Lopes
Supervisor: Prof. Nuno Claudino Pereira Lopes
Member of the Committee: Prof. José Faustino Fragoso Femenin dos Santos

November 2024

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Abstract

Compilers play a vital role in modern software systems by bridging the gap between high-level programming languages and the underlying hardware. With the rise of highly specialized languages and diverse hardware platforms, the complexity of compilation has increased. This complexity is managed using intermediate representations (IR), which enable machine-independent optimizations such as Global Value Numbering (GVN). GVN identifies and eliminates redundant computations by assigning value numbers to equivalent expressions. However, traditional GVN faces challenges in detecting more complex redundancies, particularly in loops and partially redundant computations.

This thesis introduces a GVN algorithm tailored for SSA-based intermediate representations, enhancing the precision and efficiency of GVN through techniques like algebraic simplification, partial redundancy elimination (PRE), unreachable code elimination (UCE), and the optimistic assumption. By leveraging extensions like Static Single Information (SSI) and Memory SSA (MSSA), our approach improves value numbering for memory operations and predicates, allowing for the detection of complex redundancies in both scalar and memory-based computations.

Our solution, implemented within the LLVM compiler, demonstrates performance improvements of up to 30% over existing implementations in LLVM. Additionally, the combination of optimizations enables unique transformations, such as loop-invariant code motion. However, the study also highlights cases where value numbering can degrade performance, underscoring the importance of careful optimization pipeline design.

Keywords

Compiler; Optimization; Intermediate Representation.

Resumo

Os compiladores desempenham um papel crucial nos sistemas de software modernos, ao fazerem a ponte entre linguagens de programação de alto nível e o hardware subjacente. Com o crescimento das linguagens altamente especializadas e plataformas de hardware diversificadas, a complexidade do processo de compilação aumentou. Esta complexidade é gerida através de representações intermédias (IR), que permitem otimizações independentes da máquina, como Global Value Numbering (GVN). GVN identifica e elimina cálculos redundantes, atribuindo value numbers a expressões equivalentes. No entanto, o algoritmo de GVN tradicional enfrenta desafios na deteção de redundâncias mais complexas, particularmente em ciclos e cálculos parcialmente redundantes.

Esta tese apresenta um algoritmo GVN adaptado para representações intermediárias baseadas em SSA, melhorando a precisão e a eficiência do GVN através de técnicas como simplificação algébrica, eliminação de redundâncias parcias (PRE), eliminação de código inalcançável (UCE) e a suposição otimista. Ao tirar partido de extensões como Static Single Information (SSI) e Memory SSA (MSSA), a nossa abordagem permite a deteção de redundâncias complexas tanto em cálculos escalares como em operações de memória.

A nossa solução, implementada no compilador LLVM, demonstra melhorias de desempenho de até 30% em relação às implementações existentes no LLVM. Adicionalmente, a combinação de otimizações permite transformações únicas, como loop-invariant code motion. No entanto, o estudo também destaca casos onde o GVN pode degradar o desempenho, sublinhando a importância de um desenho cuidadoso da pipeline de otimização.

Palavras Chave

Compilador; Otimização; Representação Intermédia.

Contents

List of Figures

1

Introduction

Contents

A compiler is a crucial component of any software system, acting as the bridge between human-written programs and the machines executing them. Bridging this gap has become increasingly challenging due to the rapid evolution of both programming languages and hardware platforms.

On the programming side, we now have highly specialized languages and frameworks with higher levels of abstraction. These tools enable developers to efficiently tackle cutting-edge challenges, such as blockchain, artificial intelligence, and cloud computing. However, they also widen the gap between developers and hardware, making the compilation process more complex. Another key factor contributing to this complexity is the increasing size and intricacy of modern software systems.

On the hardware side, the landscape is diverse, encompassing mobile processors, specialized AI/ML accelerators, GPUs for graphics applications, and more. The compiler's role is to optimize code for these various platforms, ensuring optimal performance for each specific use case.

To manage the complexity of modern languages and diverse hardware targets, most compilers use an intermediate representation (IR) [1, 2]. This representation serves as a middle ground between the

source language and machine code. It is high-level enough to facilitate program analysis and transformation while being low-level enough to support lowering to machine code. The IR allows source and target languages to share a set of machine-independent optimizations [3], which enhance the generated code in a target-agnostic manner by analyzing and transforming the IR.

One such optimization is Global Value Numbering (GVN) [4–9], which removes redundant operations within functions. This is achieved in two steps. First, equivalent expressions are identified by assigning value numbers. Then, redundancies are eliminated by replacing them with an equivalent instruction computed earlier.

Since determining the equivalence of expressions in general is undecidable [10], value numbering algorithms typically focus on detecting Herbrand-equivalences, which are equivalences between expressions with the same operator and Herbrand-equivalent operands. However, GVN can be unified or extended with other optimizations to produce more precise results.

For example, GVN typically leverages algebraic simplifications to uncover additional redundancies [11], allowing it to discover redundancies based on the semantic equivalence of expressions rather than just Herbrand-equivalence. In the following example, we have two programs where GVN is able to detect and eliminate the redundant computation (`b`) and consequently replace `g(b)` with `g(a)`. Note, however, that detecting the redundancy on the right requires recognizing that addition is commutative.

```
int f(int x, int y) {
    a = x + y;
    ...
    b = x + y;
    use(b);
}
```

```
int g(int x, int y) {
    a = x + y;
    ...
    b = y + x;
    use(b);
}
```

Another optimization that works well with GVN is unreachable code elimination (UCE) [1]. UCE aims to remove code from a program that is never executed. When combined with GVN, UCE can discover, for example, that an execution path is never taken based on an equivalence detected by GVN. Conversely, applying UCE to a program reduces the number of instructions that GVN needs to process, which not only decreases the runtime of the algorithm but also helps GVN discover more redundancies.

Partial redundancy elimination (PRE) is another optimization that complements GVN effectively. PRE aims to remove partial redundancies—i.e., when a computation is redundant in some but not all execution paths. This is achieved by inserting the computation into non-redundant paths, thus making the original computation fully redundant. An example transformation is presented in Figure 1.1. Traditional PRE algorithms can only detect partial redundancies among lexically identical instructions. However, combining the two optimizations allows PRE to detect partial redundancies among the instructions identified as redundant by GVN.

Another technique used to enhance the precision of GVN is the optimistic assumption [6, 12]. This

```
int f(int x, int y) {
    if (...) {
        a = x + y;
    } else {
        ...
    }
    b = y + x;
}
```

```
int f_pre(int x, int y) {
    if (...) {
        a = x + y;
    } else {
        ...
        b.pre = x + y;
    }
}
```

Figure 1.1: Example of PRE: In the `if` path, `x+y` is computed twice, while in the `else` path, it is only computed once. A new computation of the expression is inserted in the `else` path to make the value fully redundant, allowing us to safely remove the instruction after the `if-else` block.

assumption helps discover equivalences within loops. Optimistic GVN algorithms initially assume that loops execute only their first iteration, allowing them to identify equivalences that hold across all iterations once proven. The equivalence information discovered under this assumption applies to any iteration.

For example, consider a loop where the variables `x` and `y` are equivalent. An optimistic GVN algorithm assumes we are in the first iteration, assigning the constant `1` to both `x` and `y`. This allows the algorithm to determine that after the first iteration, the variables remain equivalent. An inductive argument can be made that this equivalence holds for all iterations. In contrast, a traditional or pessimistic GVN algorithm would conservatively assume that `x` and `y` could take any value and would not be able to discover the cyclic equivalence.

```
x = 1
y = 1
while(...)
    x = x + 1
    y = y + 1
```

Besides cyclic equivalences, the optimistic assumption also allows GVN to discover loop invariants when the value after the first iteration is equal to the initial value. Additionally, it may sometimes be the case that the assumption was correct and the loop actually executes only once; making this assumption is necessary to discover that.

Furthermore, there are improvements to the IR itself that enhance GVN. For instance, the Static Single Assignment (SSA) form [13, 14] is one such improvement. In an SSA-based IR, variables are defined only once, ensuring that a variable has the same value regardless of where it is used. In the context of value numbering, this eliminates the concern of an operand of an instruction being redefined, which would change the value numbering of the instructions. A generalization of SSA form is the Static Single Information (SSI) form [15].

SSI generalizes the concept of each variable being associated with a single value to other types of information beyond just its value. For instance, in an SSI-based IR, if a variable is used in a conditional branch test, the IR splits the variable into two versions: one for the block where the test is true and another for the block where it is false. The general idea behind SSA and SSI is to embed the relevant

information directly into the variable itself, resulting in a sparse representation. This means that each variable version contains information about where it is valid, rather than needing a separate mapping for this purpose.

In contrast, a dense representation explicitly tracks not only the variable or register but also the specific location (e.g., a basic block or line number) where the value is valid. This requires maintaining a more detailed mapping—(register, location) $\rightarrow$ value—resulting in additional overhead. Sparse representations like SSA and SSI, by embedding more information directly within the variables, allow for richer program information with reduced complexity.

In modern computers, much of the execution time is spent performing memory accesses. It is therefore critical for GVN to detect redundancies among memory operations. An extension of SSA that helps with this is Memory SSA (MSSA) [16]. MSSA follows the same principles as SSA but is specifically applied to memory operations. In MSSA, memory is viewed as a value defined by memory-defining operations, such as stores, and used by memory-reading operations, such as loads. Just as with variables in SSA, memory variables are defined only once. This provides us with a functional view of memory, greatly simplifying the detection of redundant loads and stores.

1.1 Contributions

This thesis focuses on developing a global value numbering algorithm for modern compilers. We have created an optimistic algorithm for SSA-based intermediate representations (IRs), which is unified with partial redundancy elimination, algebraic simplification, and dead code elimination. Additionally, it leverages the Static Single Information (SSI) and Memory SSA (MSSA) extensions to utilize information from predicates and perform value numbering of memory operations, respectively.

In Table 1.1, we present a summary of the features lacking in the existing value numbering implementations within the LLVM compiler that our solution supports. In addition to not benefiting from the individual optimizations and extensions, there are transformations that can only be performed when all optimizations and extensions interact within the same framework. For example, the interaction between the optimistic assumption and partial redundancy elimination enables GVN to perform loop-invariant code motion, which involves moving loop-invariant code out of loops.

The designed solution was implemented in the LLVM compiler. The implementation was evaluated using open-source C/C++ applications and the default LLVM optimization pipelines. In numerous test cases, our solution outperformed existing LLVM value numbering implementations, achieving performance improvements of over 30% in some instances. Additionally, we evaluated different variants of our solution by removing features from the full implementation, which allowed us to conclude that an optimistic algorithm with only algebraic simplification does not have a significant positive impact on per-

	GVN	NewGVN	OptPRE
Optimistic	×	✓	✓
Simplification	✓	✓*	✓
PRE	✓	×	✓
Memory VN	×	✓	✓
SSI	×	✓	✓

Table 1.1: Comparison of features from the LLVM implementations GVN and NewGVN in relation to our implementation, OptPRE. (*) The simplifications performed by NewGVN are incomplete, even when excluding the optimistic assumption; we discuss this issue in more detail in Chapter 5

formance in the context of a modern general-purpose optimization pipeline, such as LLVM's. For GVN to have a positive impact, it needs to be unified with other optimizations, such as partial redundancy elimination. This unification produces transformations that are unique to this optimization; otherwise, the transformations performed by GVN become redundant when interacting with other optimizations in the pipeline.

Moreover, we identified several cases where the use of value numbering degraded performance, including instances of nearly 50% regression, regardless of the implementation used. These cases highlight the difficulty of designing optimization pipelines, where seemingly beneficial optimizations can cause regressions when interacting with downstream passes of the pipeline.

1.2 Outline

This document is structured as follows. In Chapter 2 we introduce the design philosophy behind modern compilers and introduced some common program representations used by them. In Chapter 3 we present the existing GVN algorithms and how they evolved over time. Our main contribution is in Chapter 4 where we present our novel algorithm, followed by its implementation in the LLVM compiler in Chapter 5. The evaluation of our solution is presented in Chapter 6 alongside an analysis and discussion of the results. Chapter 7 concludes the document with a discussion of possible future work.

2

Background

Contents

2.1 Compilers

A compiler translates computer code from a source language to a target language. Compilers are typically divided into three distinct stages: the front end, the middle end, and the back end.

The front end analyzes and parses the source code, ensuring syntactic and semantic correctness. It outputs the code compiled into an intermediate representation (IR) that is then passed to the subsequent stages of the compiler.

The middle end optimizes the IR, improving efficiency and performance. This stage consists of analysis and transformation passes. The former gathers information about the program, while the latter transforms the program using this information. These passes constitute an optimization pipeline, where

the output of one pass serves as the input to the subsequent pass. An example of a middle end optimization is constant propagation, which discovers and propagates expressions computable at compile time.

The final stage is the back end, where the optimized IR is transformed into the desired target language. Additionally, the back end performs target-specific optimizations. In practice, modern compilers also leverage the notion of intermediate representation in the back end, as the analysis required for some of its stages might be more challenging or less efficient if conducted directly in the target language.

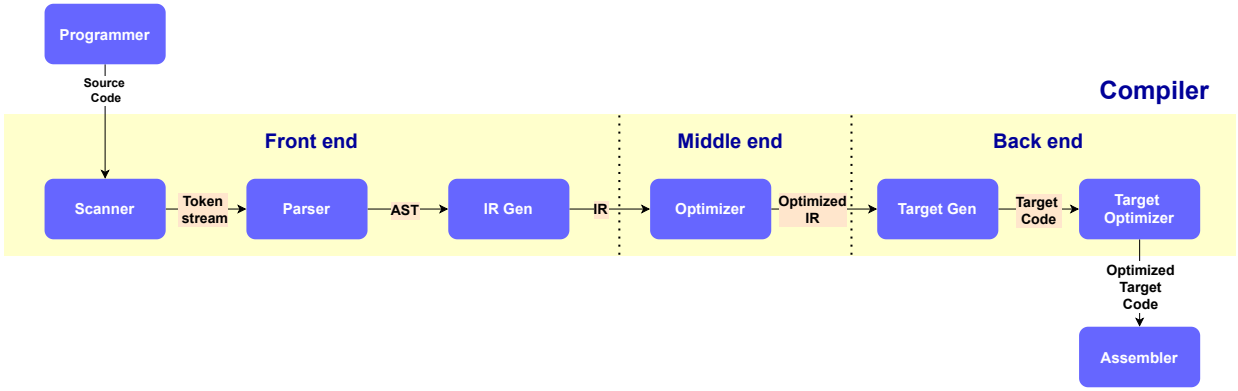


Figure 2.1: Example compiler pipeline.

This three-stage design enhances modularity and portability, allowing the middle end to be shared among various source and target languages (Figure 2.2). Each source language has its frontend, and each target language has its backend, while the middle end is shared among them.

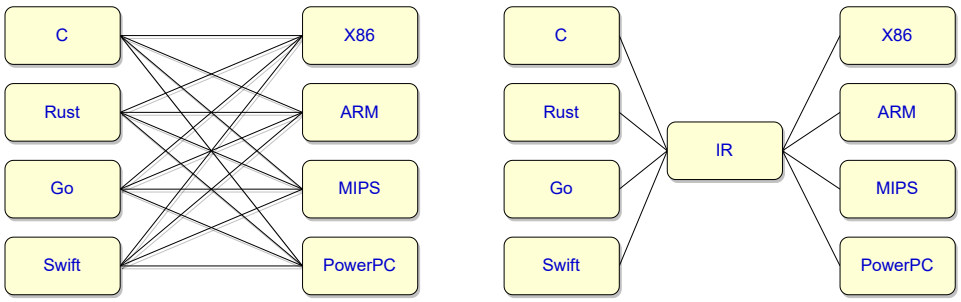


Figure 2.2: Monolithic (on the left) vs modular (on the right) compiler design

2.2 Intermediate Representation

Using a single Intermediate Representation (IR) for all source and target languages poses some design challenges. An effective IR must preserve the semantics and intent of the source program. It should also allow for safe and efficient analyses and transformations within the IR itself. Additionally, the IR

```
int f(int x, int y) {
    int a, b, c;
    a = x + y;
    b = 4 * y;
    if (a > 4) {
        c = b - a;
    } else {
        c = 2 * b;
    }
    return c;
}
```

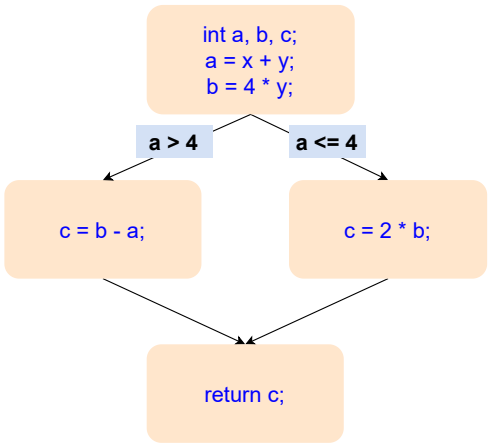


Figure 2.3: A simple function in C (left) and the corresponding CFG (right).

should facilitate efficient translation to the target language, considering the balance needed for different processor architectures.

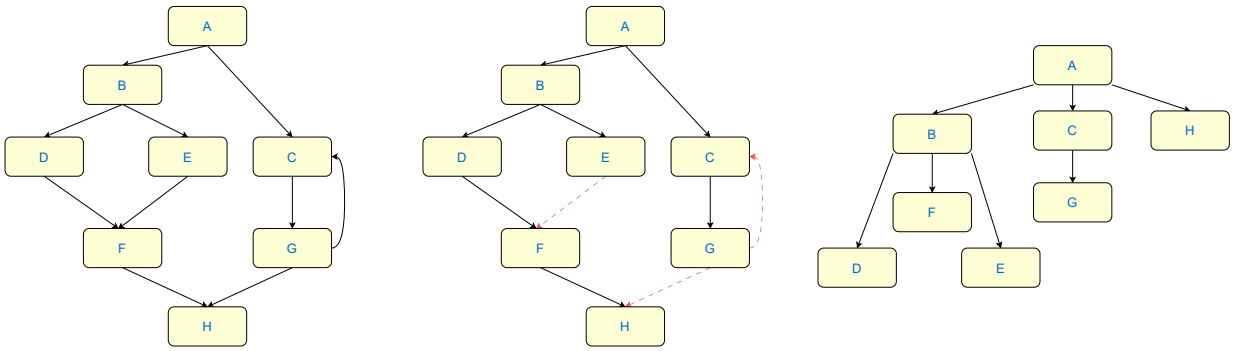
IR design and program representation is a vast field [17]. In the following section we introduce some common program representations and discuss their merits and downsides.

2.2.1 Control Flow Graph

The Control Flow Graph (CFG) [18] represents all paths that occur during the execution of a program. In this representation, the nodes of the graph are called basic blocks and the edges represent jumps in the control flow. A basic block is a sequence of program statements that ends in a statement that changes the control flow, e.g., a branch, or a return. Since execution always starts at the beginning of the basic block, and the only jump is at the end of the basic block, it follows that if one statement of the basic block is executed, then the entire basic block is executed. Figure 2.3 provides an example of a CFG.

Dominance An important property of a CFG is dominance [19]. It is a crucial part of analyses based on control flow. A basic block A dominates another basic block B, written $dom(A, B)$, if all paths from the entry basic block that reach B go through A. A basic block A strictly dominates basic block B, written $sdom(A, B)$, if $dom(A, B)$ and $A \neq B$. A basic block A immediately dominates basic block B, written $idom(A, B)$, if A strictly dominates B and A does not strictly dominate any basic block that strictly dominates B.

From the notion of immediate dominance it is possible to construct a data structure called the dominator tree. The dominator tree $DomTree$ is a directed graph where the nodes represent basic blocks



Basic Block	A	B	C	D	E	F	G	H
DFS_{In}	1	2	12	3	9	4	13	5
DFS_{Out}	16	11	15	8	10	7	14	6

Figure 2.4: An example CFG and a corresponding DFS and dominator trees. At the bottom, the DFS numbers for each basic block, where DFS_{in} marks when the block was first visited and DFS_{out} marks when the block was last visited.

and where an edge (A, B) represents that A immediately dominates B.

CFG Traversal The manner in which we traverse the graph is crucial for many analyses. Depending on the order in which the nodes are visited, we have different time complexity guarantees for the analyses.

In the Pre Order ($<_{pre}$), nodes are arranged based on their order of first visit during a depth-first search traversal (DFS_{in}). Specifically, for basic blocks A and B, $A <_{pre} B$ holds if and only if $DFS_{in}[A] < DFS_{in}[B]$.

On the other hand, in the Post Order ($<_{post}$), nodes are arranged by their last visit in the DFS traversal (DFS_{out}). For basic blocks A and B, $A <_{post} B$ is satisfied if and only if $DFS_{out}[A] < DFS_{out}[B]$.

For instance, consider a Pre Order traversal of the CFG shown in Figure 2.4 as $[A, B, D, F, H, E, C, G]$. A corresponding Post Order traversal for the same CFG could be $[H, F, D, E, B, G, C, A]$. It is important to note that both Pre and Post Order traversals are influenced by the chosen DFS traversal.

Reverse Post Order For many analyses, the preferred traversal order is Reverse Post Order (RPO). In RPO, nodes are arranged in the reverse order of their last visit in a DFS traversal. RPO ensures that a parent node is visited before its children, except for retreating edges. A retreating edge is an edge (A, B) where B is visited before A in a DFS traversal. In this context, we consider CFGs where all retreating edges are backedges, which are a special type of retreating edge where B dominates A.

An example RPO traversal for the CFG in Figure 2.4 is $[A, C, G, B, E, D, F, H]$, where the edge (G, C) is a backedge since C dominates G.

This definition implies that a basic block dominating another is always visited first in an RPO traversal.

In other words, if A dominates B, then $A <_{rpo} B$. However, it is important to note that the converse is not always true. For instance, in Figure 2.4, although $C <_{rpo} B$, C does not dominate B.

Dominator Tree DFS Another method to traverse the CFG involves using a standard DFS traversal of the Dominator Tree (DT), specifically a Pre Order of the dominator tree. For instance, in the CFG shown in Figure 2.4, a possible DT DFS traversal is $[A, B, D, F, E, C, G, H]$. Similar to RPO, this method ensures that a basic block dominating another is always visited first.

Moreover, it provides stronger guarantees regarding dominance. For basic blocks A, B, C, the following property holds:

$$A <_{dt} B <_{dt} C \wedge dom(A, C) \implies dom(A, B)$$

This property asserts that if A dominates C, then every node between them in a DT DFS traversal is also dominated by A. This is particularly valuable when searching for blocks dominated by another block in a sequence of DT DFS-ordered blocks. As soon as we encounter the first block that is not dominated, we can halt the search.

However, a drawback of using the DT DFS order is that it does not provide the same guarantees as RPO concerning visiting parent blocks before child blocks. For instance, in the DT DFS traversal $[A, B, D, F, E, C, G, H]$, block F is visited before one of its parent blocks, E.

2.2.2 Static Single Assignment

Static Single Assignment (SSA) [13, 14, 20, 21] is one of the most common IR types nowadays. SSA requires that each variable is assigned exactly once and that this assignment dominates its uses. An important consequence of this property is referential transparency, ensuring that a variable yields the same value regardless of its location in the program. This enables a more efficient type of dataflow analysis called sparse analysis, where dataflow information is directly related to the variable. Meanwhile, its counterpart, dense analysis, has to keep track of the program location at which the dataflow information is applicable.

The following example illustrate the advantages of using an SSA IR:

```
x = 1;
y = x - 1;
x = 2;
z = x - 1;
```

```
x0 = 1;
y0 = x0 - 1;
x1 = 2;
z0 = x1 - 1;
```

On the left, an example program in a non-SSA IR, where in order to discover that `y` and `z` are not equivalent we would have to keep track of the different values of `x`. Meanwhile on the right, the same program in SSA, that fact can be more easily derived since `x0` and `x1` are distinct variables.

The SSA form is built by adding a new variable whenever a variable is reassigned and then changing

every subsequent use of the old variable to refer to the new one. While variable renaming suffices when working with straight-line code, control flow is more challenging to handle.

In Figure 2.5, on the left, there is an `if-else` statement where conditionally different values are assigned to `x`, which is then returned. In the middle, variable renaming is applied; however, a problem arises as it is unclear which variable should be returned. The solution involves the introduction of a pseudo-instruction called ϕ (phi). The ϕ instruction operates as a multiplexer, returning the correct variable based on the path that reaches it. On the right, we now have a program that adheres to the SSA properties.

```
if (...)
    x = ...
else
    x = ...
return x
```

```
if (...)
    x0 = ...
else
    x1 = ...
return x?
```

```
if (...)
    x0 = ...
else
    x1 = ...
x2 = φ (x0, if), (x1, else)
return x2
```

Figure 2.5: Building SSA form in a program with control flow.

2.2.3 Static Single Information

SSA is a special case of a more general notion known as Static Single Information (SSI) [15]. Recall that SSA form ensures that the value of a variable is invariant, i.e., after the variable is defined, its value is independent of the program's execution flow, allowing for sparse analysis. SSI generalizes this idea to other types of dataflow information.

Take, for example, the program in Figure 2.6; the left side shows a simple SSA program. In each branch, the variable `c` is assigned a value based on whether `a` is equal to `b`. In the `then` branch, knowing the predicate is true allows for the simplification of `c1` to 0.

To efficiently achieve this, a pseudo-instruction π is introduced. Unlike the ϕ node, which merges multiple incoming variables into one, the π node splits a variable into multiple ones. Each new variable is dominated by an invariant dataflow fact. In this case, the dataflow fact is given by the predicate `a == b`.

Returning to Figure 2.6 on the right, the program is now in SSI form, and `a1` is consistently known to be equal to `b` across all its uses. Later we will use the SSI form with predicate information to achieve a more precise GVN algorithm.

2.2.4 Memory SSA

Another extension of SSA is Memory SSA (MSSA) [16]. It arises from the need to perform sparse analysis on memory operations. MSSA applies the same principles as SSA form to memory operations like loads and stores. Notably, MSSA provides a functional perspective on memory. In this approach,

Figure 2.6: On the left is a program in SSA. On the right is the same program, but in SSI form.

each memory-writing operation, such as a store, takes a memory as input and produces a new version of the memory. Each memory-reading operation also receive a memory as input.

For instance, in the context of value numbering, MSSA enables efficient checking of whether two loads are congruent. Since memory is just another operand of the load, two loads are congruent if they load the same type from the same pointer in the same memory. In the example below, load `l2` retrieves the same value as `l1`. Additionally, we can determine their values because they are loading from the pointer where `value` was stored at `mem1`.

```
entry:
    mem1 = store ptr, value, mem0
    l1 = load ptr, mem1
    ...
    l2 = load ptr, mem1
```

A key difference between MSSA and regular SSA is that SSA operates on scalar variables, while MSSA operates on the entire memory. When a scalar variable changes, it changes in its entirety, whereas memory instructions only read or write a portion of the memory. To illustrate this difference, we present an example in Figure 2.7.

At first glance, the two loads are not equivalent. Although they index the memory using the same pointer, they operate on different memories, namely `mem1` and `mem2`, respectively. However, if we know that pointer `ptr1` does not alias with pointer `ptr2`, then we know that for the memory blocks indexed by `ptr1`, `mem1` and `mem2` are equivalent. Therefore, the loads are equivalent. In this example, we say that the store defining `mem2` does not clobber the memory block indexed by `ptr1`. It is important to note that determining if two pointers alias, i.e., if they point to the same memory block, is a non-trivial task and requires a dedicated analysis, known as alias analysis [22, 23].

2.3 Undefined Behavior

The concept of undefined behavior [24] was popularized by the C programming language. The language aimed to provide efficient execution on various computer architectures. Achieving the stated goal is particularly difficult when handling corner cases. Take, for example, the following code:

```
entry:
    mem1 = store ptr1, value1, mem0
    l1 = load ptr1, mem1
    ...
    mem2 = store ptr2, value2, mem1
    ...
    l2 = load ptr1, mem2
```

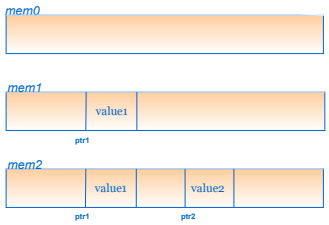


Figure 2.7: Detection of redundant load with non-clobbering memory store.

```
int leftShift(int x, int y) {
    return x << y;
}
```

When y is a value smaller than the bitwidth of the int type, all architectures exhibit the expected behavior. However, if y exceeds the bitwidth, architectures have differing semantics. For example, ARM performs a shift by y modulo the bitwidth, while on x86 the outcome is zero.

Therefore, for C to have a consistent specification, compiled code would need to systematically check these corner cases at run time. However, this is an expensive solution. C's solution was to declare these corner cases as undefined behavior (UB). This means that they are not part of the legal set of behaviors of the language. The burden of avoiding these corner cases is placed onto the user.

The downside is that users may inadvertently write programs with undefined behavior, especially in a UB-heavy language such as C. This leads to programs having unpredictable behavior and security vulnerabilities [25].

To cope with this tools and techniques for UB detection and bug prevention have been developed over the years [26, 27].

Compiler view of UB The compiler assumes that all programs abide by the specification of the language and that everything outside the bounds of this specification never happens. Essentially, the source language is communicating assumptions about the program. So far, we have seen assumptions that arise from the unsafety of the source language, but this is just one use case of UB.

The compiler has a more generalized view of UB, where UB is a way for a language at a higher abstraction level to communicate assumptions to lower abstraction levels. For example, a frontend can make guarantees about the execution of a program through its type-checking system and then communicate them to the optimizer through UB. Another example is in the middle end, where information computed in one pass can be communicated to subsequent passes using UB (i.e., UB caches derived facts).

In preparation for discussing how IR-level UB is modeled, we first discuss when a transformation is valid.

Valid transformation From a high-level view, a translation from a source program to a target program is valid if the target preserves the observable behavior of the source for all inputs, for some definition of observable behavior.

Consider, for example, two programs in the same language where the only difference is that in the second one we add a no-op instruction that takes one second to execute. Whether this translation is valid is predicated on whether or not we consider execution time to be an observable behavior. While usually we do not, in some contexts, preserving execution time may be necessary to ensure certain security properties.

The exact definition of observable behavior depends on the source languages the compiler supports. In this document we consider a model where the observable behavior of a program is its final memory state and returned value.

Immediate UB A possible view of UB is that a program that triggers UB is meaningless and, therefore, it admits any observable behavior. This form of UB is called immediate UB. Hence, from a validity standpoint, a transformation does not need to preserve the source UB. On the other hand, recall that UB communicates assumptions; therefore, a transformation cannot add UB without proper justification.

Immediate UB is the strongest form of UB we can have. We will now see that it is too restrictive for some desirable transformations. Consider the following example transformation:

```
for (int i = 0; i <= n; ++i) {
    a[i] = x + 1;
}
```

```
int b = x + 1;
for (int i = 0; i <= n; ++i) {
    a[i] = b;
}
```

In this code, we are initializing an array. In each iteration, the expression `x+1` is computed. Note, however, that this computation is loop-invariant, so a desirable optimization is to hoist it out of the loop. This transformation is known as loop-invariant code motion (LICM).

This transformation is not valid if integer overflow is immediate UB (as it is in C). Take, for example, an initial state where $(x, n) = (INT_{MAX}, 0)$. Running the source program produces a well-defined final state since control flow never enters the loop. However, the same input for the target program triggers UB due to integer overflow on the hoisted instruction.

To support this transformation, we need a form of UB that allows speculative execution. This can be achieved if UB causes the result of the instruction to have an unpredictable value, but executing the instruction with UB itself is not harmful. This form of UB is called deferred UB.

Deferred UB Deferred UB models that performing the instruction is well-defined, but using its value might not be. With this model of UB, the previous transformation is now correct, since in the case that $(x, n) = (INT_{MAX}, 0)$, the instruction with UB would execute, but the result would not be used.

However, we still reserve the use of immediate UB for cases where executing the instruction causes serious issues, for example, a CPU trap.

We now use LLVM's IR as a case study on how to implement deferred UB. This particular IR has two forms of deferred UB: *undef* and *poison*.

Undef Values An `undef` value is a set of values of a given type. The `undef` value is full if it represents all possible values and partial otherwise. Instructions involving `undef` values are performed element-wise. For example, adding `undef` to any number results in `undef`, while multiplying `undef` by 2 results in a partial `undef`:

```
a = add undef, 1 // {...,-2, -1, 0, 1, 2, ...}
b = mul undef, 2 // { ..., -2, 0, 2, ...}
```

Using an undef value as deferred UB is not only useful for hoisting UB out of control flow but also for modeling uninitialized variables. When using `undef` for uninitialized variables, the compiler does not need to materialize a concrete value.

Modeling deferred UB in this fashion still has its limitations. Take, for example, the following transformation:

```
int a = 2 * b
  =>
int a = b + b
```

This transformation is not valid if we use `undef` to model deferred UB. Suppose that `b` is `undef`. In the source program, the result is a partially `undef` value restricted to even integers, while in the target program, the result is a completely `undef` value.

The problem is that `undef` is too defined in relation to arithmetic operations, and the transformation must preserve its behavior. We need a form of deferred UB strong enough so that the compiler does not need to preserve the behavior in these situations. This is where LLVM introduced the notion of a `poison` value.

Poison Value A `poison` value taints the dataflow graph and degenerates to immediate UB only if it reaches a side-effecting instruction. Using poison for deferred UB would validate the previous transformation, as in both programs, the result would be poison.

However, there are transformations that cannot be justified using `poison` as deferred UB. Take, for example, the following transformation:

```
a = cond ? b : c
  =>
if (cond)
  a = b;
else
  a = c;
```

The goal here is to transform a ternary operator into an `if-else` statement. Suppose that the ternary operator propagates poison, while branching on `poison` is immediate UB. With that said, if `cond` is poison, the original program simply propagates it, while the target program has immediate UB.

The program captures the idea that when UB is triggered, we do not care which value is assigned. In order to match these semantics, we need a way to say that the branch is taken non-deterministically. One could achieve these semantics by introducing a new form of UB to capture this notion that either branch can be taken.

However, having two types of deferred UB with different levels of undefinedness is hard to maintain and is a continuous source of insidious bugs [28]. This is where the `freeze` instruction was introduced.

Freeze instruction The `freeze` instruction is a unary operation that stops propagation of poison values. It is a no-op if its argument is not a poison/undef value; otherwise, a value of the given type is chosen non-deterministically. Now, we are able to perform the previous transformation by freezing the branch condition (`freeze(cond)`).

A key difference between `freeze` and `undef` is that all uses of a frozen value have the same concrete value, while for undef each use may observe a different value.

Behavioral refinement The various forms of UB constitute a hierarchical structure. At the top, there is immediate UB, where no observable behavior is defined. Following it, there is `poison` that taints the dataflow graph and causes immediate UB if it reaches a side-effecting instruction. Next, we have `undef` (and all its partial subsets) that can have a different value each time they are read. Progressing further, we have `freeze(poison | undef)` that has a non-deterministic value consistent between uses. Finally, at the bottom, we have well-defined values.

Going down this hierarchy is always valid; in other words, one can always remove undefinedness from a program while preserving its semantics. This is known as behavioral refinement. However, moving up the hierarchy must be well justified.

IR Flags IR Flags serve as a means to store assumptions provided by frontends and to cache derived facts. The IR encompasses a standard instruction well-defined across its entire domain, and the introduction of flags narrows its well-defined scope.

Taking LLVM's IR as an example, consider two programs that both add one to a number and then return it. One written in a language where signed addition overflow wraps around (well-defined), and another where signed addition overflow is UB. LLVM utilizes the `nsw` flag, signifying "no signed wrap," meaning overflow does not wrap, and the operation yields *poison* if there is an overflow. Thus, (1) would be lowered to `%add = add i32 %num, 1`, while (2) would be lowered to `%add = add nsw i32 %num, 1`.

According to the behavioral refinement hierarchy, it is always valid to eliminate UB. Consequently, discarding UB flags like `nsw` is always valid. However, it is typically undesirable as `nsw` enables certain optimizations. The more undefined a program is, the more leeway the compiler has to optimize, given that a less defined program has fewer observable behaviors to preserve than its more defined counterpart.

Another use case of IR flags is to cache derived facts. When the compiler can prove, for example, that a particular instruction will not cause overflow or that two pointers will not alias, the goal is to convey this information to subsequent optimization passes. This can be efficiently achieved using flags, where they serve as a cache for the compiler's assumptions and derived facts about each instruction.

3

Existing Value Numbering Algorithms

Contents

In this chapter, we introduce the existing work on global value numbering (GVN) and related techniques. This review offers a comprehensive background, setting the stage for the subsequent development and analysis of our GVN algorithm. We start by examining the foundational theories and methodologies that have shaped GVN, tracing its evolution from early implementations to contemporary advancements.

3.1 Hash Based Value Numbering

A form of local value numbering was initially introduced by Cocke and Schwartz [4]. This algorithm uses hashing to assign value numbers. For each instruction in the program, the value numbers of its operands

are computed. The operator, along with the value numbers of the operands, is then hashed to produce the value number for the instruction.

As the algorithm progresses, it maintains a hash table that maps variables to their value numbers. When a computed hash already exists in the table, it identifies a redundancy and replaces the variable currently being processed with a canonical instance. Additionally, the algorithm keeps track of the variable representing a given value number to manage cases where a variable is redefined.

As illustrated in Figure 3.1, this algorithm can easily be extended to handle redundancies hidden behind algebraic properties.

```
a = x + y; // hash("+", x, y) = VNA
b = y + x; // hash("+", x, y) = VNA, "+" is commutative
c = a + z; // hash("+", VNA, z) = VNC
d = b + z; // hash("+", VNA, z) = VNC
e = c - d; // hash("-", VNC, VNC) = 0
```

Figure 3.1: Hash-based value numbering with algebraic simplification.

This algorithm is straightforward and easy to implement. However, a major drawback is that it only works for straight-line code, which is why it is classified as a local algorithm. For instance, in Figure 3.2, manual inspection reveals that the value of `c` is 25 in both paths. However, a local value numbering algorithm is unable to identify this redundancy. Detecting this redundancy would not only be beneficial by allowing us to eliminate the computations associated with the variable `c`, but it would also enable the removal of the `if-else` construct as a side effect.

```
a = x; // hash(x) = VNX
b = 25 + a; // hash("+", 25, VNX) = VNB
if (...)
  c = 25; // hash(25) = VN25
else
  c = b - x; // hash("-", x, VNB) = VNC
ret c; // VN(c) = ??
```

Figure 3.2: Example where basic hash-based value numbering fails to discovery the redundancies.

There is an extension to this local algorithm for SSA IRs that uses the dominance relation to perform value numbering across an entire function [29]. In this approach, basic blocks are traversed in DFS order of the dominator tree, ensuring that a definition is visited before its uses, except for phi functions with values carried by backedges. In such cases, the algorithm adopts a pessimistic approach, assigning fresh value numbers to these phi functions and treating them as non-redundant.

3.2 Kildall's Dataflow Analysis

Kildall [5] introduced a framework for conducting dataflow analyses [30–32], a type of analysis aimed at gathering information about data values at various points in a program. This framework provides a systematic approach to performing dataflow analysis across an entire function. For example, Kildall presents a global value numbering algorithm integrated with constant propagation.

In this framework, the analysis operates over a CFG and a complete lattice of dataflow facts, where each CFG node is associated with a dataflow fact. The process involves applying a set of monotone functions, called transfer functions, to the CFG nodes until a fixed point is reached. A transfer function takes an element from the lattice of dataflow facts and a CFG node, returning a new element from the lattice. Kildall's framework is known as a monotone framework [33], requiring the transfer functions to be monotonic with respect to the lattice of dataflow facts. This, combined with the lattice's completeness (each subset having a supremum and an infimum), ensures that the analysis within this framework always converges to a fixed point.

To illustrate this framework, Kildall describes a global value numbering algorithm integrated with constant propagation. In this scenario, the dataflow fact associated with each node is a partition of expressions, where expressions in the same congruence class are considered equivalent at that node. This approach is similar to value numbering combined with constant propagation, as some partitioned expressions represent program variables and constants. Kildall explains the transfer function as a restructuring of the partition. Restructuring involves adding expressions occurring at the current node and their equivalents, discovered by replacing operands with their known equivalents. When a node has multiple incoming predecessors, a class-wise intersection of the incoming partitions is performed. This intersection allows us to retain information that is valid across all incoming paths.

In Figure 3.3, we revisit the example from Figure 3.2, now applying Kildall's algorithm. We are able to discover that at the return statement, the value of `c` is 25.

Kildall's algorithm is very powerful. Although the equivalence of expressions in general is undecidable, Kildall's algorithm is complete with respect to Herbrand equivalences. Two expressions are Herbrand equivalent if they have the same operator and Herbrand equivalent operands. These expressions can also be described as being *transparently equivalent*. However, the Herbrand completeness of Kildall's algorithm comes at the cost of exponential worst-case complexity [34]. This exponential behavior arises from the exponential growth of the classes of partitions, as more combinations of expressions become possible as the algorithm progresses. Another source of complexity is the merging of partitions. Intersecting two classes, c_1 and c_2, has a worst-case complexity of $\Omega(|c_1| \cdot |c_2|)$. Combined with the fact that we need $O(|p|)$ class intersections for a partition p, this results in an overall exponential worst-case complexity.

Following Kildall's work, much of the focus on value numbering has been on achieving Herbrand

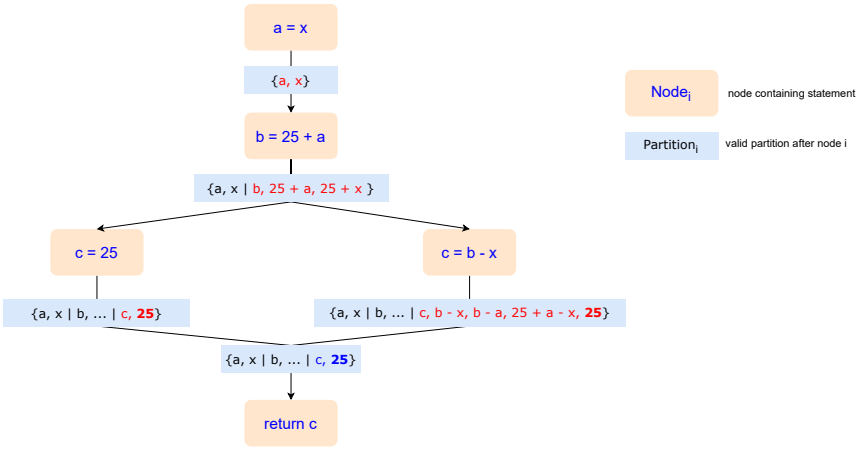


Figure 3.3: Application of Kildall's algorithm: A partition consists of sets of equivalent expressions. In each partition, we highlight the expressions derived from the previous node.

completeness more efficiently. To this end, various forms of global value numbering have been proposed over the years, concentrating mainly on efficient representations of the abstract state.

3.3 Optimistic partitioning algorithms

Alpern, Wegman, and Zadeck (AWZ) introduced an optimistic global value numbering algorithm designed for programs in SSA form [6]. The algorithm begins with an optimistic partition of the program's variables into congruence classes, where variables defined by the same operator are initially considered congruent. In each step, variables in the same congruence class but with non-congruent operands are separated into different congruence classes. This process repeats until a stable partition is achieved, meaning all congruent variables share the same operator and have congruent operands.

This algorithm features two key innovations. First, it operates over a value graph. The value graph is composed of nodes containing operators, constants, or function arguments. The nodes of the graph are connected through directed edges $\langle a, b \rangle$, representing that a uses b or that b is an operand of a. Two instructions are congruent if their corresponding nodes in the value graph have the same operator and their operands are also congruent. This approach is much more manageable than Kildall's structured partition, as the size and number of congruence classes is at most equal to the number of instructions.

The other innovation is its optimistic assumption, which is key to discovering cyclic equivalences. In Figure 3.4, we have a program in SSA form where a and b are redundant. If a and b had started in different congruence classes, a_1 and b_1 would have been deemed non-congruent, leading to the conclusion that a and b are also non-congruent. To discover cyclic equivalences, we must assume them from the start. Note, however, that this assumption is sound since the algorithm only terminates when it reaches a Herbrand-stable partition. The execution of this example is illustrated in Figure 3.5, where

```
a0 = 1;
b0 = 1;
c0 = 2;
while(...) {
  a = φ(a0, a1);
  b = φ(b0, b1);
  c = φ(c0, c1);
  a1 = a + a0;
  b1 = b + b0;
  c1 = c + c0;
}
```

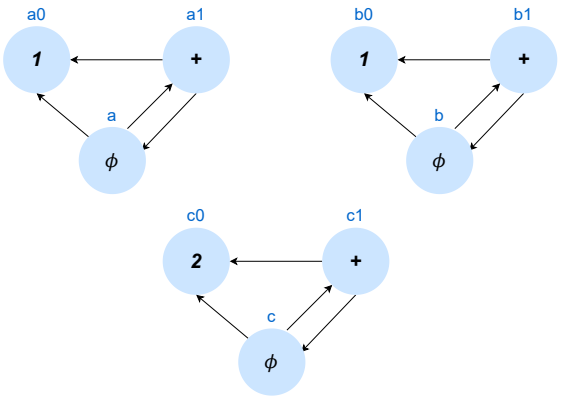


Figure 3.4: Program with cyclic equivalence (on the right) and the corresponding value graph (on the left). For simplicity, we assume that phi instructions in different basic blocks have distinct operators, thereby preventing phi instructions from different basic blocks from being considered equivalent.

all instructions with the same operator start in the same class. At each step, the algorithm checks for inconsistent classes, i.e., instructions in the same class but with non-congruent operands. In particular, we have a trivially inconsistent class $a0, b0, c0$ since $c0 = 2$ but $a0 = b0 = 1$. The corrections are propagated until the partition becomes consistent.

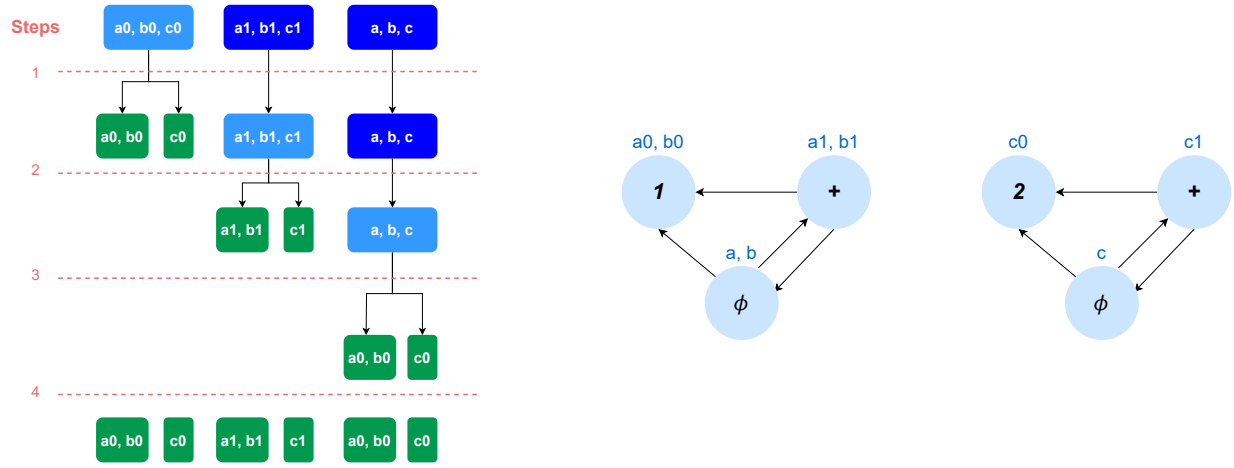


Figure 3.5: Application of the AWZ algorithm to the program of Figure 3.4. On the left, the initial optimistic partition is refined step-by-step. On the right, the collapsed value graph after partitioning. In the collapsed value graph congruent nodes have been aggregated.

Rüthing, Knoop, and Steffen [34] observed that the main limitation of the AWZ algorithm is that it treats phi instructions as uninterpreted functions. This results in a loss of precision, even in acyclic code, as illustrated in Example 3.6. In this example, `y` and `x2` are equivalent; however, since these variables are defined by different operators, the standard AWZ algorithm fails to discover this equivalence.

```
if {
  a0 = 1;
  x0 = a0 + 1;
} else {
  a1 = 2
  x1 = a1 + 1;
}
a2 = φ(a0, a1);
x2 = φ(x0, x1);
y = a2 + 1;
```

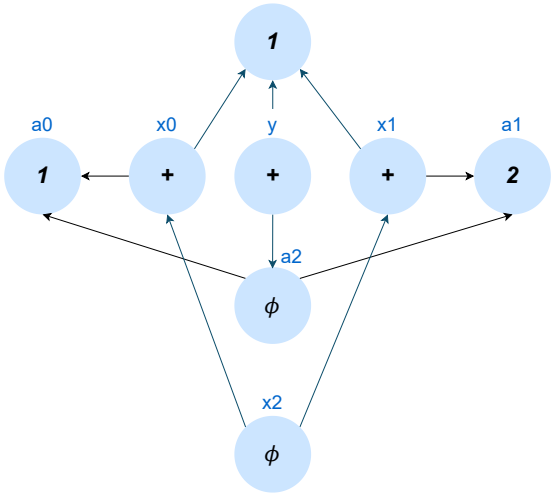


Figure 3.6: Program where the AWZ algorithm is unable to detect the redundancy (left) and the corresponding value graph (right).

This type of equivalence is hidden behind the phi instructions. To expose it, Rüthing, Knoop, and Steffen proposed an extension to the AWZ algorithm that incorporates normalization of the value graphs using the following rewrite rules:

$$\phi(a, a) \Rightarrow a \tag{3.1}$$

$$\phi(a_1 \ \mathbf{op} \ b_1, a_2 \ \mathbf{op} \ b_2) \Rightarrow \phi(a_1, a_2) \ \mathbf{op} \ \phi(b_1, b_2) \tag{3.2}$$

Rule 3.1 removes redundant phi instructions, while Rule 3.2 rewrites expressions containing phi operators to ensure that these operators are innermost. Revisiting Example 3.6 with the rewrite rules applied, we can now discover the redundancy:

$$\begin{aligned} x2 = \phi(x0, x1) = \phi(a0 + 1, a1 + 1) &\Rightarrow^{3.2} \\ \phi(a0, a1) + \phi(1, 1) &\Rightarrow^{3.1} \\ \phi(a0, a1) + 1 = a2 + 1 = y & \end{aligned}$$

3.4 Mitigating the Exponential Growth of Kildall's Algorithm

Gulwani and Necula introduced an algorithm based on random interpretation [35]. This approach arises from the recognition that the AWZ algorithm is not Herbrand complete because it treats phi functions as uninterpreted. A fully interpreted approach, similar to Kildall's algorithm, results in exponential worst-case complexity. To address this, they propose an algorithm that uses randomized inputs and a random

interpretation of branching and phi instructions. This randomized approach yields a polynomial-time algorithm. However, the drawback is its lack of soundness, as there is a small probability of the algorithm discovering false equivalences, which is unacceptable in the context of compiler optimization.

Later, Gulwani and Necula [36] observed that the exponential nature of Kildall's algorithm arises from its goal of discovering all Herbrand equivalences. Instead, they suggest limiting the search to Herbrand equivalences among program sub-expressions. This can be achieved by performing partition restructuring only up to the number of instructions in the function, as this represents the maximum size an expression can be.

Saleena and Paleri proposed a representation for redundant expressions that ensures linearly sized partitions [37]. They use the concept of value expressions, where the operands are value numbers. The advantage of value expressions is their ability to represent multiple expressions succinctly. Their algorithm addresses the problem of exponentially sized partitions; however, merging partitions remains costly.

An algorithm with more efficient partition merging was proposed by Pai [38]. For Kildall's algorithm to achieve completeness, it must identify redundancies at a program point p for later use at a subsequent program point p'. Pai's algorithm adopts a different strategy: at program point p, given an expression e, it checks for equivalent expressions in the paths leading to p. This detection can be efficiently accomplished by leveraging the SSA form and the semantics of the phi instruction.

3.5 Combining GVN with other analyses

Value numbering can be unified with other analyses to achieve stronger results [12]. By unifying analyses, we can produce more robust outcomes compared to applying them sequentially. This is because there might be feedback loops between the analyses. For example, unifying value numbering with algebraic simplification can reveal more equivalences between instructions, leading to additional opportunities for simplification.

Cooper and Simpson [11] noted that hashing and partitioning approaches do not identify the same redundancies in a given program. This discrepancy arises because hashing relies on expression values, making it more compatible with algebraic simplification. Conversely, the partitioning algorithm relies on congruency, which makes it less adaptable to algebraic simplification but suitable for applying the optimistic assumption. Cooper and Simpson illustrate this discrepancy using the examples in Figure 3.7. On the left, we have a program improved by a hashing approach that leverages algebraic simplification. Assuming that `X` and `Y` are equivalent, then `A, B, C, D` are all equal to zero. The hash-based GVN can discover this. Meanwhile, a standard partitioning algorithm places each variable in its own congruence class.

```
A = X - Y
B = Y - X
C = A - B
D = B - A
```

```
X0 = 1
Y0 = 1
while (...) {
    X1 = φ (X0, X2)
    Y1 = φ (Y0, Y2)
    X2 = X1 + 1
    Y2 = Y1 + 1
}
```

Figure 3.7: On the left is a program improved by hash-based GVN, while on the right is a program improved by a partitioning-based GVN.

On the right, we have a program improved by a partitioning algorithm. Detecting the cyclic congruences (`X1` with `Y1` and `X2` with `Y2`) requires assuming them from the start, which is facilitated by the optimistic assumption inherent in the partitioning algorithm.

They introduce an algorithm for optimistic global value numbering, using a hash-based approach which easily integrates with algebraic simplification. The design of the algorithm stems from the fact that in SSA the optimistic assumption is only relevant for cycles. Therefore the algorithm operates on the strongly connected components (SCC) of the value graph. Tarjan's DFS-based SCC finding algorithm [39] is used. It uses a stack to find the SCCs. Nodes that are popped simultaneously from the stack are known to be part of an SCC. In the context of an SSA graph, when an SCC is popped, we know that operands of instructions within the SCC are either part of the SCC or have already been popped.

Upon popping an SCC, its instructions are value-numbered in reverse post-order until the value numbers stabilize. Notably, singleton SCCs always achieve stabilization in one step.

Given the hash-based nature of the value-numbering step, this algorithm easily integrates with algebraic simplification. Furthermore, it capitalizes on the optimistic assumption by assigning instructions the initial value number of $\top$, which is disregarded when evaluating phi instructions.

```
a0 = 0;
while(...) {
    a1 = φ (a0, a2);
    c = x ^ a1;
    a2 = x - c;
}
```

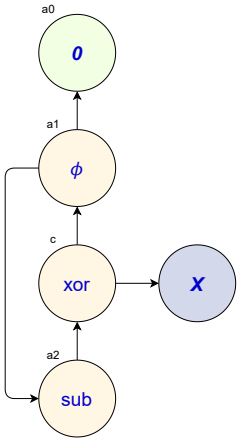


Figure 3.8: Example: Application of the SCC-based value numbering algorithm.

We illustrate the application of this algorithm using Figure 3.8. On the left, we have a program in SSA form. On the right, the corresponding SSA graph where the different colored nodes denote its SCCs. Depending on the starting node for Tarjan's algorithm, either the singleton with variable `a0` or the singleton with the function argument `x` will be popped first. Either way, processing singletons is done in a single step: `a0` is value-numbered with constant `0`, and `x` is value-numbered with the abstract value $\top$, denoting that we do not know its value.

Afterwards, the non-trivial SCC is processed. Processing starts at variable `a1` due to RPO. Since `a2` has not been processed, it has the optimistic value $\top$, which is ignored when evaluating the phi. Therefore, the variable is value-numbered to 0. Variable `c` is value-numbered to `x` due to algebraic simplification, and consequently, `a2` is value-numbered to 0. The process is repeated for the entire SCC to check if a fixed point has been reached. Since the value numbers do not change, it terminates.

This algorithm uncovers as many redundancies as a hash-based approach, as it is itself hash-based. It also identifies just as many redundancies as partitioning-based approaches. All of this is efficiently achieved by operating over the SCCs of the SSA graph.

Gargi introduced a set of practical and powerful algorithms for global value numbering [8]. These algorithms unify value numbering with algebraic simplification and unreachable code elimination. Additionally, they perform global reassociation and can derive equivalencies using information from predicates. This is achieved efficiently using a sparse formulation, where only the touched instructions are processed in each pass. If an instruction changes congruence class, all users of that instruction are also touched. This approach is similar to a worklist implementation used in dataflow analysis.

Another innovation by Gargi is the introduction of balanced value numbering. Balanced value numbering aims to combine the speed of a pessimistic algorithm with the precision of an optimistic one. An optimistic algorithm can discover more equivalences by ignoring unreachable edges, including edges speculated to be unreachable. However, optimistic algorithms require multiple passes over the function—up to the loop connectedness of the function—to verify speculated edges. In contrast, a balanced algorithm only ignores edges known to be unreachable, allowing it to discover unreachable code and leading to more simplification opportunities and equivalences. Since no speculation is involved, the balanced algorithm can terminate in one step, similar to the pessimistic algorithm.

3.6 Redundancy Elimination

Up to this point, we have explored how to identify redundancies in a program. In this section, our attention shifts to how to eliminate the identified redundancies.

Dominator-based elimination achieves redundancy elimination by using the dominance relation [29]. It iterates through the congruence classes of a given partition p and substitutes each member of the

class c with a dominating leader. A dominating leader for a specific register is a member of the class that either resides in a dominating block or appears earlier in the same block. This can be efficiently achieved by visiting the members of the class using the order given by a DFS traversal of the dominator tree (DFSDT), for registers in the same basic block the order is given by increasing order within the block.

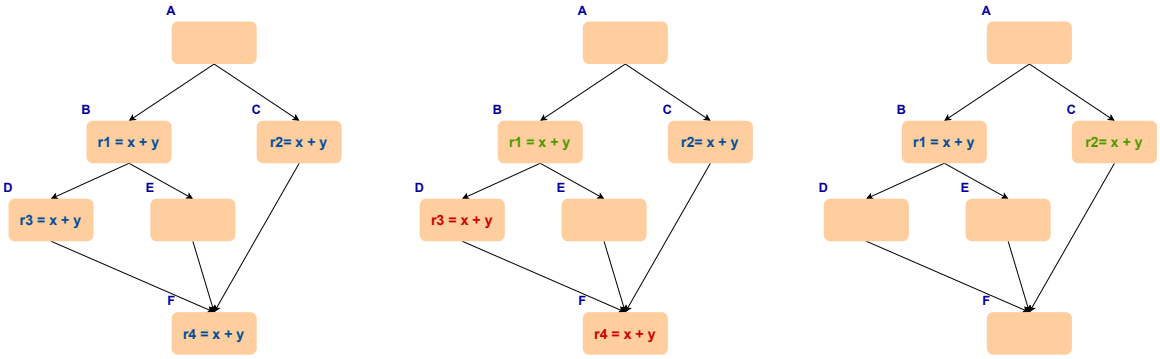


Figure 3.9: Application of dominator-based elimination. All r_i variables are equivalent. In each step a leader(•) is elected (mininum of DFSDT order) and every dominated(•) r_i is replaced with the leader.

3.6.1 Partial-Redundancy Elimination

Partial-redundancy elimination (PRE) [40, 41] is a whole-function optimization that aims to remove expressions that are redundant in some but not all execution paths. In other words, it ensures that each expression is computed at most once in each possible execution path. From this definition it follows that PRE subsumes loop-invariant code motion (LICM) since having an invariant expression inside a loop means that the expression is computed in multiple execution paths.

In the example below, we have a partial redundancy (left hand-side) and its removal (right hand-side). In the original program, the expression was computed twice when the branching condition was true, and once when it was false. PRE copies the partially redundant instruction to the `else` block making the instruction after the `if-else` block fully-redundant enabling its removal. Now the expression is computed only once in each path.

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

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

PRE algorithms aim to meet four criteria [14]: correctness, safety, computational optimality, and lifetime optimality.

Correctness For programs in SSA form, PRE is considered correct if, after insertions and deletions, the program still adheres to SSA properties; i.e., each variable is defined exactly once, and each definition dominates all its uses. Additionally the definitions that reach each use should be equivalent to the ones in the original program.

Safety A value inserted by PRE is safe if, in all paths from the insertion point to an exit node, the value is used. In SSA form, there are two ways in which a correct insertion can be unsafe. Either the original instruction was dead (i.e., it had no uses), or we are trying to insert through a critical edge, a concept that we now explain.

In Figure 3.10, on the left, we see 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 introduces redundant computation in other paths. The issue arises from the edge highlighted in red. This edge is a critical edge: its origin has multiple successors, and its target has multiple predecessors. To ensure that PRE does not introduce redundant computation, critical edges must be split. Splitting an edge involves adding an intermediate basic block, where we then insert the instruction.

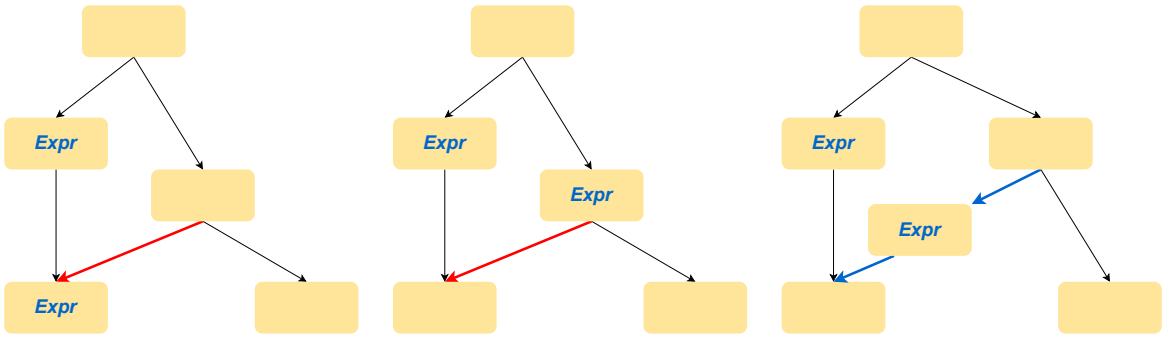


Figure 3.10: Example: PRE with critical edge splitting.

Computational Optimality A PRE algorithm is computationally optimal if it minimizes the number of computations of a value in any given execution path. For example, in Figure 3.11, on the left we have a partial redundancy, and in the center and right we have two applications of PRE. In the program in the middle, the expression is computed twice along one of the execution paths. In contrast, in the program on the right, the expression is computed once across all execution paths. Therefore, the latter is computationally optimal, while the former is not.

Lifetime Optimality A PRE algorithm meets the lifetime optimality criteria if, assuming a computationally optimal insertion, the lifetime of the inserted instruction is minimized. In other words, we aim to delay

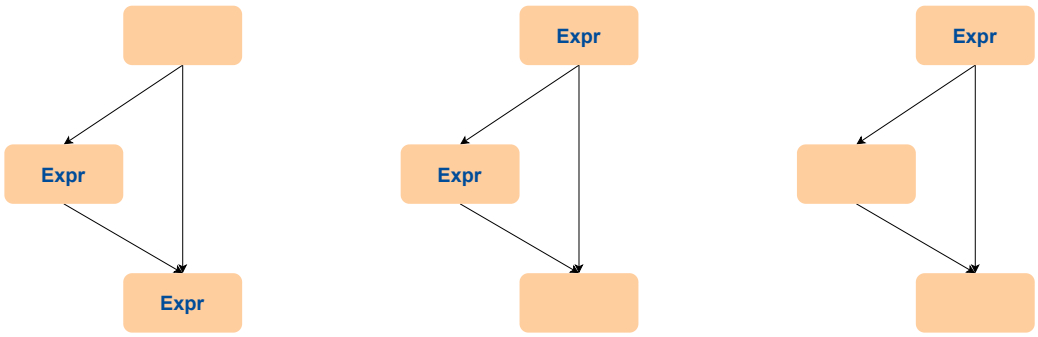


Figure 3.11: Computationally non-optimal (center) and computationally optimal (right)

the insertion for as long as possible to reduce register pressure.

3.6.2 Value Based Partial-Redundancy Elimination

PRE becomes more powerful when combined with value numbering [42–44]. In the previous example, redundancies were only based on syntatic equality, meaning they stemmed from identical instructions. However, when integrated with value numbering, PRE can eliminate redundancies not only based on syntatic equality but also on semantic equivalence, determined by the assigned value numbers.

Let us analyze another example (below), where a partial redundancy is obscured by the fact that `x + 42` and `b - y + 42` are equivalent.

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

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

In an algorithm that unifies PRE and GVN, GVN can reveal this equivalence, allowing PRE to subsequently eliminate the redundancy. Note also that if this was the only use of variable `b` then it becomes dead code and can also be eliminated from the function.

4

Global Value Numbering for Modern Compilers

Contents

In this chapter, we introduce our algorithm for Global Value Numbering. First, we define an intermediate representation (IR) over which the algorithm will operate. This IR is inspired by LLVM's SSA-based IR, incorporating simplifications for clarity while remaining sufficiently expressive. Like LLVM, our IR is a typed, assembly-like language with low-level operations over an infinite set of virtual registers. Additionally, we extend our IR to support both SSI and MemorySSA. Our algorithm integrates algebraic

$$
\begin{aligned}
function &::= (\mathcal{F}_a, block^*) \\
\mathcal{F}_a &::= reg^* \\
block &::= (label, \langle stmt_n \mid n \in \mathbb{N} \rangle) \\
stmt &::= reg = expr \mid term \\
term &::= \mathbf{jmp}\ label \mid \mathbf{br}\ ty\ value, label_t, label_f \mid ret[ty\ v] \\
expr &::= ty\ value \mid op\ [flags]\ ty\ value_1, ..., value_n \mid \boldsymbol{\phi}\ ty\ reg_1, \ldots, reg_n \mid \boldsymbol{\pi}\ ty\ (reg, reg_{pred}, const) \\
value &::= const \mid reg \mid poison \\
ty &::= iN \mid\ < N_1 \times iN_2 > \ \mid ptr \mid mem,\ N, N_1, N_2 \in \mathbb{N}
\end{aligned}
$$

Figure 4.1: IR Syntax.

simplification and unreachable code elimination. Furthermore, it leverages the SSI and MemorySSA extensions to utilize information from predicates and to identify equivalences involving memory operations.

4.1 Syntax

Since we are developing an intraprocedural algorithm, which analyzes and optimizes computations within a single function, we use functions as the top-level structure. This approach simplifies implementation and analysis but has limited optimization potential, as it cannot exploit relationships between functions. In contrast, an interprocedural algorithm considers multiple functions and their interactions for optimization [45], allowing for more substantial performance gains but requiring more complex implementations and increased analysis time. Functions are represented as control-flow graphs (CFGs) along with a set of function arguments.

The blocks that make up a CFG are sequences of program statements. Program statements consist of two types: terminators and non-terminators. Terminators direct control flow to other blocks, while non-terminators represent assignments of expressions to registers. An expression can be a constant, a register, an arithmetic expression, a memory operation, or a function call. Additionally, we have the ϕ and π pseudo-instructions, used for SSA and SSI, respectively.

4.1.1 Assumptions About the IR

We assume that phi expressions are in canonical form, which means the operands are ordered according to the basic blocks from which they originate and follow a consistent structure. This ordering simplifies the representation of phi expressions, as it is not necessary to represent the predecessors explicitly. Operands of phi expressions are always registers, which simplifies tracking optimistic dependencies—more on that later. Additionally, all functions include an argument representing the initial

memory state. This memory state is either undefined or defined by the caller. The figure below provides an example of a conventional SSA-based IR and its equivalent representation where phi nodes are in canonical form.

```
entry:
  jmp loop

loop:
  a = φ i32 [0, entry], [1, loop]
  ...
  br ..., loop, exit
exit:
  ...
```

```
entry:
    a.entry = i32 0
    jmp loop

  loop: // preds = [entry, loop]
    a = φ i32 (a.entry, a.loop)
    ...
    a.loop = i32 1
    br ..., loop, exit
  exit:
    ...
```

Figure 4.2: On the left is an example program in a conventional SSA-based IR. On the right is an equivalent program where phi nodes are in canonical form, meaning that each entry does not require a specification of the predecessor, and the phi operands are always registers.

4.2 Modeling the Optimistic Assumption

To a large extent, we can ensure that the operands of an expression are processed before the expression itself by using a Reverse Postorder (RPO) traversal method. However, there is one exception: the operands of phi expressions that are carried by retreating edges—edges in a control flow graph that lead back to an earlier block, indicating a loop. Optimistic GVN algorithms address this issue by either ignoring these operands or treating them as equivalent to other operands. Once we evaluate the ignored operand, we revisit the corresponding phi expression. This requirement for phi operands to always be registers ensures that the dependencies are clearly represented in the IR.

To model this optimistic assumption, we initially treat all registers as equivalent by beginning their evaluation with the abstract value $\perp$. This value is disregarded during the simplification of expressions. In the context of RPO, this situation arises only if the block where the operand is defined is unreachable. Additionally, in the case of edge uses (i.e., uses occurring at phi expressions), this occurs if the corresponding edge is unreachable. It is crucial to note that this unreachability can either be assumed based on the optimistic assumption or confirmed through analysis.

4.3 Computing Value Numbers and CFG Reachability

In our framework, value numbers are represented by canonical expressions. A canonical expression identifies a set of equivalent registers, known as a congruence class. The canonical expression of a register is computed based on the canonical expressions of its operands. The process involves taking the

register's defining expression and, using known equivalences and algebraic simplifications, constructing a canonical version that is equivalent to the original. Since we operate within the SSA principles, each expression corresponds to a unique static value. Therefore, registers that share the same canonical expression are equivalent.

4.3.1 Value Numbers

Value numbers are computed only for non-terminators, as we assume that terminators do not produce values. For n-ary operator-defined expressions, value numbering involves replacing each operand with its corresponding canonical expression and then simplifying the resulting expression.

In the following example given that $b = a \times z$ we compute b's value number using the known value numbers presented on the left (the notation $r \to e$ represents that register r has value number e and this mapping is given by $\mathcal{V}$):

Known Value Numbers

$$x \to x$$
$$y \to y$$
$$z \to 1$$
$$a \to x + y$$

Computation of b's Value Number

$$\begin{aligned}\mathcal{V}[b] &= simpl(\mathcal{V}[a] \times \mathcal{V}[z]) \\ &= simpl((x + y) \times 1) \text{ (identity of multiplication)} \\ &= x + y\end{aligned}$$

For phi expressions, we ignore values incoming from unreachable predecessors. If the remaining values are equivalent to each other, then the phi is also equivalent to those values. To achieve this, we define a lattice of program expressions, with $\bot$ as its least element and $\top$ as its greatest. The abstract expression $\bot$ symbolizes that the register is unreachable, either speculatively, due to optimistic assumptions, or definitively. Meanwhile, the abstract value $\top$ symbolizes that the register has multiple defining expressions depending on the path that reaches it.

To compute the value number, we perform a join ($\sqcup$) of the value numbers of the phi operands. Unreachable values are ignored in this operation, as they act as the identity element for the join. If this process results in a single value number, it indicates that all reaching definitions are equivalent, and the expression evaluates to the canonical version of that definition. Conversely, if the join yields $\top$, it means that the phi has multiple reaching definitions and thus cannot be simplified. In this case, the canonical expression is the phi expression.

In the following example, given that $a = \phi(x, y, z)$, we compute a's value number using the known value numbers presented on the left:

Known Value Numbers

$$x \to x$$
$$y \to x$$
$$z \to \bot$$

Computation of x's Value Number

$$\begin{aligned}\mathcal{V}[x] \sqcup \mathcal{V}[y] \sqcup \mathcal{V}[z] &= x \sqcup x \sqcup \bot \\ &= \{x\}\end{aligned}$$

$$\mathcal{V}[a] = x$$

The following is another example of computing the value number of the same phi instruction; however, this time, all operands are reachable:

Known Value Numbers

$$x \to x$$
$$y \to x$$
$$z \to z$$

Computation of a's Value Number

$$\begin{aligned}\mathcal{V}[x] \sqcup \mathcal{V}[y] \sqcup \mathcal{V}[z] &= x \sqcup x \sqcup z \\ &= \top\end{aligned}$$

$$\mathcal{V}[a] = \phi(x, y, z)$$

For pi expressions, we simplify by using predicate information only if the predicate is an equality. It is possible to use other types of predicates, but for the sake of simplicity, we restrict it to equalities. In such cases, the value number corresponds to the first operand of the canonical expression for the predicate. We define a partial order among all possible values, where constants precede registers, and registers are ordered based on Reverse Postorder (RPO) traversal. This order ensures consistent and predictable comparison during expression simplification. A total order for constants is unnecessary because any expression composed solely of constants can always be simplified through constant folding. The combination of using the partial order to construct the canonical expression and always using the first operand of this expression, ensures that registers defined by pi instructions are always value number either to a constant or to the most dominant equivalent register.

In the following example, given the predicate $pred = a \neq x$ and the pi instruction $a.1 = \pi(a, pred, false)$, we compute the value numbers for $a.1$ and $pred$ using the known value numbers on the left:

Known Value Numbers

$$x \to 0$$
$$a \to y - z$$

Computation of Value Numbers

$$\begin{aligned}\mathcal{V}[pred] &= simpl(\mathcal{V}[a] \neq \mathcal{V}[x]) \\ &= simpl((y - z) \neq 0) \\ &= 0 \neq (y - z)\end{aligned}$$

$$\mathcal{V}[a.1] = 0$$

In this example, a, x and 0 are all equivalent to $a.i$. If we use a as the value number we lose the information provided by the predicate. Using x or 0 give us path-sensitive information, however it is preferable to use the constant for the purpose of simplification.

4.3.2 Reachability

Changes in reachability occur only at terminator program statements. For conditional branches, if the branch condition is known to be equivalent to a constant, we mark the corresponding outgoing edge as reachable. If the condition is unknown, both outgoing edges are marked as reachable. In the case of an unconditional branch (i.e., a jump), the outgoing edge is marked as reachable. Finally, for return statements, no edges are marked as reachable, as the control flow terminates.

It is important to note that edges and blocks can only become reachable and once they become reachable they always remain that way. This is important to ensure that loops are correctly evaluated. In lattice terms consider the reachability lattice where the least element is unreachable and the greatest element is reachable. The reachability of the entire function is represented using a map lattice that maps from the set of edges of the CFG to the reachability lattice. The evaluation of terminators must be monotonic with respect to the reachability map lattice.

4.3.3 Inference Rules

In Figure 4.3, we present the inference rules for both non-terminators and terminators. A partition p maps program registers to their value numbers (canonical expressions). When a fixed point is reached, the inverse of the partition p^{-1}, which maps canonical expressions to sets of registers, defines congruence classes. Registers in the same class are known to be equivalent. This approach works because we are in SSA form, where each canonical expression corresponds to a single unique value.

Non-terminators are evaluated using $\xrightarrow{eval}$, which takes as input a partition and a non-terminator program statement. Terminators are evaluated using $\xrightarrow{eval_t}$, which takes as input a partition and a terminator program statement. To access the canonical expression of a register for a given partition p, we use the function $[\![\cdot]\!]_p$.

Below is an example of the inference rules. A rule for non-terminators can be intuitively understood as: given a partition p and an expression $expr$, if the preconditions are valid, then $expr$ is equivalent to $canonical$ under p. Similarly, a rule for terminators can be read as: given a partition p and a terminator $term$, if the preconditions are valid, then each block in $reachable$ is reachable from the current basic block.

$$\textsc{Non-Terminator}\ \frac{\text{Preconditions}}{\langle \mathsf{p}, \mathsf{expr} \rangle \xrightarrow{\mathsf{eval}} \mathsf{canonical}} \qquad \textsc{Terminator}\ \frac{\text{Preconditions}}{\langle \mathsf{p}, \mathsf{term} \rangle \xrightarrow{\mathsf{eval_t}} \mathsf{reachable}}$$

Note how every rule for terminators in Figure 4.3 either adds blocks to the reachable set or does nothing, this ensures the monotonicity requirement for reachability.

In summary, these inference rules define how non-terminator expressions are simplified to their

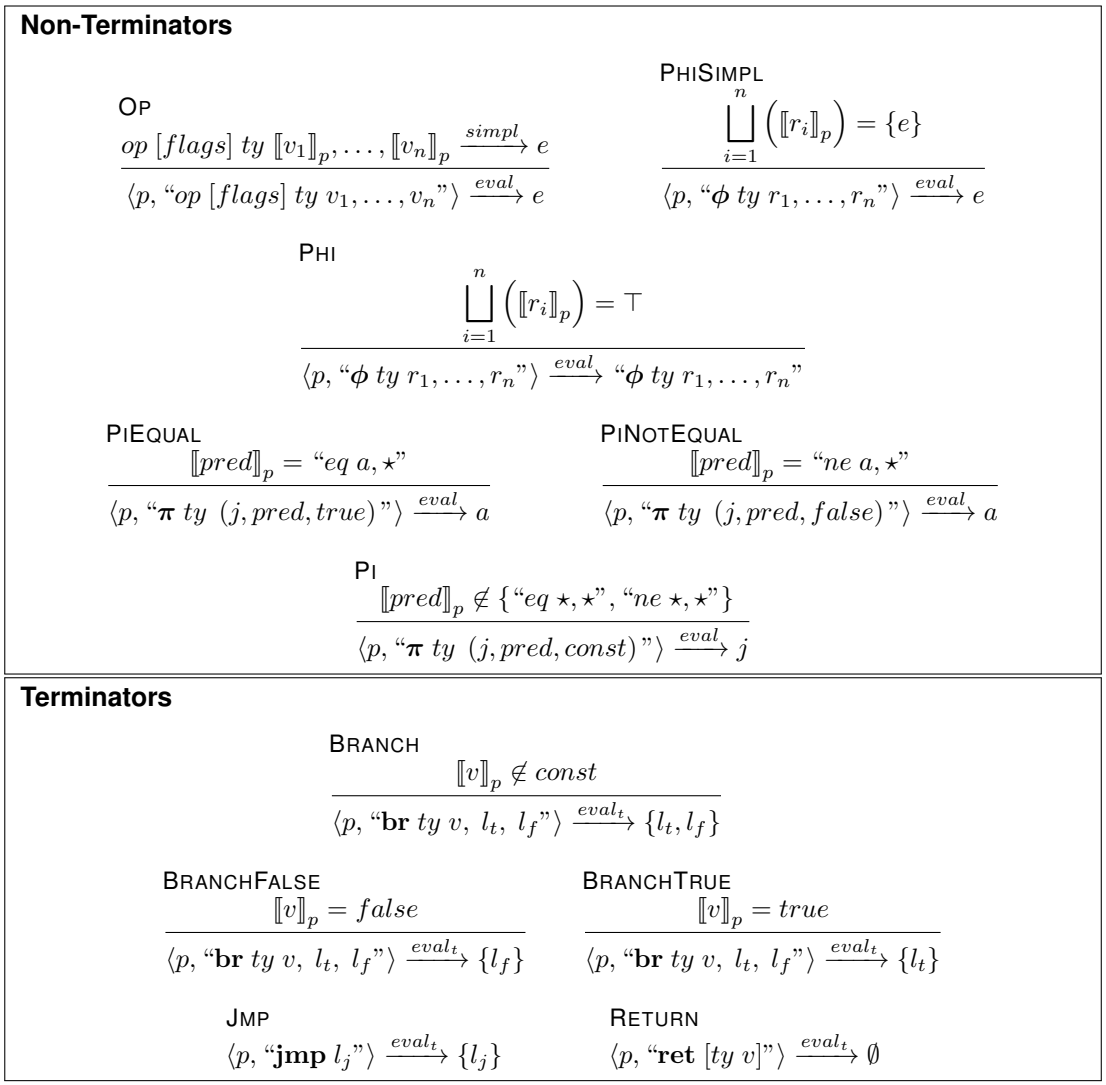


Figure 4.3: Inference rules to compute value numbers (non-terminators) and CFG reachability (terminators).

canonical forms, while terminator rules determine the reachability of basic blocks within the control flow graph. By applying these rules, the program's control flow and value numbers can be systematically computed.

4.3.4 Full Redundancy Elimination

The rules from the previous sections allow us to detect redundancies where, for an expression e, there exists an expression e' such that its computation dominates e and they are equivalent. However, there is another type of redundancy where, instead of a single e' that is fully equivalent to e regardless of the path taken, there are multiple equivalent expressions e_i, each corresponding to a different path i that reaches e.

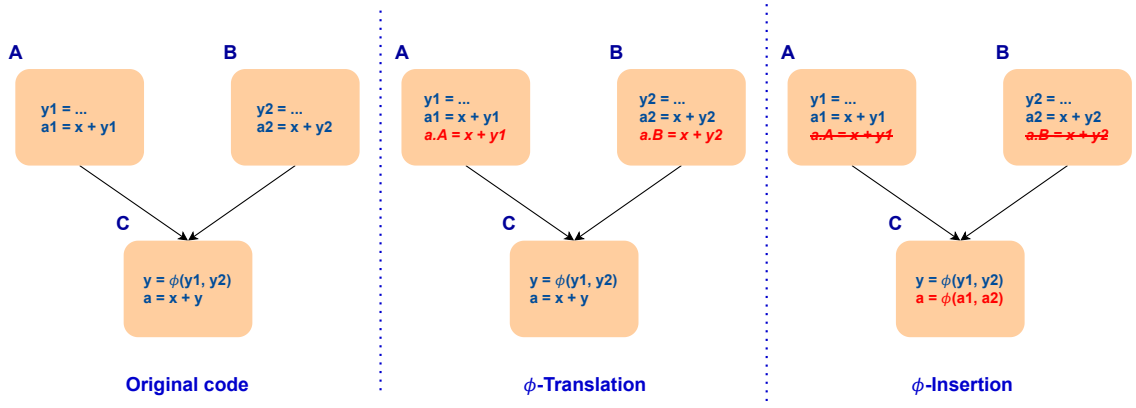


Figure 4.4: Example: Identifying and eliminating path-specific redundancies.

To detect this second type of redundancy, we employ a method based on Pai's algorithm [38]. The key idea is to find an equivalent instruction along each reaching path and then replace the redundant instruction using a phi function composed of these path-equivalent instructions. This method consists of two main steps: phi translation and phi insertion, which we will now explain.

Phi Translation Given an expression e at a basic block b, we start by translating e to each of the predecessors of b. The aim of translating an expression is to produce an expression equivalent to the original one, but only using values available in the desired predecessor. This process consists of recursively translating the expression until all operands of the expression dominate the predecessor. For example given the expression $1 + 2$, since constants are always available its translation is trivially itself regardless of the predecessor to which we are translating. A more interesting example is with expression $x + 1$. Once again the constant is trivially translated. Handling x however requires more care. If the computation of x dominates the predecessor then the translation can stop. Otherwise we must translate x's expression. For example if $x = y + 2$ then we must translate its operands y and 2. There is however another point of concern which are phi instructions. The following example illustrates the problem:

```
entry:
    ...
p1:
    a1 = x + y
p2:
    a2 = x - y
merge: // preds = [p1, p2]
    a = φ(a1, a2)
    b = a + 1
    ...
```

Translating `a` to `p1` as outlined so far yields $\phi([(x + y)], [x - y]) + 1$. This expression is nonsensical in the context of `p1` because the meaning of a phi expression is tied to the basic block where it resides.

During translation, when a phi is encountered, we should only translate the incoming value from the corresponding predecessor. Applying this to the previous example would produce the expression $a1+1$.

The following is a more complete example of a translation. The translation is performed by the function φ_{transl}, which receives a register and a basic block label. We will define it more formally later:

```
// preds = [p1, p2]
a = φ(a1, a2)
b = φ(b1, b2)
c = b + x
d = a + c;
```

$$\begin{aligned}\varphi_{transl}(d, p1) &= \varphi_{transl}(a, p1) + \varphi_{transl}(c, p1) \\ &= \varphi_{transl}(a1, p1) + (\varphi_{transl}(b, p1) + \varphi_{transl}(x, p1)) \\ &= a1 + (\varphi_{transl}(b1, p1) + x) \\ &= a1 + (b1 + x)\end{aligned}$$

In this example, the aim is to translate d to $p1$. Since d does not dominate $p1$, its operands are translated. Since a is a phi, we translate $a1$ from the incoming value in $p1$. As $a1$ dominates $p1$, its translation terminates here. On the other hand, c is not a phi, so we translate all its operands. Similarly, b is a phi in the starting block, so we translate $b1$, which dominates $p1$. Finally, we translate x, the other operand of c, which also dominates $p1$, completing the translation.

As with any recursive procedure, there is the concern of convergence. Consider the following example:

```
// preds = [p1, p2]
a = φ(a1, a2)
...
a2 = a + 1;
```

$$\begin{aligned}\varphi_{transl}(a2, p2) &= \varphi_{transl}(a, p2) + \varphi_{transl}(1, p2) \\ &= \varphi_{transl}(a2, p2) + 1 \\ &= \varphi_{transl}(a, p2) + \varphi_{transl}(1, p2) + 1 \\ &= ...\end{aligned}$$

In this example, translation will never converge due to the cyclic values a and $a2$. To prevent such cases, the phi translation returns the abstract value $\perp$ whenever it encounters cyclic values. This abstract value signifies an inability to complete the translation, effectively aborting the algorithm.

Phi Translation

$$\textsc{Dominates}\ \frac{\mathsf{dominates}(r, pred)}{\varphi_{transl}(r, pred) = r}$$

$$\textsc{Op}\ \frac{\neg\mathsf{dominates}(r, pred) \qquad r = \text{“}op\ [flags]\ ty\ v_1, \ldots, v_n\text{”} \qquad \bigwedge_{i=1}^{n} (v_{i'} = \varphi_{transl}(v_i, pred))}{\varphi_{transl}(r, pred) = op\ [flags]\ ty\ v_{1'}, \ldots, v_{n'}}$$

$$\textsc{Phi}\ \frac{\neg\mathsf{dominates}(r, pred) \qquad r = \text{“}\boldsymbol{\phi}\ ty\ (r_1, \ldots, r_n)\text{”} \qquad r_{pred} = [r_1, \ldots, r_n][pred]}{\varphi_{transl}(r, pred) = \varphi_{transl}(r_{pred}, pred)}$$

Figure 4.5: ϕ-Translation : Translating an expression to a basic block.

Availability After translation, we check if the translated expression is available in the predecessor (see Figure 4.6). Availability means whether an expression $expr$ has already been computed and can be reused at basic block $pred$. This is done by computing its value number and examining the corresponding congruence class (using partition p) to find a value that dominates the predecessor.

If the expression is fully redudannt, ie, its available in every reaching path, we can replace the original instruction with a phi expression composed of the path-available values, which we call phiofops (see Figure 4.7). We refer to this technique as full redundancy elimination (FRE). The entire process is illustrated in Figure 4.4.

Availability

AVAILABLE

$$\frac{\langle p, expr\rangle \xrightarrow{eval} expr' \qquad availVal \in p[expr'] \qquad \mathsf{dominates}(availVal, pred)}{\mathsf{available}(expr, pred, p) = availVal}$$

Figure 4.6: Availability : Checking if an expression is available in block.

Path Redundancy

PHIOFOPS

$$\frac{\bigwedge_{pred_i \in \mathsf{predecessors}(block)} (expr_i = \varphi_{transl}(\text{“}op\ [flags]\ ty\ v_1, \ldots, v_n\text{”}, pred_i) \wedge r_i = \mathsf{available}(expr_i, pred_i, p)) \qquad k = |\mathsf{predecessors}(block)| \qquad \langle p, \text{“}\boldsymbol{\phi}\ ty\ (r_1, \ldots, r_k)\text{”}\rangle \xrightarrow{eval} \mathsf{phiofops}}{\langle p, \text{“}op\ [flags]\ ty\ v_1, \ldots, v_n\text{”}, block\rangle \xrightarrow{eval} \mathsf{phiofops}}$$

Figure 4.7: ϕ-Insertion : Inserting a phi instruction that makes the value fully redundant.

4.3.5 Partial Redundancy Elimination

We now demonstrate how value-based Partial Redundancy Elimination (PRE) naturally arises from our existing framework. The key observation is that Full Redundancy Elimination (FRE) is a special case of PRE, where the value is available on all reaching paths. This approach is similar to the E-Path PRE proposed by Dhamdhere [46].

To support PRE, we extend the rule that checks for availability (see Figure 4.8). This extension allows us to insert an instruction that makes the expression available in the block if it is not already available.

The key point here is that insertions are only performed along paths where the value is not available; otherwise, we would be introducing redundant computation. Additionally, we limit the number of insertions to one. This means that if a partially redundant instruction requires more than one insertion to become fully redundant, the transformation is aborted to prevent an increase in code size.

It is important to note, however, that making these insertions is not always safe. The main concern is

Availability

AVAILABLE

$$\frac{\langle p, expr\rangle \xrightarrow{eval} expr' \qquad availVal \in p[expr'] \qquad \mathsf{dominates}(availVal, pred)}{\mathsf{available}(expr, pred, p) = availVal}$$

INSERTION

$$\frac{\langle p, expr\rangle \xrightarrow{eval} expr' \qquad \bigwedge_{r \in p[expr']} (\neg \mathsf{dominates}(r, pred)) \qquad \mathsf{Insert}(expr, pred) = r}{\mathsf{available}(expr, pred, p) = r}$$

Figure 4.8: Availability : Checking if an expression is available in a block. If it is not then try to insert it.

the speculative execution of unsafe code. When speculating, it is crucial to ensure that the instruction in question has no side effects and does not invoke undefined behavior. This includes avoiding operations like division by zero, loading from invalid pointers, or speculatively executing instructions like malloc or alloca, which could cause resource issues, such as memory leaks.

In conclusion, while partial redundancy elimination can improve performance by reducing redundant computations, speculative execution should be handled with care. Proper analysis is necessary to ensure that it does not introduce undefined behavior, unwanted side effects, or incorrect memory operations, all of which can lead to subtle, hard-to-detect bugs.

Correction The original instruction is replaced by a phi function made up of the available values from each reachable predecessor; therefore, it is equivalent to the original instruction. Additionally, the phi function is inserted at the beginning of the basic block of the original instruction, thus dominating the original instruction. Since the original instruction dominates all its uses, the transitivity of the dominance relation implies that the inserted phi instruction also dominates the uses of the original instruction.

Safety The only safety concern arises from critical edges. For simplicity, we assume there is a pass before GVN that splits critical edges. If splitting critical edges is not possible, an alternative approach is to simply avoid performing PRE in cases that would require critical edge splitting to be safe.

Computational Optimality Regarding computational optimality, although the PRE algorithm does not always perform optimal insertions, optimality is still achieved when the entire algorithm terminates. This is because non-optimal insertions are non-optimal only because they dominate equivalent instructions; thus, the value numbering algorithm will detect and eliminate this redundancy.

Lifetime Optimality Lifetime optimality is ensured because insertions are only performed at the end of the predecessor. Inserting it any later would likely violate SSA properties, while inserting it any earlier would be unnecessary, as the inserted instruction is only used at the successor.

4.4 Value Numbering of Memory Operations

So far, we have ignored memory instructions such as loads and stores. Since we are using MemorySSA, handling memory operations is largely seamless within our existing framework. This is because the memory version can be treated as any other value. However, there are cases that require special attention.

Dead Store Elimination Consider the following pair of instructions, annotated with their value numbers:

```
mem1 = store i32 v, ptr p, mem mem0 --> <store i32, v, p, mem0>
mem2 = store i32 v, ptr p, mem mem1 --> <store i32, v, p, mem1>
```

As it stands, GVN is unable to detect that the second store is redundant. When the lookup for the value number `<store i32, v, p, mem0>` fails, it indicates that the store is unique. Since it is unique, we should construct its value number using the memory version defined by it rather than the memory version it uses. This produces the value number `<store i32, v, p, mem1>`. Subsequently, when evaluating the second store, we are able to find the class defined by the previous store:

```
mem1 = store i32 v, ptr p, mem mem0 --> <store i32, v, p, mem1>
mem2 = store i32 v, ptr p, mem mem1 --> <store i32, v, p, mem1>
```

Store-Load Forwarding In the following example, GVN is unable to detect that the load is redundant because the value being loaded is already known from the store:

```
mem1 = store i32 v, ptr p, mem mem0 --> <store i32, v, p, mem1>
a = load i32, ptr p, mem mem1 --> <load i32, p, mem1>
b = sub i32 v, a --> <sub i32, v, a>
```

The difference in value numbers arises from the use of different operators. To address this, we introduce the concept of superclasses of equivalence. A superclass of equivalence consists of multiple equivalence classes and can be treated as a single class under certain conditions. In this context, we aim to treat classes of stores and loads with the same pointer, type, and memory operand as super-equivalent. In the example above, the load is dominated by a super-equivalent store, so we can use the stored value as the load's canonical expression. As a result, we would be able to simplify the computation at `b` to `0`.

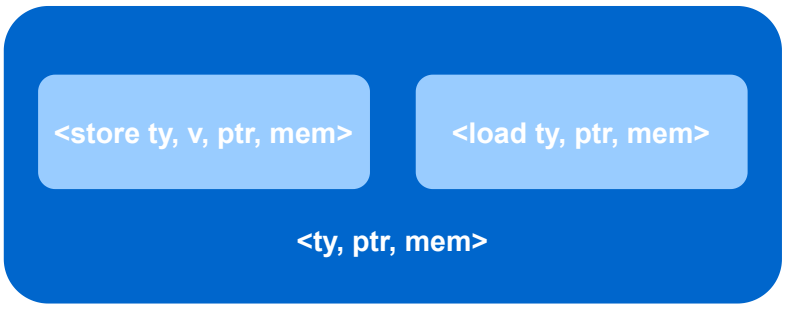


Figure 4.9: Load/Store superclass.

Value Coercion In the following example GVN fails to detect that the loads are redundant since the first one loads a superset of the bits loaded by the second one:

```
a = load i32, ptr p, mem mem1 --> <load i32, p, mem1>
b = load i16, ptr p, mem mem1 --> <load i16, p, mem1>
```

To support this case, we again leverage the concept of a superclass of equivalences. Both loads should be placed in the superclass identified by `<p, mem1>`. Note, however, that a hierarchical structure arises, meaning a superclass composed of other superclasses. For instance, to replace `b` from the previous example, we could use any dominating load/store with the same memory and type, with at least the same size as `i16`. This means the load could have been an `i64` or even a load of an aggregate type, such as `<2 × i8>`.

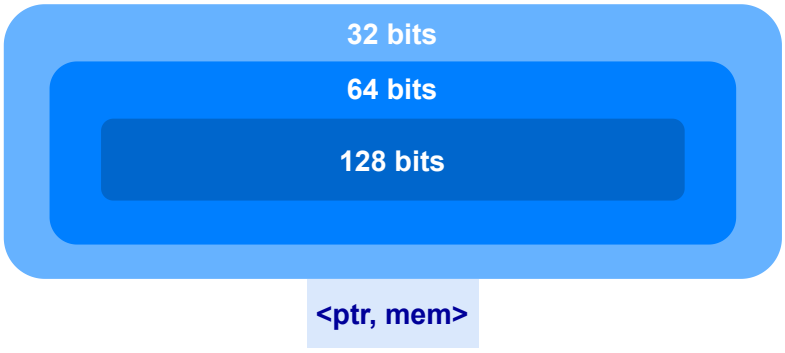


Figure 4.10: A superclass where the top level superclass contains all loads and stores at pointer ptr with memory mem with size of at least 32. Notice how the top level class has the smallest bitwidth.

GVN can now detect the redundant load. However, for the first load to replace the uses of the second one, its value must be coerced to the type of the second load. To achieve this, we can use a `trunc ... to` instruction:

```
a = load i32, ptr p, mem mem1 --> <load i32, p, mem1>
b = trunc i32 a to i16 --> <load i16, p, mem1>
```

Memory Operations

UNIQUE STORE

$$\frac{e = store\ ty\ [\![v]\!]_p, ptr\ [\![q]\!]_p, mem[\![m]\!]_p \qquad e \notin p}{\langle p, \text{“}store\ ty\ v, ptr\ q, mem\ m\text{”}\rangle \xrightarrow{eval} none}$$

REDUNDANT STORE

$$\frac{e = store\ ty\ [\![v]\!]_p, ptr\ [\![q]\!]_p, mem[\![m]\!]_p \qquad e \in p}{\langle p, \text{“}store\ ty\ v, ptr\ q, mem\ m\text{”}\rangle \xrightarrow{eval} e}$$

Figure 4.11: Value numbering of memory operations.

4.5 Fixed-point computation

In this section, we discuss how the fixed point is computed. We use a worklist implementation, where whenever an instruction changes its value number, all instructions that depend on it are added to the worklist. This approach helps prevent redundant work. We define the set of dependent instructions for an instruction as follows:

$$\mathsf{Dep}(s, \mathsf{visited}) = \begin{cases} \emptyset & s \in \mathsf{visited} \\ \mathsf{users}(s) \setminus visited \cup (\bigcup_{u \in \mathsf{users}(s)} \mathsf{Dep}(u, \mathsf{visited} \cup \{s\})) & \text{otherwise} \end{cases}$$

We keep a visited set to prevent infinite recursion. Therefore, with this definition of Dep, we are not at risk of missing simplification opportunities. Note that if the entire program forms a strongly connected component, this would imply that the entire program could be added to the worklist. To prevent adding an excessive number of statements to the worklist, it is possible to assume that simplification axioms (rules used to simplify expressions) have a finite height. In such a scenario, we would only add the users up to that specified height.

The following example illustrates the computation of the dependent registers of a register. In this case, we are computing it for `a`. The visited set is initially empty. The first step is to add the immediate users `b` and `c`. Then, the function is called recursively for the immediate users, with the visited set now containing `a`. Since `b` has no users, an empty set is returned. On the other hand, `c` has a user, `a`; however, because it has already been visited, it is not added, and the base case is reached.

```
entry:
    jmp loop
loop:
    a = φ i32 (x, c)
    b = sub i32 a, x
    c = add i32 a, 1
    br .., loop, exit
exit:
    ...
```

$$\begin{aligned} \mathsf{Dep}(a, \emptyset) &= (\mathsf{users}(a) \setminus \emptyset) \cup \mathsf{Dep}(b, \{a\}) \cup \mathsf{Dep}(c, \{a\}) \\ &= \{b, c\} \cup \emptyset \cup \mathsf{Dep}(c, \{a\}) \\ &= \{b, c\} \cup (\mathsf{users}(c) \setminus \{a\}) \cup \mathsf{Dep}(a, \{a, c\}) \\ &= \{b, c\} \cup \emptyset \cup \emptyset \\ &= \{b, c\} \end{aligned}$$

RPO Retrieval We now have a worklist containing the program statements that require processing. However, we have not defined the order in which we retrieve them. We use Reverse Post Order (RPO) with increasing order within the same basic block. This choice ensures that a definition is processed before its uses, following the SSA principle where each variable has a single definition dominating all its

uses. This approach is advantageous because, except for back edges, all dependencies of a register are processed before the register itself, leading to faster convergence to a fixed point.

Worklist formulation We now present the full algorithm in 4.1. Initializing the partition requires two steps. First singleton classes are created for each of the function arguments, since we want to treat them as unique values. Next a congruence class with all other registers is created. This class is identified by the abstract expression $\perp$, this represents that all these registers are equivalent due to being unreachable. The worklist initially only contains instructions from the function's entry block. Finally to conclude the initialization we create the set of reachable blocks which optimistically only contains the entry block. The three previous steps give the algorithm its optimistic nature.

The main loop occurs until the worklist is empty. In each iteration we retrieve the minimum instruction as given by RPO. We start by checking if the instruction is reachable by checking if the block where it resides is part of the set of reachable blocks. If the instruction is reachable then we process it. In case of terminator instructions, we process it using the $eval_t$ function, updating the set of reachable blocks in the process. Otherwise it is a non-terminator so it is processed using the $eval$ function producing its canonical expression. The partition is update accordingly. In case changes occurred to the partition, the instructions dependent on the current one are added to the worklist.

Algorithm 4.1 Optimistic GVN

```
1:  procedure GVN-PARTITIONING(function F)
2:      partition := {arg ↦ arg | arg ∈ F_a}
3:      partition := partition[stmt[reg] ↦ ⊥] for stmt ∈ ⋃(block), for block ∈ F
4:      W_l := F.entryBlock.stmts
5:      ReachableBlocks := {F.entryBlock.label}
6:      while W_l ≠ ∅ do
7:          current := min_{≺_RPO}(W_l)
8:          W_l := W_l \ {current}
9:          if current.Block ∈ ReachableBlocks then
10:             if current ∈ term then
11:                 ReachableBlocks := ReachableBlocks ∪ eval_t(p, current[expr])
12:             else
13:                 canonical := eval(p, current[expr])
14:                 partition' := partition[current[reg] ↦ canonical]
15:                 if partition' ≠ partition then
16:                     W_l := W_l ∪ Dep(current)
17:                     partition := partition'
18:                 end if
19:             end if
20:         end if
21:     end while
22:     return partition
23: end procedure
```

```
entry(x, y):
    a.0 = i32 0
    b.0 = i32 0
    jmp loop
loop:
    a = φ i32 (a.0, a.i)
    b = φ i32 (b.0, b.i)
    c = xor i32 x, a
    a.i = sub i32 x, c
    b.i = add i32 b, 1
    cmp = ne i32 b.i, y
    br i1 cmp, loop, exit
exit:
    ret i32 a
```

```
entry(x, y):
    b.0 = i32 0
    jmp loop
loop:
    b = φ i32 (b.0, b.i)
    b.i = add i32 b, 1
    cmp = ne i32 b.i, y
    br i1 cmp, loop, exit
exit:
    ret i32 0
```

Figure 4.12: Example function where we have a loop-invariant value a. On the left the initial function and on the right the function after GVN.

4.5.1 Example application

In this section, we present a step-by-step application of the algorithm to a function containing a loop. The function is shown in Figure 4.12. This function's loop contains a loop-invariant value, `a`, which our optimistic algorithm is able to detect. In Table 4.1, we present the evolution of the value numbers as the execution of the optimistic algorithm progresses. Each column p_i represents the partition after pass i over the function.

reg	p_0	p_1	p_2	p_3
a.0	$\bot$	0	0	0
b.0	$\bot$	0	0	0
a	$\bot$	0	0	0
b	$\bot$	0	$\phi(0, b.i)$	$\phi(0, b.i)$
c	$\bot$	x	x	x
a.i	$\bot$	0	0	0
b.i	$\bot$	1	$1 + \phi(0, b.i)$	$1 + \phi(0, b.i)$
cmp	$\bot$	$1 \neq y$	$y \neq 1 + \phi(0, b.i)$	$y \neq 1 + \phi(0, b.i)$

Table 4.1: Value numbers as the execution of GVN on the function from Figure 4.12 progresses.

Partition p_0 is the initial partition where all registers are assumed to be unreachable. The first pass is an optimistic one, where we assume that `a` and `b` are equal to their values at loop entry—`0` from `a.0` and `b.0`, respectively. This assumption leads GVN to simplify `c` to `x`, which in turn leads to the discovery that `a.i` is equal to `0`.

The optimistic assumption also allows GVN to conclude that `b.i` is equal to `1`. Since the backedge-carried values `a.i` and `b.i` were processed, this means they are reachable, causing the phi expressions where they are used to be added to the worklist for reprocessing. In the second pass, GVN corroborates the facts uncovered by the optimistic assumption. It confirms that `a` is indeed loop-invariant and equal to `0`. On the other hand, `b`'s value number changed, implying the optimistically derived value number was

incorrect. As a consequence, its users also need to be corrected. In the third and final pass, only register `b` and its dependent registers are processed, because the registers related to `a` have already reached a fixed point. The value numbers remain the same as in the previous pass, the worklist becomes empty, and the algorithm terminates.

4.6 Loop Invariant Code Motion

Loop Invariant Code Motion [47] (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 our framework, LICM arises naturally from the interaction between PRE 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.

The process is ilustrated in Figure 4.13. 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 for both predecessors, thus confirming the speculation performed in the previous pass.

4.7 Elimination Phase

Up to this point, we have explored how to identify redundancies in a program by partitioning registers into congruence classes. In this section, our attention shifts to leveraging these congruence classes to eliminate redundancies efficiently.

Dominator-based elimination Dominator-based removal achieves redundancy elimination by using the dominance relation [29]. It iterates through the congruence classes of a given partition p and substitutes each member of the class c with a dominating leader. A dominating leader for a specific register is a member of the class that either resides in a dominating block or appears earlier in the same block.

Given a partition p, p^{-1} denotes the inverse of the partition. The inverse maps expressions to the set of program registers that have it as the canonical expression in p. Each entry of the inverse denotes

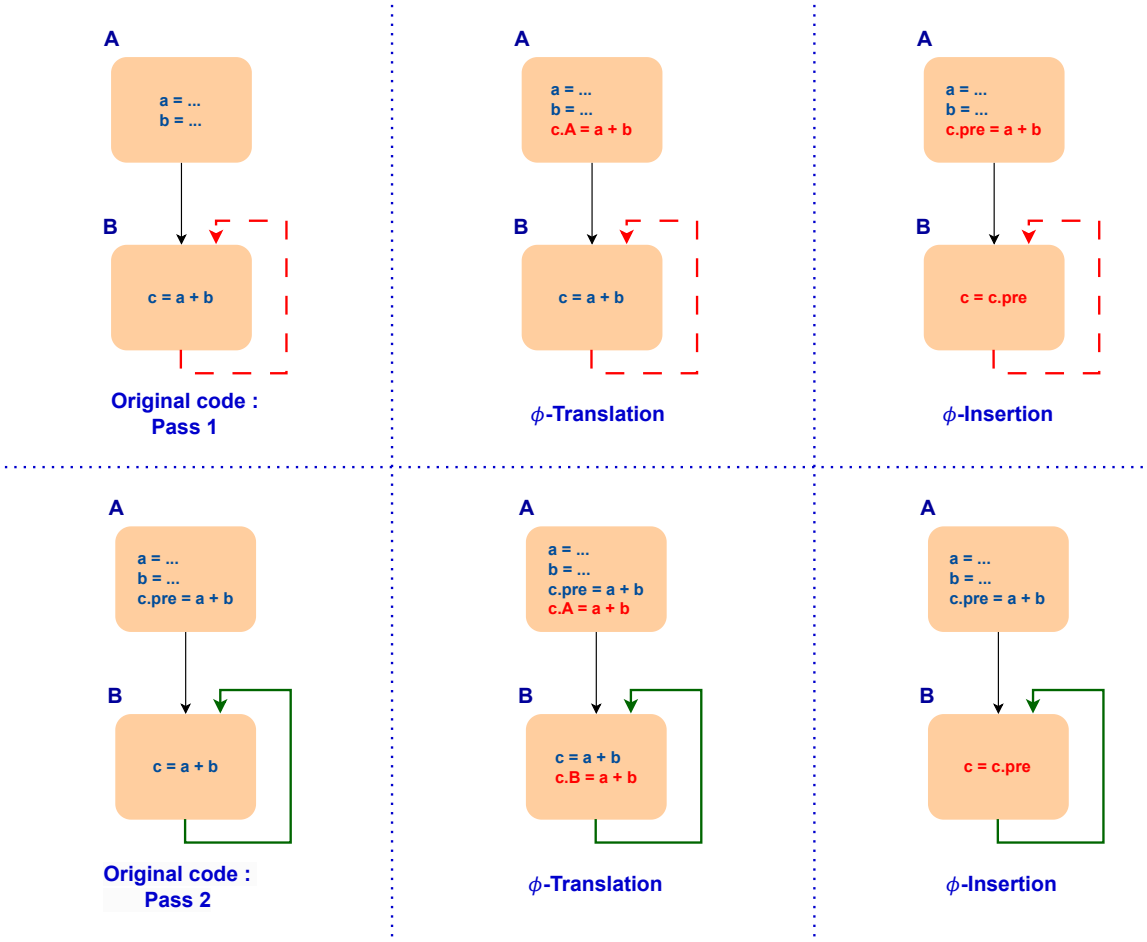


Figure 4.13: Performing LICM by using speculative PRE: In the first pass (top row), the backedge is assumed to be unreachable (•). In the second pass (bottom row), after discovering that the backedge is reachable (•), we have to corroborate the speculated PRE.

a congruence class. We use the inverse of the partition to iterate through the identified congruence classes. Using the invariant that members of the same congruence class are equivalent, our goal is to minimize both the number and the size of the congruence classes.

If the canonical expression of the congruence class is a constant or a function argument, there is a trivially dominating leader for every member of the class. This is because constants and function arguments have no dependencies and thus can be used anywhere in the program. Otherwise, we need to find a dominating leader for each register. The members of the class are visited using the order given by a DFS traversal of the dominator tree ($\prec_{dt}$). For registers in the same basic block, the order is given by increasing order within the block. The complete algorithm is presented in 4.2.

We now present an example of the application of these rules. Take the function in Figure 4.14 with the following arguments $F_A = \{c1, c2, x, y\}$ and the congruence class $c = \{r_1, r_2, r_3 : expr\}$ where

Algorithm 4.2 Elimination for GVN

```
1: procedure GVN-ELIMINATION(partition p)
2:     for all (expr, class) ∈ p^{-1} do
3:         if expr ∈ const ∪ F_a then
4:             for all reg ∈ class do
5:                 reg.replaceWith(expr)
6:             end for
7:         else
8:             Leader := ⊥
9:             for all reg ∈ sorted(class, ≺_dt) do
10:                if dominates(Leader, reg) then
11:                    reg.replaceWith(Leader)
12:                else
13:                    Leader := reg
14:                end if
15:            end for
16:        end if
17:    end for
18: end procedure
```

$r_1 \in B$, $r_2 \in C$, $r_3 \in D$ and $expr$ is the class-defining expression. We process the registers using the order given by the DFS traveral of the dominator tree (DFSDT), so r_1 is first then its r_3 and finally r_2. We start with register r_1, for which we do not have a dominating leader in the class, therefore we cannot replace it. However, r_1 might be a dominating leader for some other register so we set it as such. Next we process register r_3 as per DFSDT order and check if the current leader, in this case r_1, dominates r_3. Since it dominates, we can safely replace r_1 with r_3 in the function. Lastly we process r_2 which is not dominated by the current leader r_1, therefore we set r_2 as the leader. Since there are no more registers in the class the algorithm terminates. When the algorithm terminates, all the registers of the class that were not replaced, are the ones for which there is not a dominating leader.

Register	Leader	Dominates?	Action
r_1	ϵ	-	Set r_1 Leader
r_3	r_1	Yes	Replace r_3 with r_1
r_2	r_1	No	Set r_2 Leader
-	r_2	-	Done

Table 4.2: Steps to eliminate class c from the program in Figure 4.14. Initially there is no Leader we denote this using ϵ.

.

Using the DFS order of the dominator tree ensures that a register dominating another is always processed first. If we encounter a register not dominated by the current leader, it implies that subsequent registers are also not dominated by it.

In the example depicted in Figure 4.14, we eliminated `r3` from the program. However, it is crucial to recognize that this transformation may not always be beneficial. While our IR operates with a virtually

```
A:
  br i1 c1, label B, label C

B:
  r1 = add i32 x, y
  use(r1)
  br i1 c2, label D, label E

C:
  r2 = add i32 x, y
  use(r2)
  br label %F

D:
  r3 = add i32 x, y
  use(r3)
  e7 = br label F

E:
  e8 = br label F

F:
  ret 0
```

```
A:
  br i1 c1, label B, label C

B:
  r1 = add i32 x, y
  use(r1)
  br i1 c2, label D, label E

C:
  r2 = add i32 x, y
  use(r2)
  br label %F

D:
  use(r1)
  e7 = br label F

E:
  e8 = br label F

F:
  ret 0
```

Figure 4.14: Elimination IR example. On the left we have the program before elimination of class c. On the right after its elimination.

infinite set of registers, target machines typically have a finite set. Consequently, during the lowering from IR to assembly, the compiler performs register allocation [48]. Due to the limited number of target registers, not all IR registers can be directly assigned, leading to a process called spilling, where variables are moved to memory, and a reload is required when they are used. As memory accesses are slower than register accesses, the compiler aims to minimize spilling to enhance performance. High register pressure occurs when a program requires frequent spilling, indicating a large number of live variables in use.

There are situations where the cost of spilling a variable (storing its value and later reloading) exceeds the cost of recomputing its value. Some register allocators can identify these cases and employ redundant computation to avoid spilling, a technique known as rematerialization [49]. This contrasts with the goal of GVN, which aims to eliminate redundant computations, which can potentially increase register pressure.

4.8 Handling Undefined Behavior

The dominator-based elimination introduced in the previous section relies on the idea that, given two members of the same class, if one dominates the other, then it is always valid to replace the dominated one. In other words, it assumes that members of the same class are equivalent. However, due to

the presence of undefined behavior (UB), this is not always true. The canonical expression produced during value numbering is not necessarily equivalent to the original expression; it simply guarantees a refinement of the original expression. The following is a possible simplification where the expression for `r` is refined (from left to right):

```
t1 = mul nsw i8 a, b                    t1 = add i8 b, c
t2 = mul nsw i8 a, c                    r = mul i8 a, t1
r = add i8 t1, t2
```

Note that for this to be a valid refinement, the `nsw` flag needs to be removed. The `nsw` stands for "no signed wrap" meaning that in the case of signed overflow, the result is `poison`. Since `poison` can be refined to any value, this essentially tells the compiler that it can ignore the case of signed overflow, assuming it does not occur. For these specific expressions, it means the compiler can assume that $a \times b$ and $a \times c$ do not overflow. However, this does not imply that $a \times (b + c)$ also does not overflow. Therefore, the flag must be removed for this to be a valid refinement.

With that in mind, suppose we have the following program:

```
t1 = mul nsw i8 a, b
t2 = mul nsw i8 a, c
r1 = add i8 t1, t2
...
t3 = add i8 b, c
r2 = mul i8 a, t3
...
use(r2)
```

```
t1 = mul nsw i8 a, b
t2 = mul nsw i8 a, c
r1 = add i8 t1, t2
...
use(r1)
```

The expression for `r1` is $a \times_{nsw} b + a \times_{nsw} c$, which simplifies to the canonical expression $a \times (b + c)$, as we saw earlier. The expression for `r2` is $a \times (b + c)$, which is already in canonical form. Since registers `r1` and `r2` have the same canonical expression, they will end up in the same congruence class. During the elimination phase, because `r1` dominates `r2`, `r1` will replace `r2`, as shown on the right. This transformation, however, violates refinement.

Consider an execution where the assumption that $a \times b$ does not overflow is violated. Since we are working with $i8$ values, this could occur, for example, with $a = 50$ and $b = 5$, resulting in a poison value. As poison taints the entire expression tree, register `r1` will also become poison. In this scenario, replacing the uses of `r2` with the poison value would violate refinement. The issue here is that `r1`'s expression refines to one that can appear after `r1`.

To address this, we introduce an invariant in the simplifier: it must return either a subexpression of the original expression or a constant. In other words, no new instructions are created. The key observation is that this invariant ensures the refined expression always dominates the original register being simplified. This guarantees that a less-defined register can never replace a more-defined one, thereby preserving correctness.

5

Implementation

Contents

We implemented our optimistic algorithm in the LLVM compiler [50]. This compiler is the basis for many popular programming languages, such as Rust, Swift, and Julia, and is widely used in industry for its modular design and powerful optimization capabilities. It targets a broad range of hardware platforms including x86, ARM, and PowerPC architectures.

LLVM's machine-independent optimizer has 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 until a fixed point is reached. It is naively unified with partial redundancy elimination, where a PRE algorithm is run after the value numbering reaches a fixed point. To distinguish the GVN algorithm from the LLVM implementation, we denote this implementation as `GVNPRE`.

On the other hand, `NewGVN` is an optimistic algorithm based on the algorithms by Gargi [8]. It was introduced in LLVM in 2016 to address the limitations of `GVNPRE`, such as long compile times and the

[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

```
int sum(int a, int b) {
    if (a != b)
        return 0;
    return a + b;
}
```

```
define i32 @sum(i32 %0, i32 %1){
  %3 = icmp eq i32 %0, %1
  %4 = add nsw i32 %1, %0
  %5 = select i1 %3, i32 %4, i32 0
  ret i32 %5
}
```

```
sum:
    lea     ecx, [rsi + rdi]
    xor     eax, eax
    cmp     edi, esi
    cmove   eax, ecx
    ret
```

Figure 5.1: C program (left), LLVM IR after optimization (middle), x86 assembly (right).

absence of value numbering for memory operations, among other issues.[3]

Since it is an optimistic algorithm we decided to develop our solution on top of `NewGVN`.

5.1 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`[4] 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 value numbers are `GVNExpressions`[5] 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 5.2 twice reveals additional equivalences that a single run does not detect (refer to each output in Figure 5.3).[6] 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.

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

[3] https://lists.llvm.org/pipermail/llvm-dev/2016-November/107110.html
[4] https://github.com/llvm/llvm-project/blob/main/llvm/lib/Analysis/InstructionSimplify.cpp
[5] https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/Transforms/Scalar/GVNExpression.h
[6] https://github.com/llvm/llvm-project/issues/38031

```
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 5.2: Input program for which `NewGVN` does not reach a fixed point.

`%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 5.3: On the left, the output when running `NewGVN` once on the program from Figure 5.2. On the right, the output when running `NewGVN` a second time.

5.2 IR Checkpointing

To address the issue of handling optimistic assumptions in redundancy detection, we implemented a checkpointing mechanism that allows for dynamic updates to the IR based on optimistically discovered redundancies, while also providing the ability to roll back changes if necessary. To simplify this process, we modified the worklist fixpoint to a naive fixpoint, reprocessing the entire function during each pass.

This approach avoids the complexity of tracking dependencies when arbitrary changes can be made to the function. However, this is an area we plan to revisit in the future to explore more efficient solutions.

When an instruction moves from one congruence class to another, its definition is updated to use the canonical expression (`UpdateIR`). Before this update, the old uses of the instruction are saved (`SnapshotIR`). If the changes were made under optimistic assumptions, we must validate these assumptions by performing another pass. Prior to this validation pass, the updates made to the IR during the optimistic pass are rolled back (`RollbackIR`).

There are two scenarios where the optimistic assumption is applied: during the evaluation of phi instructions and when performing FRE (Full Redundancy Elimination). If any optimistic assumption in a given pass leads to the simplification of a phi expression or FRE, we classify that pass as an optimistic pass.

The termination of the algorithm is predicated on the fact that, although the changes made to the IR are reset between passes, the partition is reused. This means there can only be a finite number of optimistic passes, namely as many as D(SSA) + 1, where D(SSA) is the loop connectedness of the SSA graph [11], which is essentially a proxy for how nested the loops of the function are. The "+ 1" accounts for second-order effects of the optimistic assumption, which are only visible in the next iteration, such as backedge-carried values. The complete algorithm is presented in 5.1.

In the following section, we provide an example of the optimistic algorithm with checkpointing in action. For each pass, we show the initial state of the function and its state after all updates (Figures 5.4 and 5.5).

In the first pass (Figure 5.4), the algorithm initially assumes that the loop executes only once, causing it to ignore the values carried by the backedge `(%loop, %loop)` when evaluating the phi instructions. This simplifies `%a`, `%b`, `%c` to `0`, and the uses of these registers are updated accordingly, while snapshots are taken before each update. As a result, `%a.i`, `%b.i` simplify to `1`, and `%c.i` simplifies to `0`. Again, snapshots are taken, and updates are applied based on these discovered equivalences. In this case, the updated uses are primarily backedge uses at phi instructions. The transformed function after this first pass is presented on the right side of Figure 5.4.

If GVN is unable to prove the optimistic assumption (i.e., that the backedge is unreachable), the changes made to the IR based on this assumption must be rolled back. This ensures that any updates made during the optimistic pass are undone. However, even if the assumption proves incorrect, some of its second-order effects may still hold, allowing GVN to discover equivalences that might not have been found otherwise.

In the second pass (Figure 5.5), GVN determines that the backedge `(%loop, %loop)` is reachable, so the values it carries must be considered. However, the equivalences discovered in the optimistic pass can still be used. For example, `%a` and `%b` are found to be equivalent because their backedge values, `%a.i`

Algorithm 5.1 Optimistic GVN with Checkpointing

```
1: procedure GVN-CHECKPOINTING(function F)
2:     partition := {arg ↦ arg | arg ∈ F_a}
3:     partition := partition[stmt[reg] ↦ ⊤] for stmt ∈ ⋃(block), for block ∈ F
4:     partition' := None
5:     ReachableBlocks := {F.entryBlock.label}
6:     while partition' ≠ partition do
7:         for all (inst, uses) ∈ Snapshot do                          ▷ RollbackIR
8:             for all use ∈ uses do
9:                 use.value = inst
10:            end for
11:        end for
12:        for all block ∈ sorted(ReachableBlocks, ≺_RPO) do
13:            for all current ∈ block do
14:                if current ∈ term then
15:                    ReachableBlocks := ReachableBlocks ∪ eval_t(p, current[expr])
16:                else
17:                    canonical := eval(p, current[expr])
18:                    partition' := partition[current[reg] ↦ canonical]
19:                    if partition' ≠ partition then
20:                        Snapshot[current] := current.uses           ▷ SnapshotIR
21:                        current[expr] := canonical                    ▷ UpdateIR
22:                        partition := partition'
23:                    end if
24:                end if
25:            end for
26:        end for
27:    end while
28:    return p
29: end procedure
```

```
define i32 @foo() {
entry:
    br label %loop
loop:
    %a = phi i32 [0, %entry], [%a.i, %loop]
    %b = phi i32 [0, %entry], [%b.i, %loop]
    %c = phi i32 [0, %entry], [%c.i, %loop]
    %a.i = add i32 %a, 1
    %b.i = add i32 %b, 1
    %c.i = add i32 %c, %a
    br i1 ...

exit:
    ...
}
```

```
define i32 @foo() {
entry:
    br label %loop
loop:
    %a = phi i32 [0, %entry], [1, %loop]
    %b = phi i32 [0, %entry], [1, %loop]
    %c = phi i32 [0, %entry], [0, %loop]
    %a.i = add i32 0, 1
    %b.i = add i32 0, 1
    %c.i = add i32 0, 0
    br i1 ...

exit:
    ...
}
```

Figure 5.4: First GVN Pass

and `%b.i`, are equivalent. This discovery is still based on the optimistic assumption, requiring a further pass for confirmation. Since `%a` and `%b` are equivalent, the uses of `%b` are replaced with `%a`, making `%a.i` and `%b.i` lexically equal and trivially equivalent. The uses of `%b.i` are replaced with `%a.i`.

Regarding `%c` and `%c.i`, in the optimistic pass, they were determined to be constants equal to `0`. However, in this second pass, since `%a` is no longer `0`, `%c.i`'s expression cannot be simplified, which changes `%c`'s value number.

```
define i32 @foo() {
entry:
    br label %loop
loop:
    %a = phi i32 [0, %entry], [%a.i, %loop]
    %b = phi i32 [0, %entry], [%b.i, %loop]
    %c = phi i32 [0, %entry], [%c.i, %loop]
    %a.i = add i32 %a, 1
    %b.i = add i32 %b, 1
    %c.i = add i32 %c, %a
    br i1 ...

exit:
    ...
}
```

```
define i32 @foo() {
entry:
    br label %loop
loop:
    %a = phi i32 [0, %entry], [%a.i, %loop]
    %b = phi i32 [0, %entry], [%a.i, %loop]
    %c = phi i32 [0, %entry], [%c.i, %loop]
    %a.i = add i32 %a, 1
    %b.i = add i32 %a, 1
    %c.i = add i32 %c, %a
    br i1 ...

exit:
    ...
}
```

Figure 5.5: Second GVN pass.

In the third pass, GVN confirms the second-order effects of the optimistic assumption. Registers `%a` and `%b` are confirmed to be equivalent, as they have equivalent values both from the loop preheader and the loop backedge—`0` and `%a.i`, respectively. Such equivalence can only be detected using an optimistic algorithm; a pessimistic approach would conservatively assume that the backedge values for `%a` and `%b` are different.

However, some second-order effects from the optimistic pass were incorrect, particularly the assumption that `%c` was loop-invariant and equal to `0`. Since a rollback was performed, there is no risk of discovering redundancies based on an incorrect assumption. This third pass produces the same partition as the second pass, indicating that a fixed point has been reached and that the algorithm can terminate.

In conclusion, the checkpointing mechanism enables GVN to make optimistic assumptions and explore potential simplifications without committing to them permanently. By rolling back changes when assumptions prove incorrect, the algorithm can still leverage useful equivalences discovered in the process. This method, while conservative in its final validation, allows for more aggressive optimizations that would be missed by purely pessimistic approaches, ultimately leading to improved redundancy elimination.

6

Evaluation

Contents

In this chapter we compare the performance of different types of GVN algorithms, including our optimistic algorithm unified with PRE. GVN, at its core, is a simple algorithm designed to discover and eliminate redundant instructions. Much of the complexity arises when various extensions are introduced to increase its precision. One of the goals of this thesis was to understand how these variations and extensions impact performance in real-world applications.

6.1 Setup

All experiments were conducted using an `Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz` machine running the `CentOS Stream 9` operating system. Benchmarking was performed using the open-source,

automated benchmarking tool Phoronix Test Suite (PTS)[1]. PTS made the benchmarking process easier to set up and reproduce. Additionally, it provides a wide range of test profiles.[2] A test profile includes a source application, an installation script, and a set of tests. We selected a total of 18 profiles comprising a diverse set of C/C++ applications. The complete set is presented in Table 6.1.

Profile	Description	kLOC
aircrack-ng-1.3.0	WiFi network security	67
botan-1.6.0	cryptography	148
compress-pbzip2-1.6.0	compression	11
compress-zstd-1.6.0	compression	87
crafty-1.4.5	chess engine	25
draco-1.6.0	graphics compression	52
encode-flac-1.8.1	audio compression	60
fftw-1.2.0	FFT	256
graphics-magick-2.1.0	image manipulation	263
jpegxl-1.5.0	image compression	284
mafft-1.6.2	bioinformatics	94
quantlib-1.2.0	quantitative finance	397
redis-1.4.0	in-memory database	171
rnnoise-1.0.2	noise supression	14
scimark2-1.3.2	science and engineering	1
simdjson-2.0.1	JSON parser	244
sqlite-speedtest-1.0.1	SQL database engine	277
tjbench-1.2.0	JPEG (de)compression	58

Table 6.1: Selected profiles from PTS with a short description and the numbers of lines of code (rounded to the nearest thousand).

6.2 GVN Variants

The main focus of our work was on the impact of algebraic simplification and partial redundancy elimination. The complete set of variants is presented in Table 6.2. Algebraic simplification in `NewGVN` is partial compared to the others, due to the issues discussed in Chapter 5.

The source applications were compiled using the default `-O2` and `-O3` optimization pipelines. The only difference was that in each instance where GVN was called, one of the variants from Table 6.2 was used.

The set of 18 selected profiles produced a total of 109 test cases. Each test case was run for each variant and optimization pipeline, giving us a total of 1,526 entries.

[1] https://www.phoronix-test-suite.com/
[2] https://github.com/phoronix-test-suite/test-profiles

Variant	Description
`NoGVN`	skips GVN
`Opt -Simpl`	optimistic without algebraic simplification
`Opt`	optimistic with algebraic simplification
`Opt +FRE`	`Opt` + full redundancy elimination
`Opt +PRE`	`Opt +FRE` + partial redundancy elimination
`GVN`	LLVM: pessimistic with partial redundancy elimination
`NewGVN`	LLVM: optimistic with algebraic simplification (partial)

Table 6.2: GVN variants evaluated.

6.3 Performance Results

Figure 6.1 presents a summary of the results for each variant, including a line representing the mean speedup.

In `-O2`, our full implementation, which includes PRE, outperforms all other variants, including the two LLVM implementations: `GVNPRE` and `NewGVN`. Regarding the optimistic algorithm, its performance improves as more aggressive techniques are applied. An unexpected result is that, on average, the basic optimistic algorithm—without simplification or other techniques—performs worse than not having a value numbering pass at all. We expected this variant to perform slightly better than having no value numbering, or at least perform similarly. Whatever the root cause of this regression, it is likely also affecting the performance of the other optimistic variations, as the more advanced variants subsume the basic one. Thus, the problematic transformations are also being applied in the more advanced variants. Once these issues are resolved, we should see an even greater performance gain relative to the LLVM implementations.

In `-O3`, the best-performing variants are `NewGVN` and `Opt`, but even they produce, on average, the same performance as not having a value numbering pass, while the other variants degrade performance. The likely explanation for these results is twofold. First, the `-O3` pipeline makes some of the transformations performed by value numbering redundant. As the pipeline becomes longer and more complex, the likelihood that a pass or interaction between passes subsumes value numbering increases. However, this alone would result in value numbering having no impact, which is what we observe with the `NewGVN` and `Opt` variants. The second part of the explanation addresses why the other four variants degrade performance. This is because modern compiler optimization pipelines are extremely complex, and for many optimization passes, determining in advance whether a specific transformation will be profitable is very costly or even impossible. Therefore, they rely heavily on heuristics. While this approach is generally effective, it can be problematic in certain cases. For example, performing a statically optimal PRE transformation—i.e., one that minimizes the number of computations for a given value—might unexpectedly disrupt a heuristic used by a downstream optimization pass. If that optimization was crucial

for performance, then performing an otherwise seemingly profitable transformation could result in a net performance loss.

In Table 6.3 we present for each variant and optimization level the percentage of test cases where adding value numbering degraded the performance. The fact that in `-O3` the two variants with the most degradation cases are `GVN` and `Opt +PRE` points to the idea that the pipeline interaction issues are exacerbated by the PRE stage in GVN.

	GVN	NewGVN	Opt -Simpl	Opt	Opt +FRE	Opt +PRE
-O2	43.9	32.7	45.7	40.1	36.4	30.8
-O3	56.0	43.9	50.4	41.1	45.8	55.1

Table 6.3: Percentage of tests cases with degradation (speedup less than 1).

In Figures 6.2 to 6.5, we present the results for each test case under both `-O2` and `-O3`. In the following sections, we highlight some of the more noteworthy cases.

6.3.1 Results Analysis for `-O2`

We begin by analyzing the test cases for `-O2`. In the `redis` benchmark, our implementation outperforms the LLVM implementations in most test cases. The notable exception is `redis-12`, where `GVN` produced an improvement of 86%. Another interesting case is `redis-15`, where `GVN` and `Opt+PRE`, theoretically the most complete implementations, resulted in the worst performance, while one of the simpler variants, notably the optimistic variant without algebraic simplification, FRE, or PRE, delivered the best result.

The `simdjson` benchmark was one of the few where value numbering had a positive effect across most variants, with the worst case being neutral. The best-performing variant was `GVN`, which provided moderate improvements (5-10%) across all test cases.

In contrast to `simdjson`, we have `zstd`, where value numbering had little impact, regardless of the variant used. The exception was a few test cases where `GVN` produced a slight improvement.

In the `jpegxl` benchmark, we observe that the optimistic variants with FRE and PRE degrade performance in a few test cases by about 10%. The likely explanation is that the code introduced by these transformations disrupts optimization heuristics further down the pipeline, and the gains from these transformations are outweighed by the losses from missing other optimizations. The remaining variants do not appear to significantly impact performance.

The `fftw` benchmark showed little benefit from value numbering. The overall impact was mostly neutral, except for a few cases where the optimistic variant without algebraic simplification degraded performance by up to 20%. If this regression is common to the more advanced optimistic variants, fixing this case may reveal performance gains in the optimistic variants with algebraic simplification, full, and partial redundancy elimination.

Finally, in the `rnnoise` benchmark, the optimistic variant without the more advanced features outperformed all other variants, improving performance by 20%, while every other variant had no noticeable impact.

6.3.2 Results Analysis for `-O3`

We now analyze the test cases for `-O3`. In the `redis` benchmark, we observe that `GVN` significantly degrades performance. A notable case is `redis-12`, where all variants degrade performance by about 50% compared to having no value numbering, which is the opposite of what we see in `simdjson`, where every variant has a positive impact. In `simdjson`, `GVN` appears to be the most performant. For `zstd`, aside from a few cases where `GVN` provides a slight improvement, the performance impact of any variant is generally negative or neutral. The remaining test cases show behavior similar to their `-O2` counterparts.

6.4 Code Size Results

In addition to running the test cases, we gathered the size of each benchmark after compilation. These results are presented in Figures 6.6 and 6.7, relative to a baseline of having no value numbering pass.

For the `fftw` benchmark, we observe that `NewGVN` produces a larger code size compared to compiling without a value numbering pass. However, the higher number of instructions deleted by the other variants did not translate into better performance. In contrast, for the `simdjson` benchmark, there is a stronger correlation between code size reduction and test performance. Notably, `GVN`, `Opt +FRE`, and `Opt +PRE` showed the largest code size reductions and also performed the best. A similar pattern is observed with `-O3` compilation, except with even greater code size reductions, due to the more aggressive optimizations performed by the `-O3` pipeline.

The lack of correlation between code size reduction and performance is due to the fact that not all eliminated redundancies have the same impact on performance. For instance, eliminating a single redundancy from a hot code path may result in a greater speedup than removing multiple redundancies from a cold code path. Another factor relates to the type of instructions eliminated. Typically, memory access instructions, such as loads, are more expensive than arithmetic operations, like additions or subtractions. Therefore, eliminating a single load might have a bigger performance impact than removing multiple arithmetic instructions. In such cases, there is no direct correlation between code size reduction and performance.

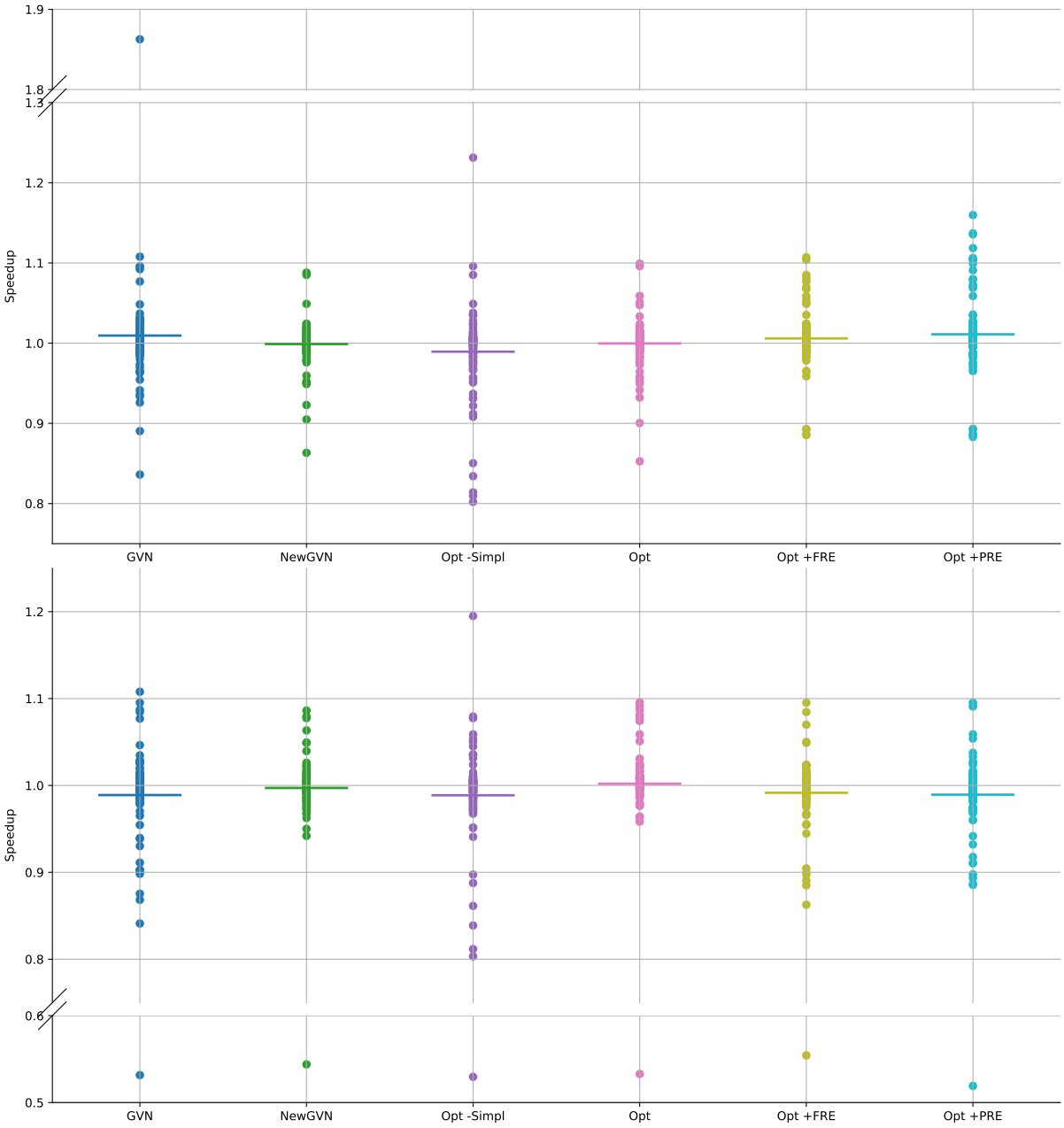


Figure 6.1: Summary of speedups for each variant in `-O2` (top) and `-O3` (bottom) relative to having no GVN pass. Each point is a test case and the horizontal lines present the average speedup for each variant.

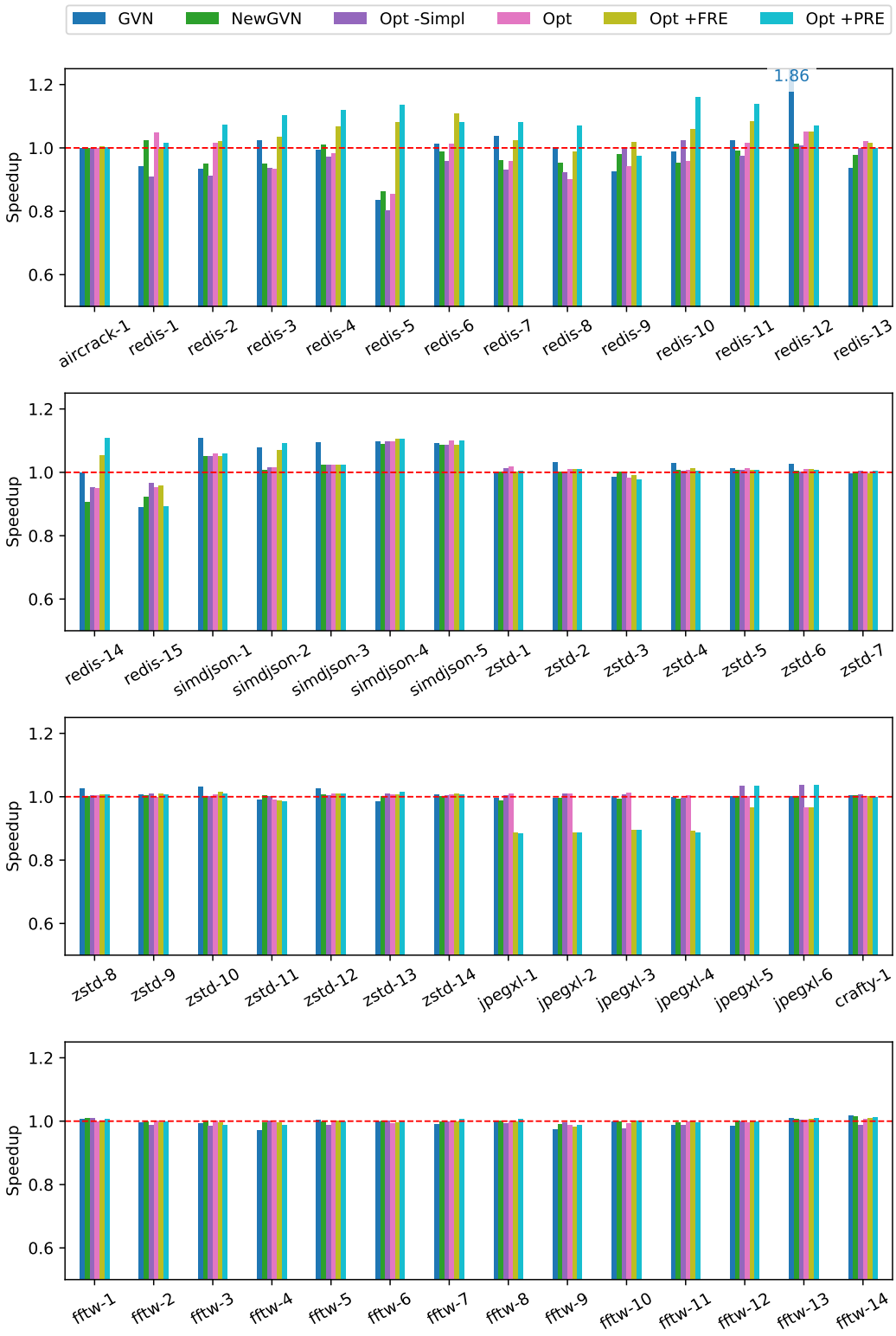


Figure 6.2: Performance results O2 (part 1/2)

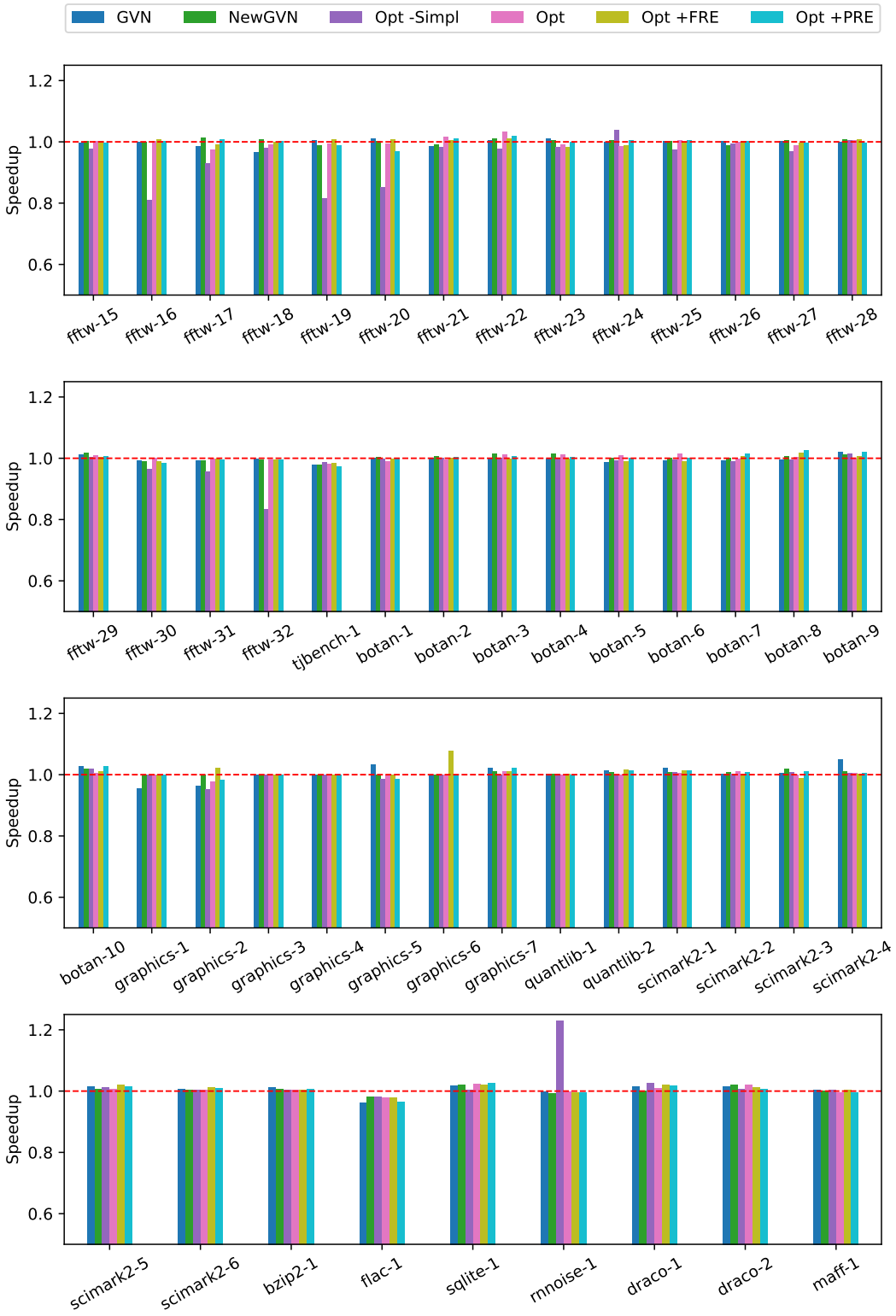


Figure 6.3: Performance results O2 (part 2/2)

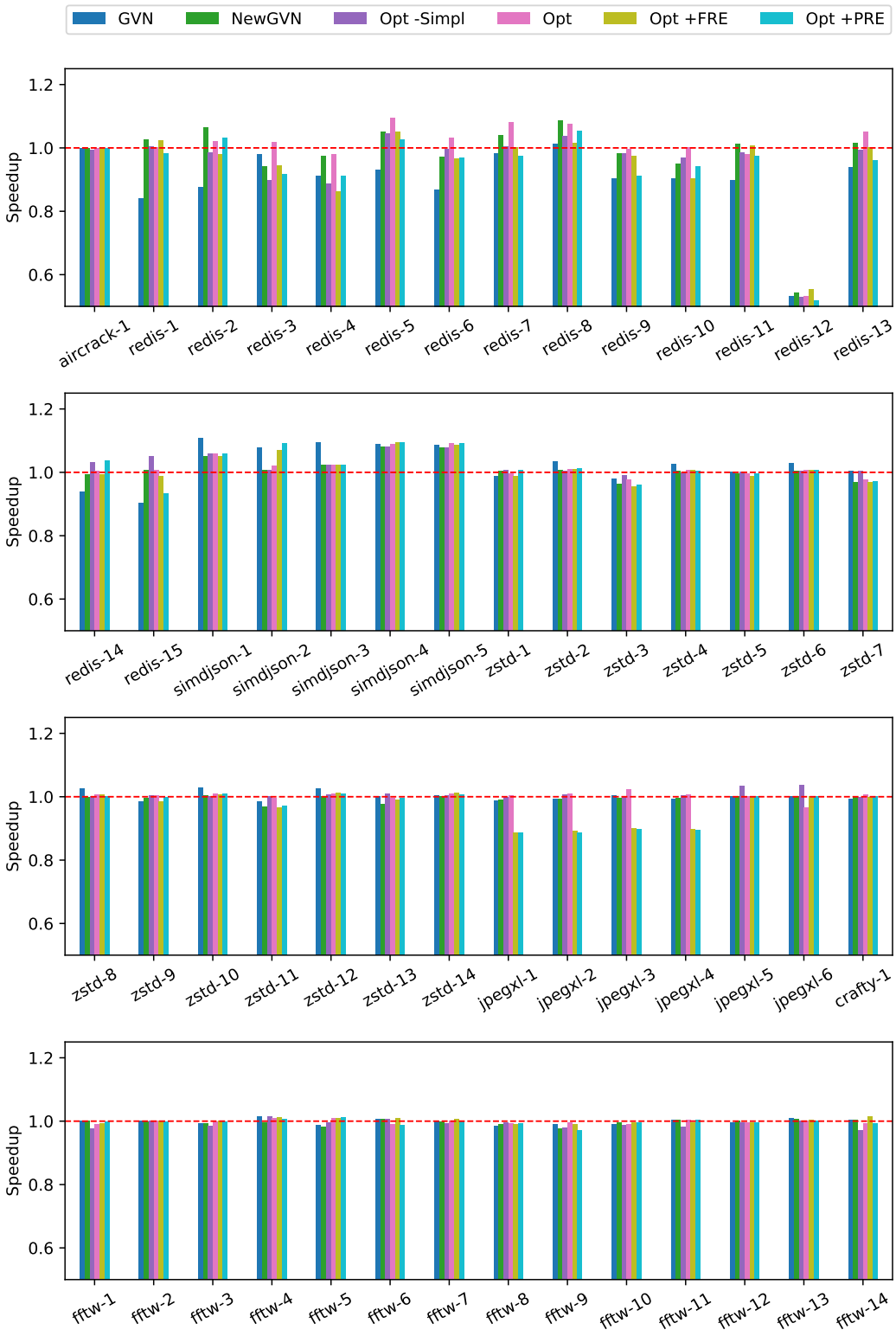


Figure 6.4: Performance results O3 (part 1/2)

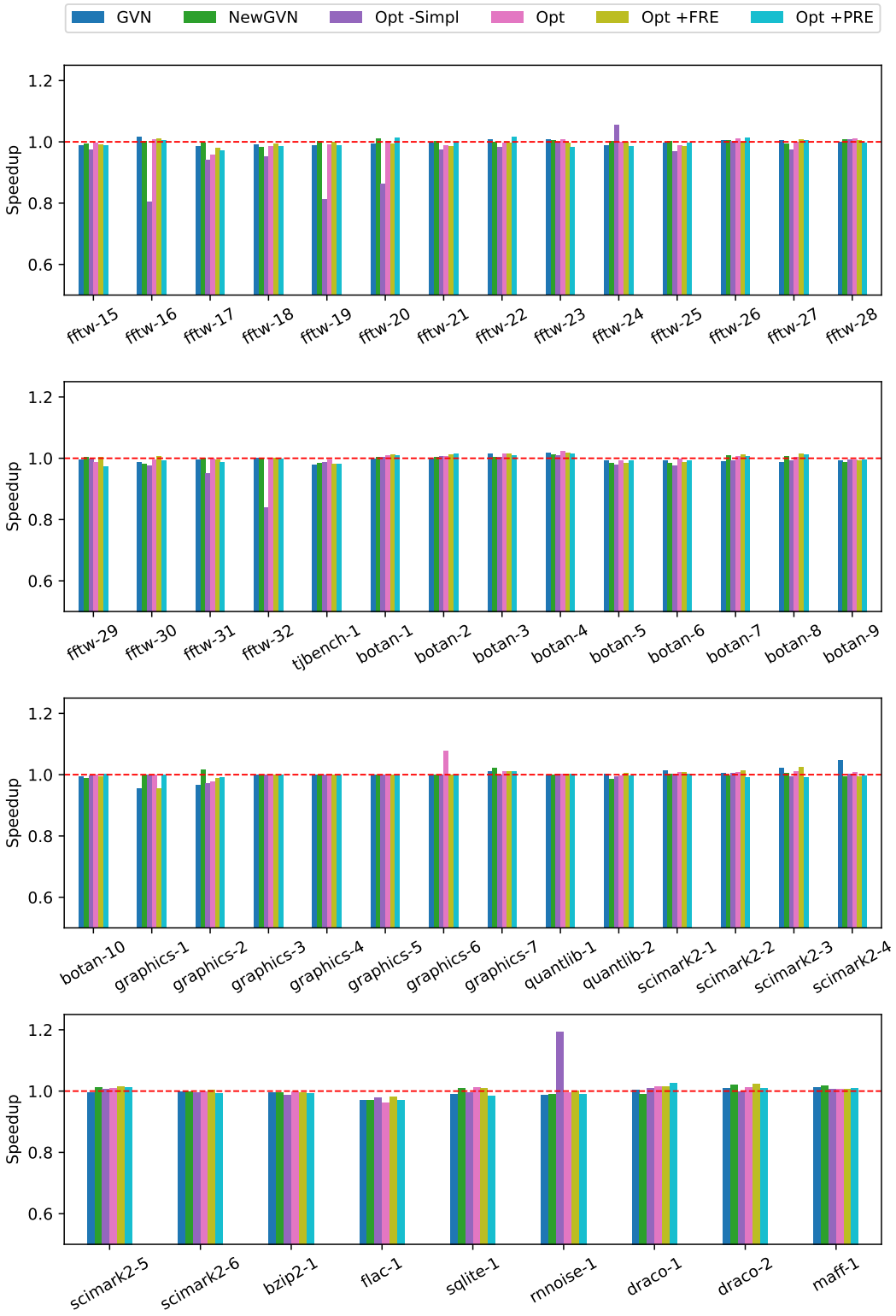


Figure 6.5: Performance results O3 (part 2/2)

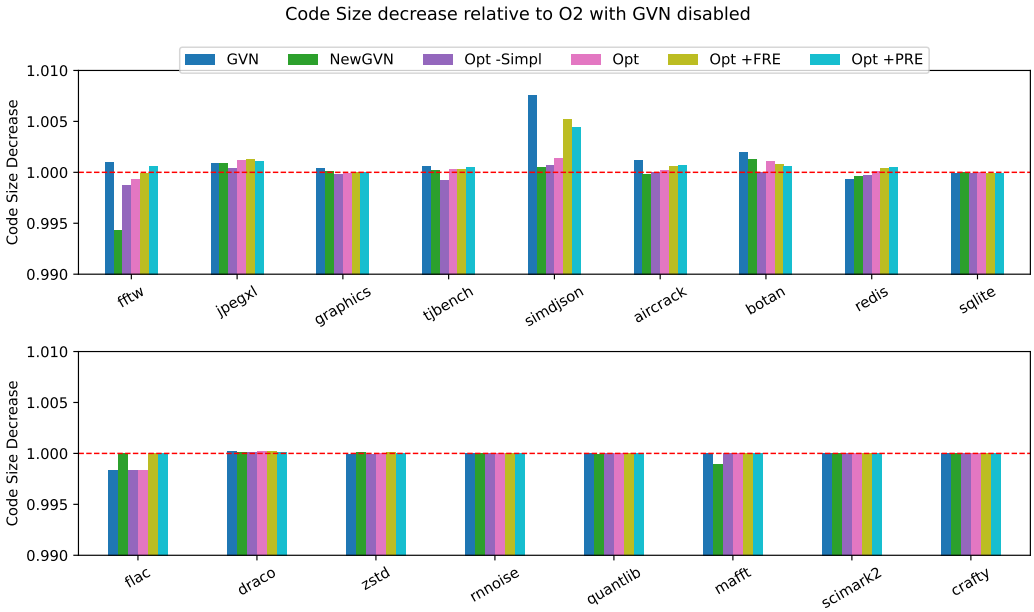


Figure 6.6: Binary size decrease for `-O2` compilation

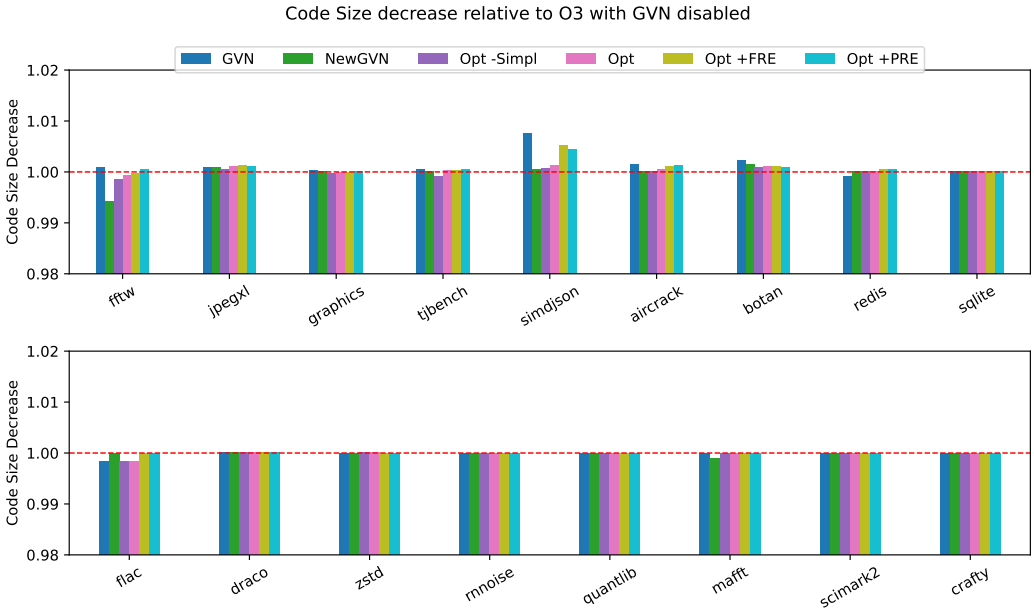


Figure 6.7: Binary size decrease for `-O3` compilation

7

Conclusions and Future Work

Contents

In this work, we studied existing GVN algorithms. The original hash-based algorithm [4] is straightforward, but many developments have been made over the years to enhance its precision and efficiency. These improvements include the optimistic assumption, the unification with other optimizations such as algebraic simplification and partial redundancy elimination, and advances in program representations like SSA. However, these enhancements have been scattered across different works.

We developed and implemented the first optimistic GVN algorithm that incorporates partial redundancy elimination, algebraic simplification, and unreachable code elimination, while operating on an SSA-based IR extended with SSI and MSSA. Our solution not only detects redundant computations but also identifies redundant loads and stores, which are crucial for performance. Furthermore, unifying partial redundancy elimination under an optimistic GVN framework allows us to perform loop-invariant code motion.

This solution was implemented in the open-source LLVM compiler, where it proved to be competitive with existing value numbering solutions, often surpassing them in performance. Our implementation introduced a novel fixed-point algorithm, enabling GVN to optimistically modify functions and revert

changes when necessary. Additionally, we identified problematic cases where value numbering caused regressions, regardless of the variant used.

7.1 Future Work

While GVN reduces redundant computations, it can also increase register pressure by adding more variables to live ranges. Future work should examine how the algorithm affects register allocation and resource usage during compilation. High register pressure may lead to performance degradation as the compiler generates more instructions to spill variables to memory. Therefore, a detailed analysis of the trade-offs between reduced computation and increased register pressure is needed, as well as strategies to minimize any negative impacts. In particular, the effects of increased register pressure on various hardware platforms should be explored.

Additionally, our current implementation and evaluation were conducted on the X86 architecture. To gain a more comprehensive understanding of the proposed GVN algorithm's performance, it is essential to run benchmarks on different architectures, such as ARM, RISC-V, GPUs, and specialized accelerators like TPUs. Testing across diverse hardware platforms will help assess the algorithm's scalability and its ability to generate optimized code tailored to different architectural characteristics, including memory hierarchies, instruction sets, and parallel processing capabilities.

Although this thesis unified GVN with optimizations like partial redundancy elimination (PRE), algebraic simplification, and dead code elimination, future research could explore unifying GVN with additional optimizations. For example, combining GVN with loop optimizations [51,52], auto-vectorization [53], and interval analysis [54] could further enhance performance. However, managing the interactions between different optimizations presents a key challenge. Careful analysis is needed to ensure these optimizations work together effectively within the same framework.

The thesis leverages Static Single Assignment (SSA) form and its extensions, such as Static Single Information (SSI) and Memory SSA (MSSA), to improve GVN. However, future work could explore other SSA variants [55,56] for potential benefits. For instance, Predicated SSA (PSSA) [57], which associates variable definitions with predicates, could optimize control flow more effectively. Additionally, Parallel SSA (PSPA), designed for concurrent programs, could be investigated in the context of GVN to optimize multi-threaded code. Exploring these variants could lead to improved GVN performance in specific scenarios.

The optimistic assumption in GVN allows for detecting equivalences by initially assuming that loop iterations execute only once, which is later validated through analysis. Future work could explore more fine-grained control over where and when this optimistic assumption is applied. Instead of applying it globally, selective application based on profiling information, loop depth, or heuristics could yield more

precise optimizations and reduce unnecessary overhead from overly optimistic assumptions.

Profile-guided optimization (PGO) [58], which relies on runtime profiling data, could further enhance GVN precision. By leveraging real-world execution data, profiling can identify frequently executed paths where GVN can be applied aggressively, while avoiding unnecessary value numbering in cold paths. PGO could also help fine-tune the optimistic assumption and decide where to apply aggressive optimizations based on actual runtime behavior.

Another potential research direction is the detection of quasi-invariants [59, 60]. These values are not strictly invariant but remain constant for many loop iterations or under certain conditions. Extending GVN to detect and optimize quasi-invariants could yield significant performance gains, particularly in long-running loops or programs with stable but non-constant values. Developing algorithms to identify and exploit such quasi-invariants would represent a natural extension of traditional invariant code motion and GVN.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[3] P. B. Schneck, "A survey of compiler optimization techniques," in *ACM*, 1973.

[4] J. Cocke and J. Schwartz, *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 1970.

[5] G. A. Kildall, "A unified approach to global program optimization," in *POPL*, 1973.

[6] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *POPL*, 1988.

[7] C. Click, "Global code motion/global value numbering," in *PLDI*, 1995, pp. 246–257.

[8] K. Gargi, "A sparse algorithm for predicated global value numbering," in *PLDI*, 2002.

[9] K. Ansar and N. Saleena, "Redundancy elimination using global value numbering," in *ICECCT*, 2017, pp. 1–5.

[10] J. H. Reif and H. R. Lewis, "Symbolic evaluation and the global value graph," in *POPL*, 1977.

[11] K. D. Cooper and T. Simpson, "SCC-based value numbering," 1995.

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

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, oct 1991.

[14] F. Rastello and F. B. Tichadou, *SSA-based Compiler Design*. Springer, 2022.

[15] S. Ananian, “The static single information form,” 2001.

[16] D. Novillo, “Memory SSA-a unified approach for sparsely representing memory operations,” in *Proc of the GCC Developers' Summit*, 2007.

[17] J. Stanier and D. Watson, “Intermediate representations in imperative compilers: A survey,” *ACM Comput. Surv.*, vol. 45, no. 3, Jul. 2013.

[18] F. E. Allen, “Control flow analysis,” in *Symposium on Compiler Optimization*, 1970.

[19] R. T. Prosser, “Applications of boolean matrices to the analysis of flow diagrams,” in *IRE-AIEE-ACM*, 1959.

[20] G. Bilardi and K. Pingali, “Algorithms for computing the static single assignment form,” *J. ACM*, vol. 50, no. 3, p. 375–425, may 2003.

[21] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson, “Practical improvements to the construction and destruction of static single assignment form,” *Softw. Pract. Exper.*, vol. 28, no. 8, p. 859–881, jul 1998.

[22] V. Raman, “Pointer analysis – a survey,” 2004.

[23] Y. Smaragdakis, G. Balatsouras *et al.*, “Pointer analysis,” *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[24] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, “Undefined behavior: what happened to my code?” in *APSYS*, 2012.

[25] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, “Towards optimization-safe systems: analyzing the impact of undefined behavior,” in *SOSP*, 2013.

[26] ——, “A differential approach to undefined behavior detection,” *ACM Transactions on Computer Systems (TOCS)*, 2015.

[27] R. V. Baev, L. V. Skvortsov, E. A. Kudryashov, R. Buchatskiy, and R. Zhuykov, “Preventing vulnerabilities caused by optimization of code with undefined behavior,” *Programming and Computer Software*, 2022.

[28] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, “Taming undefined behavior in llvm,” in *PLDI*, 2017.

[29] P. BRIGGS, K. D. COOPER, and L. T. SIMPSON, “Value numbering,” *Software: Practice and Experience*, vol. 27, no. 6, pp. 701–724, 1997.

[30] F. E. Allen and J. Cocke, “A program data flow analysis procedure,” *Commun. ACM*, Mar. 1976.

[31] U. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., 2009.

[32] A. Møller and M. I. Schwartzbach, “Static program analysis,” *Notes. Feb*, 2012.

[33] J. B. Kam and J. D. Ullman, “Monotone data flow analysis frameworks,” *Acta informatica*, vol. 7, no. 3, pp. 305–317, 1977.

[34] O. Rüthing, J. Knoop, and B. Steffen, “Detecting equalities of variables: Combining efficiency with precision,” in *SAS*, 1999.

[35] S. Gulwani and G. C. Necula, “Global value numbering using random interpretation,” in *POPL*, 2004.

[36] ——, “A polynomial-time algorithm for global value numbering,” *Sci. Comput. Program.*, p. 97–114, 2007.

[37] N. Saleena and V. Paleri, “A simple algorithm for global value numbering,” *arXiv preprint arXiv:1303.1880*, 2013.

[38] R. R. Pai, “Detection of redundant expressions,” *Comput. Lang. Syst. Struct.*, vol. 46, no. C, p. 167–181, nov 2016.

[39] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM J. Comput.*, 1972.

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

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

[42] P. Briggs and K. D. Cooper, “Effective partial redundancy elimination,” in *PLDI*, 1994.

[43] T. VanDrunen and A. L. Hosking, “Value-based partial redundancy elimination,” in *CC*, 2004.

[44] R. Odaira and K. Hiraki, “Partial value number redundancy elimination,” in *LCPC*, 2004.

[45] K. D. Cooper, M. Hall, K. Kennedy, and L. Torczon, “Interprocedural analysis and optimization,” *Communications on Pure and Applied Mathematics*, vol. 48, no. 9, pp. 947–1003, 1995.

[46] D. M. Dhamdhere, “E-path_pre: partial redundancy elimination made easy,” *SIGPLAN Not.*, 2002.

[47] B. Steffen, J. Knoop, and O. Rüthing, “The value flow graph: A program representation for optimal program transformations,” in *ESOP*, 1990.

[48] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47–57, 1981. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0096055181900485

[49] P. Briggs, K. D. Cooper, and L. Torczon, "Rematerialization," *SIGPLAN Not.*, vol. 27, no. 7, p. 311–321, 1992.

[50] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[51] J.-C. Huang and T. Leng, "Generalized loop-unrolling: a method for program speedup," in *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122)*. IEEE, 1999, pp. 244–248.

[52] P. Lokuciejewski, F. Gedikli, and P. Marwedel, "Accelerating wcet-driven optimizations by the invariant path paradigm: a case study of loop unswitching," in *SCOPES*, 2009.

[53] D. Nuzman and R. Henderson, "Multi-platform auto-vectorization," in *CGO*, 2006.

[54] H. Schichl and A. Neumaier, "Interval analysis on directed acyclic graphs for global optimization," *Journal of Global Optimization*, vol. 33, pp. 541–562, 2005.

[55] M. Mantione and F. Chow, "Hashed SSA form: HSSA," in *SSA-based Compiler Design*, 2021, pp. 213–225.

[56] K. Knobe and V. Sarkar, "Array SSA form and its use in parallelization," in *PLDI*, 1998.

[57] L. Carter, B. Simon, B. Calder, and J. Ferrante, "Predicated static single assignment," in *PACT*, 1999.

[58] B. Wicht, R. A. Vitillo, D. Chen, and D. Levinthal, "Hardware counted profile-guided optimization," *arXiv preprint arXiv:1411.6361*, 2014.

[59] J.-Y. Moyen, T. Rubiano, and T. Seiller, "Loop quasi-invariant chunk detection," in *International Symposium on Automated Technology for Verification and Analysis*, 2017.

[60] L. Song, Y. Futamura, R. Glück, and Z. Hu, "A loop optimization technique based on quasi-invariance," in *Proceedings of IFIP Conference on Software: Theory and Practice*, 2000.