LLVM Optimization Passes: A Study of Analysis and Transformation

Amaan Ali Sayed

1 Introduction

LLVM provides a powerful set of analysis and transformation passes designed to improve the intermediate representation (IR) used in compilers. This report explores three of these optimization passes in depth:

- **Mem2Reg** (Promote Memory to Register)
- **GVN** (Global Value Numbering)
- **Loop Unroll**

For each pass, we provide a description, illustrate its effect using a small Javalette program, and compare the original and optimized LLVM IR code. All code examples are compiled from a simple Javalette compiler and transformed using `opt` with the respective pass.

2 Mem2Reg (Promote Memory to Register)

Description

The Mem2Reg pass transforms memory-based variables (defined via `alloca`) into SSA-form register variables. It uses dominance frontiers to place `phi` functions where control flow converges, eliminating redundant memory accesses.

Javalette Program

```
int main() {
  int x;
  x = 42;
  return x;
}
```

LLVM IR Before Optimization (Generated by compiler)

```
define i32 @main() {
entry:
  %t0 = alloca i32
  store i32 0, i32* %t0
  store i32 42, i32* %t0
```

```
  %t1 = load i32, i32* %t0
  ret i32 %t1
}
```

LLVM IR After `opt -mem2reg`

```
define i32 @main() {
entry:
  ret i32 42
}
```

Explanation

The pass eliminates the stack allocation and replaces the load-store pair with a direct use of the constant value. This simplifies the IR and enables subsequent optimizations like constant folding.

3 GVN (Global Value Numbering)

Description

GVN identifies redundant expressions across the program and reuses their values. It assigns a unique number to each expression and eliminates those with equivalent computations.

Javalette Program

```
int main() {
   int a = 10;
   int b = 10;
   int c = a + b;
   int d = a + b;
   return c + d;
}
```

LLVM IR Before Optimization

```
define i32 @main() {
entry:
  %t0 = alloca i32
  store i32 10, i32* %t0
  %t1 = alloca i32
  store i32 10, i32* %t1
  %t3 = load i32, i32* %t0
  %t4 = load i32, i32* %t1
  %t5 = add i32 %t3, %t4
  %t2 = alloca i32
  store i32 %t5, i32* %t2
  %t7 = load i32, i32* %t0
  %t8 = load i32, i32* %t1
```

```
  %t9 = add i32 %t7, %t8
  %t6 = alloca i32
  store i32 %t9, i32* %t6
  %t10 = load i32, i32* %t2
  %t11 = load i32, i32* %t6
  %t12 = add i32 %t10, %t11
  ret i32 %t12
}
```

LLVM IR After `opt -gvn`

```
define i32 @main() {
entry:
  %t0 = alloca i32, align 4
  store i32 10, i32* %t0, align 4
  %t1 = alloca i32, align 4
  store i32 10, i32* %t1, align 4
  %t2 = alloca i32, align 4
  store i32 20, i32* %t2, align 4
  %t6 = alloca i32, align 4
  store i32 20, i32* %t6, align 4
  ret i32 40
}
```

Explanation

The pass detects that `a + b` is computed twice and eliminates the second computation. Furthermore, since both `a` and `b` are constants, constant propagation and folding are applied, resulting in a direct return of `40`.

4 InstCombine (Instruction Combining)

Description

The InstCombine pass performs peephole optimizations on LLVM IR, combining sequences of instructions into simpler ones. It is not a canonicalization pass but rather focuses on local simplification, folding constants, and eliminating unnecessary computations. It often runs multiple times during optimization pipelines.

Javalette Program

```
int main() {
  int a = 3;
  int b = 4;
  int x = (a * 1) + (b * 0);
  return x;
}
```

LLVM IR Before Optimization

```
define i32 @main() {
entry:
  %t0 = alloca i32
  store i32 3, i32* %t0
  %t1 = alloca i32
  store i32 4, i32* %t1
  %t3 = load i32, i32* %t0
  %t4 = mul i32 %t3, 1
  %t5 = load i32, i32* %t1
  %t6 = mul i32 %t5, 0
  %t7 = add i32 %t4, %t6
  %t2 = alloca i32
  store i32 %t7, i32* %t2
  %t8 = load i32, i32* %t2
  ret i32 %t8
}
```

LLVM IR After `opt -instcombine`

```
define i32 @main() {
entry:
  ret i32 3
}
```

Explanation

The instruction combining pass applies the following simplifications:

- `%a * 1 → %a`
- `%b * 0 → 0`
- `%a + 0 → %a`

This reduces the entire computation to a constant value (`3`), allowing the function to return it directly. These small local simplifications make the IR more efficient and prepare it for further optimizations like constant propagation and dead code elimination.