LLVM Pass Study

Bodong Jia, Hongguang Chen

May 28, 2024

Introduction

- Optimizations are implemented as Passes that traverse some portion of a program to either collect information or transform the program.These passes perform various transformations on the intermediate representation (IR) of programs, targeting areas such as memory management, loop optimization, and dead code elimination. By applying these passes, LLVM optimizes code to make it faster, smaller, and more maintainable.

mem2reg Pass

- Promote Memory to Register¶

  This pass promotes memory references to be register references. An alloca is transformed by using dominator frontiers to place phi nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate. This is just the standard SSA construction algorithm to construct "pruned" SSA form, which is a prerequisite for some other passes.

- Traverse all basic blocks of a function, build data flow diagrams between basic blocks. This data flow diagram describes the relationship between the definition and use of variables. Analyze the liveness of each variable. For variables whose liveness is within the entire basic block, replace the memory access assigned to them with the corresponding register operation. For variables whose liveness spans basic blocks, phi nodes need to be inserted between basic blocks for further judgment and processing.

- Javalette code

```
int main() {

  int c1 =11;
  int c2 =15;
  int c3 = c1+c2;
  printInt(c3);
  return 0;
}
```

- LLVM origin code

```


define i32 @main() {
entry:
  %c1 = alloca i32, align 4
  store i32 11, ptr %c1, align 4
  %c2 = alloca i32, align 4
  store i32 15, ptr %c2, align 4
  %c11 = load i32, ptr %c1, align 4
  %c22 = load i32, ptr %c2, align 4
  %add = add i32 %c11, %c22
  %c3 = alloca i32, align 4
  store i32 %add, ptr %c3, align 4
  %c33 = load i32, ptr %c3, align 4
  call void @printInt(i32 %c33)
  ret i32 0
}
```

- LLVM opt code

```
define i32 @main() {
entry:
  %add = add i32 11, 15
  call void @printInt(i32 %add)
  ret i32 0
}
```

sccp Pass

- This pass achieves constant folding by analyzing conditional expressions and propagating known constant values through the program's control flow graph. When SCCP identifies a conditional expression with known constant operands, it evaluates the expression to its constant result and replaces the original expression with the computed constant value. This optimization reduces unnecessary runtime computations and simplifies the program's control flow, ultimately improving performance.
- Javalette code same as above
- LLVM opt Pass

```
define i32 @main() {
entry:
  call void @printInt(i32 26)
  ret i32 0
}
```

dse Pass

- A trivial dead store elimination that only considers basic-block local redundant stores. It's an optimization technique used to identify and remove redundant store operations in a program, specifically those that write to memory locations which are not subsequently read before being overwritten
- Perform data flow analysis to determine the lifetime of each variable.During the data flow analysis, identify all store operations in each basic block. For each store operation, check if the stored value is read later in the program, if not be used, the store operation is considered "dead".Once dead stores are identified, remove these store operations from the code
- Javalette code

```
int main() {
    int x = 10;
    printInt(x);
    x = 20;

    return 0;
}
```

- LLVM origin code

```
define i32 @main() {
entry:
  %x = alloca i32, align 4
  store i32 10, ptr %x, align 4
  %x1 = load i32, ptr %x, align 4
  call void @printInt(i32 %x1)
  store i32 20, ptr %x, align 4
  ret i32 0
}
```

- LLVM opt Pass

```
define i32 @main() {
```

```
entry:
  %x = alloca i32, align 4
  store i32 10, ptr %x, align 4
  %x1 = load i32, ptr %x, align 4
  call void @printInt(i32 %x1)
  ret i32 0
}
```

indvar Pass

- This is an optimization technique aimed at simplifying and canonicalizing induction variables within loops. Induction variables are variables that change in a predictable way with each iteration of a loop. With the help of mem2reg pass, this pass will identify the induction variables and fold them.
- Javalette code

```
int main(){
    int x =0;
    int a =1;
    while(a<=10){
        x=x+a;
        a++;
    }
    printInt(x);
    return 0;
}
```

- LLVM origin code

```
define i32 @main() {
entry:
  %x = alloca i32, align 4
  store i32 0, ptr %x, align 4
  %a = alloca i32, align 4
  store i32 1, ptr %a, align 4
  br label %while.cond

while.cond:                                       ; preds = %while.loop, %entry
  %a1 = load i32, ptr %a, align 4
  %le = icmp sle i32 %a1, 10
  br i1 %le, label %while.loop, label %while.end

while.loop:                                       ; preds = %while.cond
  %x2 = load i32, ptr %x, align 4
  %a3 = load i32, ptr %a, align 4
  %add = add i32 %x2, %a3
  store i32 %add, ptr %x, align 4
  %a4 = load i32, ptr %a, align 4
  %add5 = add i32 %a4, 1
  store i32 %add5, ptr %a, align 4
  br label %while.cond

while.end:                                        ; preds = %while.cond
  %x6 = load i32, ptr %x, align 4
  call void @printInt(i32 %x6)
  ret i32 0
}


###enable mem2reg pass

define i32 @main() {
entry:
  br label %while.cond

while.cond:                                       ; preds = %while.loop, %entry
  %a.0 = phi i32 [ 1, %entry ], [ %add5, %while.loop ]
  %x.0 = phi i32 [ 0, %entry ], [ %add, %while.loop ]
  %le = icmp sle i32 %a.0, 10
  br i1 %le, label %while.loop, label %while.end

while.loop:                                       ; preds = %while.cond
```

```
  %add = add i32 %x.0, %a.0
  %add5 = add i32 %a.0, 1
  br label %while.cond

while.end:                                        ; preds = %while.cond
  call void @printInt(i32 %x.0)
  ret i32 0
}
```

- LLVM opt Pass

```
define i32 @main() {
entry:
  br label %while.cond

while.cond:                                       ; preds = %while.loop, %entry
  %a.0 = phi i32 [ 1, %entry ], [ %add5, %while.loop ]
  br i1 false, label %while.loop, label %while.end

while.loop:                                       ; preds = %while.cond
  %add5 = add nuw nsw i32 %a.0, 1
  br label %while.cond

while.end:                                        ; preds = %while.cond
  call void @printInt(i32 55)
  ret i32 0
}
```

loop-deletion Pass

- This pass aims at removing loops that are deemed unnecessary or redundant. It makes a decision on whether a loop should be deleted based on certain criteria. These criteria may include:

  The loop does not have any side-effects and does not affect program semantics. The loop is empty or contains only dead code. The loop is redundant and can be replaced by equivalent, more efficient code.

- Javalette code same as above
- LLVM opt Pass This pass will work after the indvars pass finished in the previous section. This pass is responsible for eliminating loops with non-infinite computable trip counts that have no side effects or volatile instructions, and do not contribute to the computation of the function's return value.

```
define i32 @main() {
entry:
  br label %while.end

while.end:                                        ; preds = %entry
  call void @printInt(i32 55)
  ret i32 0
}
```

dce Pass

- Dead code elimination is similar to dead instruction elimination, but it rechecks instructions that were used by removed instructions to see if they are newly dead. This pass identifies and removes code that is guaranteed to never be executed during program execution. It achieves this by statically analyzing code to determine its liveness, and then removing any code that is deemed unreachable or unnecessary. This optimization improves program performance and readability by reducing code size and simplifying control flow.
- Javalette code

```
void foo() {
    int a = 10;
    int b = 20;
    int c = a + b;

}
```

```
int main() {
    foo();
    return 0;
}
```

- LLVM origin code

```
define void @foo() {
entry:
  %a = alloca i32, align 4
  store i32 10, ptr %a, align 4
  %b = alloca i32, align 4
  store i32 20, ptr %b, align 4
  %a1 = load i32, ptr %a, align 4
  %b2 = load i32, ptr %b, align 4
  %add = add i32 %a1, %b2
  %c = alloca i32, align 4
  store i32 %add, ptr %c, align 4
  ret void
}

define i32 @main() {
entry:
  call void @foo()
  ret i32 0
}
```

- LLVM opt Pass

```
define void @foo() {
entry:
  ret void
}

define i32 @main() {
entry:
  call void @foo()
  ret i32 0
}
```

mergefunc Pass

- Mergefunc pass is designed to merge similar functions to reduce program size and improve execution efficiency. When there are multiple similar functions in a program, Mergefunc pass will try to merge them into a single function to reduce duplicate code and eliminate redundancy. Mergefunc Pass will determine which functions can be merged based on the results of the similarity analysis. Once the functions to be merged are identified, MergeFunctions Pass will start merging them. This involves moving the contents of one function into another function and updating accordingly everywhere where the function is called.
- Javalette code

```
int add(int a, int b) {
    return a + b;
}

int sum(int x, int y) {
    return x + y;
}

int main() {
    int result1 = add(3, 4);
    int result2 = sum(3, 4);
    printInt(result1);
    printInt(result2);
    return 0;
}
```

- LLVM origin code

```
define i32 @add(i32 %a, i32 %b) {
entry:
  %a1 = alloca i32, align 4
  store i32 %a, ptr %a1, align 4
  %b2 = alloca i32, align 4
  store i32 %b, ptr %b2, align 4
  %a3 = load i32, ptr %a1, align 4
  %b4 = load i32, ptr %b2, align 4
  %add = add i32 %a3, %b4
  ret i32 %add
}

define i32 @sum(i32 %x, i32 %y) {
entry:
  %x1 = alloca i32, align 4
  store i32 %x, ptr %x1, align 4
  %y2 = alloca i32, align 4
  store i32 %y, ptr %y2, align 4
  %x3 = load i32, ptr %x1, align 4
  %y4 = load i32, ptr %y2, align 4
  %add = add i32 %x3, %y4
  ret i32 %add
}

define i32 @main() {
entry:
  %add = call i32 @add(i32 3, i32 4)
  %result1 = alloca i32, align 4
  store i32 %add, ptr %result1, align 4
  %sum = call i32 @sum(i32 3, i32 4)
  %result2 = alloca i32, align 4
  store i32 %sum, ptr %result2, align 4
  %result11 = load i32, ptr %result1, align 4
  call void @printInt(i32 %result11)
  %result22 = load i32, ptr %result2, align 4
  call void @printInt(i32 %result22)
  ret i32 0
}
```

- LLVM opt Pass

```
define i32 @add(i32 %a, i32 %b) {
entry:
  %a1 = alloca i32, align 4
  store i32 %a, ptr %a1, align 4
  %b2 = alloca i32, align 4
  store i32 %b, ptr %b2, align 4
  %a3 = load i32, ptr %a1, align 4
  %b4 = load i32, ptr %b2, align 4
  %add = add i32 %a3, %b4
  ret i32 %add
}

define i32 @main() {
entry:
  %add = call i32 @add(i32 3, i32 4)
  %result1 = alloca i32, align 4
  store i32 %add, ptr %result1, align 4
  %sum = call i32 @add(i32 3, i32 4)
  %result2 = alloca i32, align 4
  store i32 %sum, ptr %result2, align 4
  %result11 = load i32, ptr %result1, align 4
  call void @printInt(i32 %result11)
  %result22 = load i32, ptr %result2, align 4
  call void @printInt(i32 %result22)
  ret i32 0
}

define i32 @sum(i32 %0, i32 %1) {
  %3 = tail call i32 @add(i32 %0, i32 %1)
  ret i32 %3
}
```