Compiler Design Notes

Ayush Raina

February 23, 2025

Understanding Intermediate Representations (IR)

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

/* Compile this using clang -O0 -S -emit-llvm code1.c -o code1_O0.ll */
```

Listing 1: Example Code 1

We get the following LLVM IR:

```llvm
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4 ;Allocate space for a 32-bit integer
  %4 = alloca i32, align 4 ;Allocate space for another 32-bit integer
  store i32 %0, ptr %3, align 4 ; Store the first argument in the first space
  store i32 %1, ptr %4, align 4 ; Store the second argument in the second space
  %5 = load i32, ptr %3, align 4 ; Load the first argument
  %6 = load i32, ptr %4, align 4 ; Load the second argument
  %7 = add nsw i32 %5, %6 ; Add the two arguments
  ret i32 %7 ; Return the result
}
```

Listing 2: LLVM IR Example

Due to **optnone** attribute, we cannot apply any optimizations to that function. As a result we can see that values of the arguments are stored in memory and then loaded back before the addition operation and not directly used. Next we will compile the same code with **-O1** flag and see the difference in the IR generated.

```llvm
; Function Attrs: mustprogress nofree norecurse nosync nounwind willreturn memory(none) uwtable
define dso_local i32 @add(i32 noundef %0, i32 noundef %1) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0 ; Performs addition,  nsw - No Signed Wrap which means signed integer overflow should be ignored.
  ret i32 %3 ; Return the result
}
```

Listing 3: Optimized LLVM IR

Consider another example:

```c
int add(int a, int b) {
    int c = a + b;
    return c * 2;
}
```

Listing 4: Example Code 2

```llvm
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4 ; Allocate space for a 32-bit integer
  %4 = alloca i32, align 4 ; Allocate space for another 32-bit integer
  %5 = alloca i32, align 4 ; Allocate space for another 32-bit integer
  store i32 %0, ptr %3, align 4 ; Store the first argument in the first space
  store i32 %1, ptr %4, align 4 ; Store the second argument in the second space
  %6 = load i32, ptr %3, align 4 ; Load the first argument
  %7 = load i32, ptr %4, align 4 ; Load the second argument
  %8 = add nsw i32 %6, %7 ; Add the two arguments
  store i32 %8, ptr %5, align 4 ; Store the result of the addition
  %9 = load i32, ptr %5, align 4 ; Load the result of the addition
  %10 = mul nsw i32 %9, 2 ; Multiply the result by 2
  ret i32 %10 ; Return the result
}
```

Listing 5: Generated LLVM IR

```
; Function Attrs: mustprogress nofree norecurse nosync nounwind willreturn memory(none) uwtable
define dso_local range(i32 -2147483648, 2147483647) i32 @add(i32 noundef %0, i32 noundef %1)
    local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  %4 = shl nsw i32 %3, 1 ; shl - Shift Left equivalent to multiplying by 2
  ret i32 %4
}
```

Listing 6: Optimized LLVM IR

Next consider a example with a loop:

```
void loop_example(int *arr, int n) {
    for (int i = 0; i < n; i++) {
        arr[i] += 1;
    }
}
```

Listing 7: Example Code 3

```
define dso_local void @loop_example(ptr noundef %0, i32 noundef %1) #0 {
    %3 = alloca ptr, align 8
    %4 = alloca i32, align 4
    %5 = alloca i32, align 4
    store ptr %0, ptr %3, align 8 ; Store the pointer to the array
    store i32 %1, ptr %4, align 4 ; Store the (n) value
    store i32 0, ptr %5, align 4 ; Store counter value (i)
    br label %6

6:                                                ; preds = %17, %2, block for checking loop condition
    %7 = load i32, ptr %5, align 4 ; Load the counter value
    %8 = load i32, ptr %4, align 4 ; Load the (n) value
    %9 = icmp slt i32 %7, %8 ; Compare the counter value with (n), icmp - signed less than
    br i1 %9, label %10, label %20 ; If true go to label 10 else go to label 20

10:                                               ; preds = %6, increments array element
    %11 = load ptr, ptr %3, align 8 ; Load the pointer to the array
    %12 = load i32, ptr %5, align 4 ; Load the counter value
    %13 = sext i32 %12 to i64 ; Sign extend the counter value to 64-bit
    %14 = getelementptr inbounds i32, ptr %11, i64 %13 ; Get the pointer to the i-th element
    %15 = load i32, ptr %14, align 4 ; Load the i-th element
    %16 = add nsw i32 %15, 1 ; Add 1 to the i-th element
    store i32 %16, ptr %14, align 4 ; Store the result at the i-th element back.
    br label %17

17:                                               ; preds = %10, increments i
    %18 = load i32, ptr %5, align 4 ; Load the counter value
    %19 = add nsw i32 %18, 1 ; Increment the counter value
    store i32 %19, ptr %5, align 4 ; Store the incremented value
    br label %6, !llvm.loop !6 ; Go back to the loop condition

20:                                               ; preds = %6
    ret void
}
```

Listing 8: Generated LLVM IR

```
define dso_local void @loop_example(ptr noundef captures(none) %0, i32 noundef %1) local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 0 ; checks n > 0, sgt - signed greater than
  br i1 %3, label %4, label %6

4:                                                ; preds = %2
  %5 = zext nneg i32 %1 to i64 ; Zero extend the value of n to 64-bit, (left padding with 0s)
  br label %7

6:                                                ; preds = %7, %2
  ret void

7:                                                ; preds = %4, %7
  %8 = phi i64 [ 0, %4 ], [ %12, %7 ] ; Phi node to select between 0 and the value of i
  %9 = getelementptr inbounds nuw i32, ptr %0, i64 %8 ; Get the pointer to the i-th element
  %10 = load i32, ptr %9, align 4, !tbaa !5 ; Load the i-th element
  %11 = add nsw i32 %10, 1 ; Add 1 to the i-th element
  store i32 %11, ptr %9, align 4, !tbaa !5 ; Store the result back at the i-th element
  %12 = add nuw nsw i64 %8, 1 ; Increment the counter value
  %13 = icmp eq i64 %12, %5 ; Check if the counter value is equal to n
  br i1 %13, label %6, label %7, !llvm.loop !9 ; If true go to label 6 else go to label 7
}
```

Listing 9: Optimized LLVM IR