# 605.615 Compiler Design with LLVM
# Optimization Research

Sabbir Ahmed

March 27, 2022

## 1 Introduction

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse maximus eu lorem interdum dignissim. Mauris imperdiet euismod nisl finibus fringilla. Curabitur sagittis nec ligula nec tempus. Phasellus quis fringilla massa. Aliquam malesuada libero at dolor rutrum consequat. Nulla efficitur dictum laoreet. Etiam feugiat turpis at condimentum euismod. Aliquam leo magna, egestas ac sagittis vel, fringilla at dui. Aliquam ut enim justo. Vestibulum et velit id leo tristique viverra. Integer sagittis placerat augue eget condimentum. Integer hendrerit ultricies dui. Aliquam tincidunt porttitor ornare. [1]

## 2 `argpromotion`

The `argpromotion` pass implements the argument promotion optimization. This optimization is a transform pass that aims to convert function arguments that are passed "by reference" to "by value". [2] This "argument promotion" is performed when the compiler determines the values passed do not have any store instructions performed on them. By converting the references, the pass optimizes away unnecessary instructions of storing the arguments into temporary registers.

Consider the following listing:

Listing 1: A method to return the sum of 2 integers

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

void callAdd() {
    int a = 1;
    int b = 2;
    int c = add(&a, &b);
}
```

The `add` method expects references to 2 integers as its arguments. The method makes references to the variables to retrieve their values and compute the sum. This operation does not modify the variables. Therefore, the compiler can be expected to optimize the `add` method by converting its arguments to be passed by value. However, running clang without optimizations generate something similar to the following:

Listing 2: `add` method compiled without optimizations

```c
int add(int* a, int* b) {
    int tmpa;
    int tmpb;
    tmpa = *a;  // store
    tmpb = *b;  // store
    int sum;
    sum = tmpa + tmpb;  // load
    return sum;
}

void callAdd() {
    int a;
    int b;
    a = 1;
    b = 2;
    int c;
    c = add(&a, &b);
}
```

Whereas the compiler is expected to generate something similar to the following:

Listing 3: Expected `add` method compiled with optimizations

```
int add(int a, int b) {
    int sum;
    sum = a + b;  // load
    return sum;
}

void callAdd() {
    int a;
    int b;
    a = 1;
    b = 2;
    int c;
    c = add(a, b);
}
```

For the optimization, the `argpromotion` pass must be able to perform the following tasks:

1. change the prototype of the `add` method
2. update all the call sites according to the modified prototype
3. know the reference does not alias other pointers in the `add` method
4. know the loaded values remain unchanged from function entry to the load
5. know the reference is not being used to store

The pass expects other transform passes to be ran in conjunction, including `instcombine` to combine redundant instructions and `function-attrs` to look for functions which do not access or only read non-local memory.[1]

## 3 `simple-loop-unswitch`

The `simple-loop-unswitch` pass performs the loop unswitch optimization. This optimization is a transform pass that aims to reduce branching inside of loops. A loop is ”unswitched” when the conditionals inside get extracted outside the loop. The following listing demonstrates a simple loop unswitch:

Listing 4: Before Unswitching

```
// lic stands for loop-invariant
// code
for (...) {
    A
    if (lic) {
        B
    }
    C
}
```

Listing 5: After Unswitching

```
if (lic) {
    for (...) {
        A; B; C
    }
} else {
    for (...) {
        A; C
    }
}
```

The loop ends up getting duplicated for every conditional branches with loop-invariant expressions or statements. This optimization allows for individual loops to be safely parallelized and hurter optimized. However, the optimization may on average double the amount of instructions written. [4]

LLVM provides options to determine the form of transformation for this optimization. The switch can be either trivial (when the condition can be unswitched without cloning any code from inside the loop) or non-trivial (requires code duplication). By default, the pass always does trivial, full-unswitching (when the branch or switch is completely moved from inside the loop to outside the loop) for the branches and switches. [3]

The pass expects the `licm` optimization to be ran before it to hoist invariant conditions out of the loop.

## References

[1] LLVM's Analysis and Transform Passes. `https://llvm.org/docs/Passes.html`.

[2] ArgumentPromotion.cpp File Reference. `https://llvm.org/doxygen/ArgumentPromotion_8cpp.html`.

[3] SimpleLoopUnswitch.cpp File Reference. `https://llvm.org/doxygen/SimpleLoopUnswitch_8cpp.html`.

[4] llvm::SimpleLoopUnswitchPass Class Reference. `https://llvm.org/doxygen/classllvm_1_1SimpleLoopUnswitchPass.html`.