ENPM617 Final Project Report

I. Introduction

The LLVM is a compiler that is now maintained by Apple Inc. The goal of this project is to implement the inline function into the LLVM compiler. To be specific, inlining means when a function is called, copy the callee's code into the call-site and the original callee function is not deleted. Inlined functions will have the same functionality as the original code.

```
main () {
      int diff;
      diff1 = sub(3, 7);
      printf("diff = %d\n", diff);
}
int sub(int x, int y) {
       int z;
       z = x – y;
       return z;
}
```

The above C code needs to be transformed into:

```
main () {
      int diff;

      int x = 3;
      int y = 7;
      int z;
       z = x – y;
      diff = z;

      printf("diff = %d\n", diff);
}
```

In this project, inlining is done only when there are less than 10 instructions and all the arguments in an instruction are all constants. Furthermore, we are assuming that each function will only have one call-site, meaning each function will only be called once. The image on the left is an example provided in the project description.

II. Design and Implementations

1. Iterate over all the instructions in the main function.
    - In this step, we will cast each instruction into a CallInst, and from the CallInst, we can use the getCalledFunction() to get the callee.
    - Using inst_begin() and inst_end() we can iterate over the instruction list of a function.
2. Iterate over all the arguments of an instruction.
    - After we have the instruction, we can access the arguments and values inside that instruction.
    - arg_begin () and arg_end() will iterate over the argument list of an instruction. getOperand() would get the value of the constant in the instruction.

3. Check whether an argument is constant or not
    - If one or more arguments are not constant, then we do not inline this instruction and skip to the next instruction.
4. Create a constant symbol to convert constants into local variables
    - In order to convert all the constants into local variables, we need to create a new ConstantInt using the value we got from the formal argument.
5. Replace all uses of the formal argument with constants we just created
    - By using replaceAllUsesWith(x), we can replace all the formal arguments with the new ConstantInt we just created. This would modify the callee's instruction by replacing those values. For example, if in the main function we are assigning "x" the value of "7", this step will replace the use of "x" in the calle into the int value "7".
6. Copy the modified instruction from the callee to the call-site.
    - In this step, we copied all of the modified instructions from callee to the call-site. The effective steps are clone(), insert and Remap.
7. Remove the call instruction
    - Lastly, we need to use eraseFromParent() to remove the original instruction in the main functions.

The implementation of our project follows the above steps. First, after we get the main function, we will iterate all the instructions to check whether main() has more than 10 instructions or not. If main() has more than 10 instructions, inline will not happen.

```
// errs() << F.getName() << "\n";
if (F.getName() == "main") {
  errs() << "Got Main\n";

  // check if main has over 10 instructions
  unsigned counter = 0;
  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    if (counter > 10) {
      return PreservedAnalyses::all();
    } else {
      counter++;
    }
  }
```

Then we will loop through all of the instructions in the main() function using the inst_iterator. The erase_inst list will store all the instruction that needs to be

erased/inlined, after the loop ends. Because we cannot delete instructions inside the instructions loop otherwise the loop will fail on the next round after we delete the current instruction.

```
// Loop over every instruction in main
std::list<CallInst *> erase_inst;
for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
```

The following code will first cast the instruction into CallInst and get the called function from it. Then it checks whether this function is a declared function or not. The for loop uses arg_iterator to iterate through all of the arguments inside this function. This for loop will check if all of the arguments in the function are constant are not. If any of them is not a constant, the has_non_constnats will be set to true and the loop will break.

```
CallInst *current_inst = cast<CallInst>(&*I);
Function *f = cast<CallInst>(*I).getCalledFunction();

// Check if f is a declaired function
if (!f->isDeclaration()) {
  unsigned argcounter = 0;
  bool has_non_constants = false;

  // Check if all of arguments in this instruction is constansts
  // If not, break the loop
  for (Function::arg_iterator argI = f->arg_begin();
       argI != f->arg_end(); ++argI) {
    Value *value = cast<CallInst>(*I).getArgOperand(argcounter);
    // Check an argument is a constant or not
    if (!isa<Constant>(*value)) {
      has_non_constants = true;
      break;
    }
    argcounter++;
  }
```

If all of the arguments are constants, we will insert the current instruction into the erase_inst to delete the instruction in the future. Then we will go through the argument lists again, but this time we will create a ConstantInt for each argument and replace the formal arguments with the newly created local variables.

```
// If all the arguments are constants, we can inline this instruction
if (has_non_constants == false) {
  // Push this instruction into the erase_inst list for erasing the
  // instruction in the future
  erase_inst.push_back(current_inst);
  unsigned second_argcounter = 0;
  // Loop through every argument in the instructions to create and
  // replace the formal argument
  for (Function::arg_iterator argI = f->arg_begin();
       argI != f->arg_end(); ++argI) {
    Value *value =
        cast<CallInst>(*I).getArgOperand(second_argcounter);
    // Creating a ConstantInt
    LLVMContext &context = value->getContext();
    ConstantInt *v = ConstantInt::get(
        context, cast<ConstantInt>(*value).getValue());
    // Replace all uses of formal argument
    argI->replaceAllUsesWith(v);
    second_argcounter++;
  }
```

After we replace all the arguments, we will clone and insert each instruction from the callee to the call-site. The clone() function will clone the modified instruction into a new instruction we created and we will insert the instruction before the current instruction that we are at in the loop. The value map would store all the new instructions and save them for Remap in the future.

```
  // Clone and insert each instruction
llvm:
  ValueToValueMapTy vmap;
  // Created a list of cloned instructions
  std::list<Instruction *> cloned_instructions;
  // Loop over every instructions in the called function
  for (inst_iterator calledI = inst_begin(f), calledE = inst_end(f);
       calledI != calledE; ++calledI) {
    // Copy over the instructions
    const Value *inst = cast<Value>(&*calledI);
    auto *new_inst = calledI->clone();
    new_inst->insertBefore(current_inst);
    cloned_instructions.push_back(new_inst);
    vmap[inst] = new_inst;
    errs() << *new_inst << "\n";
  }
```

After Remapping all the instructions, we will also need to handle the “return” instruction from the callee. Obviously we do not need 2 “return” instructions in the main function, but cannot just simply delete it. We first check if the “return” instruction has a return value or not. If the return value is not void, we need to replace all uses of this value and then delete the “return” instruction. If the return value is void, we simply delete it.

```
// Remap Instructions
for (auto *i : cloned_instructions) {
  RemapInstruction(
      i, vmap, RF_NoModuleLevelChanges | RF_IgnoreMissingLocals);
}

// For the last instruction, which is return, we determine whehter
// return type is not void, replace all use of it.
// Then we remove the return instruction that is cloned from the
// called function
Instruction *last_inst = cloned_instructions.back();
Value *last_return_value =
    cast<ReturnInst>(last_inst)->getReturnValue();
if (last_return_value != NULL) {
  // Return type is NOT void
  current_inst->replaceAllUsesWith(last_return_value);
}
// else, Return type is void, no need to replace
// Delete the remove instruction
last_inst->eraseFromParent();
```

Lastly outside the instruction loop, we will erase all of the instructions stored in the erase_inst list.

```
// Remove Instrucitons
for (auto *i : erase_inst) {
  i->eraseFromParent();
}
```

III. Testings

In this project, we tested our implementation with the provided test in the glue machine and the project description. We selected a few test examples from the glue machine and copied the example included in the project description. The corresponding files are example0.c, example1.c and example2.c.

IV. Result

The results are shown in the example_inline.ll file. The example.ll is the original IR file for the test and the example_inline.ll is the modified IR file by our inline function. We can observe that all of the instructions inside the callee are added to the main

```
void pow2(int x)
{
    int y;
    y = x*x;
}

int main ()
{
    pow2(7);
    printf("calculated pow2 of 7");
    return 0;
}
```

function. The modified IR file will have the same functionality as the original file. Below is the comparison of the original example1.ll and example1_inline.ll.

```
define void @pow2(i32 %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 7, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = load i32, i32* %2, align 4
  %6 = mul nsw i32 %4, %5
  store i32 %6, i32* %3, align 4
  ret void
}

; Function Attrs: noinline nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 7, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = load i32, i32* %2, align 4
  %6 = mul nsw i32 %4, %5
  store i32 %6, i32* %3, align 4
  %7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([21 x i8], [21 x i8]*
  ret i32 0
}
```

example1_inline.ll

```
define void @pow2(i32 %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = load i32, i32* %2, align 4
  %6 = mul nsw i32 %4, %5
  store i32 %6, i32* %3, align 4
  ret void
}

; Function Attrs: noinline nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  call void @pow2(i32 7)
  %2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([21 x i8], [21 x i8]*
  ret i32 0
}
```

example1.ll

V. Issue

For this project, there are some issues that I think should be changed or be mentioned. First of all, for the path to the llvm glue machine (correct one):

"setenv PATH /afs/glue.umd.edu/class/old/enee/759c/llvm/llvm-3.4-install/opt/bin/:$PATH"

For the library part, there are some changes for the library. The InstIterator file was different between the glue machine and llvm 14 version. For the glue machine should be:

```
#include "llvm/Support/InstIterator.h"
```

But the VM linux llvm 14 version:

```
#include "llvm/IR/InstIterator.h"
```

Another issue that I encountered was the command to convert the bytecode to the IR file. For the glue machine the command in terminal should be:

```
./bin/opt -load lib/LLVMHello.so -hello  < hello.bc > hello.ll
```

The VM linux llvm version 14 version requires an extra extension which is adding "-enable-new-pm=0":

```
./bin/opt -load lib/LLVMOurPass.so -OurPass  -enable-new-pm=0 < hello.bc > hello.ll
```

For the glue machine, the register pass was require to add this in our pass:

```
static RegisterStandardPasses Y(
    PassManagerBuilder::EP_EarlyAsPossible,
    [](const PassManagerBuilder &Builder,
       legacy::PassManagerBase &PM) { PM.add(new Hello()); });
```

For the llvm 14th version, we do not need to add this in our pass.

The last issue is about logging into the glue machine, our teammate suffered an issue about logging into the glue machine that the professor provided us. He got permission denied for logging in to the system. Maybe this is another issue that has students checking their account first and to inform the IT department in Maryland.

VI. Conclusion

In conclusion, the inline function was successfully implemented but this is a limited version of the inline function. The real inline function should not have restrictions like maximum 10 instructions or arguments are all constants. Nevertheless our design and implementation should still increase some of the compiling speed of the compiler.