

# ENPM617 LLVM FUNCTION CLONING PASS

## Table of Contents

1.INTRODUCTION.....	2
1.1 Project Overview.....	2
1.2 Setting the environment.....	3
2. IMPLEMENTATION OF FUNCTION CLONING PASS.....	4
2.1 Algorithm .....	4
2.2 Pass generation.....	4
2.3 Libraries used .....	7
3. MODIFIED MAKE FILE FOR COMPILING THE PASS .....	8
4. RESULTS.....	9
5. ISSUES.....	14
6. CONCLUSION.....	15

## CHAPTER 1

### INTRODUCTION

#### 1.1) PROJECT OVERVIEW:

LLVM is the name of the compiler which is open-source software. The primary goal of this project is to implement a **function cloning pass** that can automatically transform indirect call-transfer instructions to direct unconditional control-transfer instructions. This is because indirect call-transfer instructions can be slow on architectures that do not have suitable branch prediction hardware.

In this project, we have to clone the function which starts with the letter “p”.

#### Sample C program:

Let us consider the sample C program which is operating on indirect call-transfer instructions.

```
main ()
{
    int a = 10;
    int p;
    p = pow2(a);
    printf("pow2 of A = %d\n", p);
}
int pow2(int x) {
    return x * x;
}
```

fig 1.1

The above-mentioned C program has to transform to direct control-transfer instructions by cloning the function **pow2()**.

```
int g = 0;           //Global Variable Declaration
main ()
{
    int a = 10;
    int p;
    pow2_clone(a);   //Cloned function
    label:
        p = g;
        printf("pow2 of A = %d\n", p);
}
int pow2(int x) {
    return x * x;
}
void pow2_clone(int x) {
    g = x * x;
    __asm__(pop);
    goto label;
    return;
}
```

fig 1.2

`_asm_(pop)` and `goto label` instructions are replaced by using `pop_direct_branch()`. This function provides the exact same functionality of `pop` instruction followed by the `goto` statement.

**1.2) SETTING THE ENVIRONMENT:** We have used the GLUE machine for the completion of this project which has an installed version of LLVM 3.4. We have followed the below procedure to begin our project.

- 1) Setting the environment to our current user id using  
`setenv PATH /afs/glue.umd.edu/class/old/enee/759c/llvm/llvm-3.4-install/opt/bin/:$PATH`
- 2) Copying the test codes to our current user directory
- 3) Before starting the pass generation, we used a prewritten hello program to confirm that the procedure would work with LLVM.

```
#define DEBUG_TYPE "hello"
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
struct Hello : public FunctionPass
{
    /** Constructor. */
    static char ID;
    Hello() : FunctionPass(ID) {}

    //DEFINE_INTPASS_ANALYSIS_ADJUSTMENT(PointerAnalysisPass);

    /**
     * @brief Runs this pass on the given function.
     * @param [in,out] func The function to analyze
     * @return true if the function was modified; false otherwise
     */
    virtual bool runOnFunction(llvm::Function &F){
        errs() << "Hello: " ;
        errs().write_escaped(F.getName())<< "\n";
        return false;
    }
};

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass", false, false);
```

.cpp for hello program

```
#include <stdio.h>
#include "pop_direct_branch.c"

void scan_int(int *x)
{
    scanf("%d", x);
    return;
}

void print_int(int x)
{
    printf("%d \n", x);
    return;
}

int main()
{
    int a;
    scan_int(&a);
    printf("The entered value is: ");
    print_int(a);
    return 0;
}

Hello: pop_direct_branch
Hello: scan_int
Hello: print_int
Hello: main
```

Test case and respective output for hello program

## CHAPTER 2 : IMPLEMENTATION OF FUNCTION CLONING PASS:

### 2.1) Algorithm:

STEP 1: Start

STEP 2: Declare a global variable to store the return value of the function which is cloned.

STEP 3: Iter over every module instruction and check for call instructions.

STEP 4: Get the called function.

STEP 5: Clone the called function if the called function name starts with the letter “p” and not a declaration.

STEP 6: Create a new store instruction that transfers the output of the return value of the cloned instruction into the global variable.

STEP 7: Call the pop\_direct\_branch function

STEP 9: Create a new Load instruction for loading the return value from the global variable to the appropriate variable.

STEP 10: Replace the call site with the newly cloned function.

STEP 11: return true for pass transformation

STEP 12: Register the pass

STEP 13: End

### 2.2) Pass Generation

1) Consider the *ModulePass* class where we consider an entire program as a unit and modify the module by the transformation using *virtual bool runOnModule(Module &M)*.

```
using namespace llvm;

namespace {
    struct finalproject : public ModulePass {
        static char ID;
        finalproject() : ModulePass(ID) {}
        virtual bool runOnModule(Module &M) override {
            LLVMContext &C = M.getContext();
            GlobalVariable *g = new GlobalVariable(M, IntegerType::get(M.getContext(), 32), false, GlobalValue::CommonLinkage, 0, "g");
            g->setAlignment(4);
            ConstantInt* const_int32_1 = ConstantInt::get(C, APInt(32, 0));
            g->setInitializer(const_int32_1);
            Function *output = M.getFunction("pop_direct_branch");
        }
    };
}
```

A new GlobalVariable needs to be added to the Module, where Module is assumed to be M. By converting the global variable declaration's.c file to a.cpp file, the appropriate constructor and necessary attributes for creating a new Global variable are extracted. Since the module

needs to contain the function *pop\_direct\_branch*. The function will therefore be added to the module by calling *getFunction()*.

2) We must iterate over the module's instructions in order to check each instruction for the call function and cast each instruction to *CallInst* in order to determine whether it is a call instruction.

```
for(Module::iterator b = M.begin(); b != M.end(); ++b){for(Function::iterator f = b->begin(); f != b->end(); ++f){
    for(Function::iterator f = b->begin(); f != b->end(); ++f){
        for(BasicBlock::iterator i = f->begin(); i != f->end(); ++i){
            Instruction *I = &*i; // iterating over every instruction in the module
            if (CallInst *call = dyn_cast<CallInst>(I)){ // casting the instruction to CallInst
                if(Function *fun = call->getCalledFunction()){ // get the function which has been called
                    StringRef start = fun->getName(); // obtaining the function name
                    errs() << start << "\n";
                    if(start[0] == 'p' && !fun->isDeclaration()){
                        if(start != "pop_direct_branch"){
```

It is required to get the called function to check whether the name of the function starts with the letter "p" or whether the function is a declaration. *getCalledFunction()* obtains the function which has been called and *getName()* gives the name of the function.

3) Clone the function and include it in the module if it is not a declaration and if the function begins with the letter "p". Function is duplicated by *CloneFunction()*. *getParent()* retrieves the module that the original function belongs to, and it is necessary to obtain the list of functions inside the module using *getFunctionList()*. The cloned function is pushed inside the module using the *push\_back()* function.

```
llvm::ValueToValueMapTy VMap;
// clone the function and insert the clone function inside the module
Function *clone = CloneFunction(fun,VMap,false);
fun->getParent()->getFunctionList().push_back(clone);
```

4) It is necessary to iterate through the function's instructions after cloning the function in order to check for return instructions in the clone function, and to store the return value in a global variable. This is accomplished by getting the return value using *getReturnValue* and casting each instruction to the *ReturnInst* class. The return value from the global value is stored using the *StoreInst* class. Load the return value from the global variable into the appropriate variable after where the call instruction is cloned in the original function. *IRBuilder* is used for creating the load instruction through *CreateLoad()*.

```

for(Function::iterator F = clone->begin(); F != clone->end(); ++F){
    for(BasicBlock::iterator i = F->begin(); i != F->end(); ++i){
        Instruction *I = &*i;
        errs() << *I << "\n";
        // obtaining the return value
        if(ReturnInst *return_val = dyn_cast<ReturnInst>(&*I)){
            if(Value* return_clone = return_val->getReturnValue()){
                // storing the return value to the global variable
                StoreInst *store = new StoreInst(return_clone, g, return_val);
                Instruction *output_pll = CallInst::Create(output, "", return_val);
                //Loading the return value from the global variable
                IRBuilder<>builder(call->getNextNode());
                LoadInst *load = builder.CreateLoad(g, "");
                call->replaceAllUsesWith(load);
            }
        }
    }
}

```

Replace all the uses using the new load instruction using *replaceAllUsesWith()*

- 6) Since we have replaced the old function with the new cloned function. It is required to set the call to a new cloned function instead of the old function. *setCalledFunction()* is used to set the call from the old function to the new cloned function.

```

    }
    call->setCalledFunction(clone);
}
errs() << "after modification" << *clone << "\n";

return true;

;

char finalproject::ID = 0;
static RegisterPass<finalproject> X("finalproject", "finalproject", false, false);

```

As the pass is transformed, we need to *return true*. The pass has to be registered for the implementation of the changes to the user programs.

## 2.3) LIBRARIES USED

The following libraries are used for extracting the appropriate class and constructors for completing the project.

```
#define DEBUG_TYPE "finalproject"
#include <llvm/IR/LegacyPassManager.h>
#include <llvm/Pass.h>
#include <llvm/Analysis/Verifier.h>
#include <llvm/Assembly/PrintModulePass.h>
#include <llvm/IR/BasicBlock.h>
#include <llvm/IR/CallingConv.h>
#include <llvm/IR/Constants.h>
#include <llvm/IR/DerivedTypes.h>
#include <llvm/IR/Function.h>
#include <llvm/IR/GlobalVariable.h>
#include <llvm/IR/InlineAsm.h>
#include <llvm/Support/raw_ostream.h>
#include <llvm/IR/Function.h>
#include <llvm/IR/Instruction.h>
#include <llvm/IR/Instructions.h>
#include <llvm/ADT/SetVector.h>
#include <llvm/ADT/SmallVector.h>
#include <llvm/ADT/Twine.h>
#include <llvm/Analysis/InlineCost.h>
#include <llvm/Transforms/Utils/ValueMapper.h>
#include <llvm/IR/BasicBlock.h>
#include <llvm/IR/SymbolTableListTraits.h>
#include <llvm/IR/LLVMContext.h>
```

```
#include <llvm/IR/Module.h>
#include <llvm/Support/FormattedStream.h>
#include <llvm/Support/MathExtras.h>
#include <llvm/IR/User.h>
#include <llvm/IR/Value.h>
#include <llvm/IR/InstrTypes.h>
#include <llvm/ADT/StringRef.h>
#include <llvm/IR/DerivedTypes.h>
#include <llvm/Transforms/Utils/Cloning.h>
#include <llvm/IR/InstrTypes.h>
#include <llvm/ADT/ArrayRef.h>
#include <llvm/IR/IRBuilder.h>
#include <llvm/IR/Type.h>
#include <llvm/Support/InstIterator.h>
#include <llvm/Support/Compiler.h>
#include <iterator>
#include <functional>
#include <memory>
#include <vector>
#include <algorithm>
#include <utility>
#include <cassert>
```

### CHAPTER: 3

#### MODIFIED MAKE FILE FOR COMPILING THE PASS

The project sub-directory must be included in the make file for compiling the pass and a new make file has been created for the program generated.

```
##===- projects/sample/lib/Makefile -----*- Makefile -*===##
#
# Relative path to the top of the source tree.
#
LEVEL=..
#
# List all of the subdirectories that we will compile.
#
#DIRS=Scratch
DIRS=Hello
DIRS=finalproject
include $(LEVEL)/Makefile.common
```

*DIRS= finalproject* is the added sub-directory in the make file for compiling the pass.

```
LEVEL = ../..

LIBRARYNAME = finalproject
BUILD_RELINKED=1
SHARED_LIBRARY=1
#LOADABLE_MODULE = 1

include $(LEVEL)/Makefile.common
```

Created make file for the program to get executed with the appropriate requirements.



## CHAPTER: 4

### RESULTS

We have loaded the program in the test cases provided on the GLUE machine. We have two differentiated examples for testing. We have extracted the modified .ll files for the test cases provided.

#### Example 0:

```
#include <stdio.h>
#include "pop_direct_branch.c"

void scan_int(int *x)
{
    scanf("%d", x);
    return;
}

void print_int(int x)
{
    printf("%d \n", x);
    return;
}

int main()
{
    int a;
    scan_int(&a);
    printf("The entered value is: ");
    print_int(a);
    return 0;
}
```

The .c file of example0 is provided above.

```
define void @pop_direct_branch() #0 {
    call void @asm.sideeffect "popq %rbp@09addq $58, %rsp@0A\09leave@0A\09movq (%rsp), %rax@0A\09addq $58, %rsp@0A\09", !srcloc !1
    ret void
}

; Function Attrs: nounwind uwtable
define void @scan_int(i32* %x) #0 {
    %1 = alloca i32*, align 8
    store i32* %x, i32** %1, align 8
    %2 = load i32** %1, align 8
    %3 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32* %2)
    ret void
}

declare i32 @__isoc99_scanf(i8*, ...) #1

; Function Attrs: nounwind uwtable
define void @print_int(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32* %1, align 4
    %3 = call i32 @__isoc99_printf(i8* getelementptr inbounds ([5 x i8]* @.str1, i32 0, i32 0), i32 %2)
    ret void
}

declare i32 @__isoc99_printf(i8*, ...) #1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %a = alloca i32, align 4
    store i32 0, i32* %1
    call void @scan_int(i32* %a)
    %2 = call i32 @__isoc99_printf(i8* getelementptr inbounds ([23 x i8]* @.str2, i32 0, i32 0))
    %3 = load i32* %a, align 4
    call void @print_int(i32 %3)
    ret i32 0
}
```

.ll file for example0

The example0 test case has four functions including `pop_direct_branch`. According to our algorithm. Cloning of the function should be expected only when the function name starts with the letter “p” and it is not a declaration. After cloning the function if the function does not expect any return value there is no necessity for the addition of instructions.

```
define void @pop_direct_branch() #0 {
    call void @asm_sideeffect "popq %rbp\0A\09addq $$8, %rsp\0A\09leave\0A\09movq (%rsp), %rax\0A\09addq $$8, %rsp\0A\09", !srcloc !1
    ret void
}

; Function Attrs: nounwind uwtable
define void @scan_int(i32* %x) #0 {
    %1 = alloca i32*, align 8
    store i32* %x, i32** %1, align 8
    %2 = load i32** %1, align 8
    %3 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32* %2)
    ret void
}

declare i32 @__isoc99_scanf(i8*, ...) #1

; Function Attrs: nounwind uwtable
define void @print_int(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32* %1, align 4
    %3 = call i32 @__isoc99_printf(i8* getelementptr inbounds ([5 x i8]* @.str1, i32 0, i32 0), i32 %2)
    ret void
}

declare i32 @__isoc99_printf(i8*, ...) #1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %a = alloca i32, align 4
    store i32 0, i32* %1
    call void @scan_int(i32* %a)
    %2 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([23 x i8]* @.str2, i32 0, i32 0))
    %3 = load i32* %a, align 4
    call void @print_int1(i32 %3)
    ret i32 0
}

; Function Attrs: nounwind uwtable
define void @print_int1(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32* %1, align 4
    %3 = call i32 @__isoc99_printf(i8* getelementptr inbounds ([5 x i8]* @.str1, i32 0, i32 0), i32 %2)
    ret void
}
```

The above code is the modified .ll file where there is no change from the old .ll file to the new .ll file. `print_int()` is the function before cloning and `print_int1()` is the function cloned. Since there is no return value for `print_int()` function no instruction has been added.

## Example1:

```
#include <stdio.h>
#include "pop_direct_branch.c"

void scan_int(int *x)
{
    scanf("%d", x);
    return;
}

int pow2(int x)
{
    return x*x;
}

void print_int(int x)
{
    printf("%d \n", x);
    return;
}

int main ()
{
    int a;
    scan_int(&a);
    int p;
    p = pow2(a);
    printf("pow2 of A = ");
    print_int(p);
    return 0;
}
```

.c file of Example1

```
; Function Attrs: nounwind uwtable
define void @scan_int(i32* %x) #0 {
    %1 = alloca i32*, align 8
    store i32* %x, i32** %1, align 8
    %2 = load i32** %1, align 8
    %3 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32* %2)
    ret void
}

declare i32 @__isoc99_scanf(i8*, ...) #1

; Function Attrs: nounwind uwtable
define i32 @pow2(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32* %1, align 4
    %3 = load i32* %1, align 4
    %4 = mul nsw i32 %2, %3
    ret i32 %4
}
```

```
; Function Attrs: nounwind uwtable
define void @print_int(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32* %1, align 4
    %3 = call i32 @__printf(i8* getelementptr inbounds ([5 x i8]* @.str1, i32 0, i32 0), i32 %2)
    ret void
}

declare i32 @__printf(i8*, ...) #1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %a = alloca i32, align 4
    %p = alloca i32, align 4
    store i32 0, i32* %1
    call void @scan_int(i32* %a)
    %2 = load i32* %a, align 4
    %3 = call i32 @pow2(i32 %2)
    store i32 %3, i32* %p, align 4
    %4 = call i32 @__printf(i8* getelementptr inbounds ([13 x i8]* @.str2, i32 0, i32 0))
    %5 = load i32* %p, align 4
    call void @print_int(i32 %5)
    ret i32 0
}
```

.ll of Example1

The example1 test case has five functions including *pop\_direct\_branch()*. There are two functions starting with the letter “p” i.e., *pow2()* and *print\_int()*. The *print\_int()* does not return any value. So, no new instructions are added to the function *print\_int()*. As *pow2()* has a return value we need to add the store and load instruction for storing the return value into the global variable and reading the value from the global variable by calling the function *pop\_direct\_branch()*.

```
; Function Attrs: nounwind uwtable
define void @scan_int(i32* %x) #0 {
    %1 = alloca i32*, align 8
    store i32* %x, i32** %1, align 8
    %2 = load i32** %1, align 8
    %3 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32* %2)
    ret void
}

declare i32 @__isoc99_scanf(i8*, ...) #1

; Function Attrs: nounwind uwtable
define i32 @pow2(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32* %1, align 4
    %3 = load i32* %1, align 4
    %4 = mul nsw i32 %2, %3
    ret i32 %4
}

; Function Attrs: nounwind uwtable
define void @print_int(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32* %1, align 4
    %3 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([5 x i8]* @.str1, i32 0, i32 0), i32 %2)
    ret void
}
```

```
; Function Attrs: nounwind uwtable
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %a = alloca i32, align 4
    %p = alloca i32, align 4
    store i32 0, i32* %1
    call void @scan_int(i32* %a)
    %2 = load i32* %a, align 4
    %3 = call i32 @pow2(i32 %2)
    %4 = load i32* @g
    store i32 %4, i32* %p, align 4
    %5 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([13 x i8]* @.str2, i32 0, i32 0))
    %6 = load i32* %p, align 4
    call void @print_int2(i32 %6)
    ret i32 0
}

; Function Attrs: nounwind uwtable
define i32 @pow21(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32* %1, align 4
    %3 = load i32* %1, align 4
    %4 = mul nsw i32 %2, %3
    store i32 %4, i32* @g
    call void @pop_direct_branch()
    ret i32 %4
}

; Function Attrs: nounwind uwtable
define void @print_int2(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32* %1, align 4
    %3 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([5 x i8]* @.str1, i32 0, i32 0), i32 %2)
    ret void
}
```

Modified .ll file of Example1

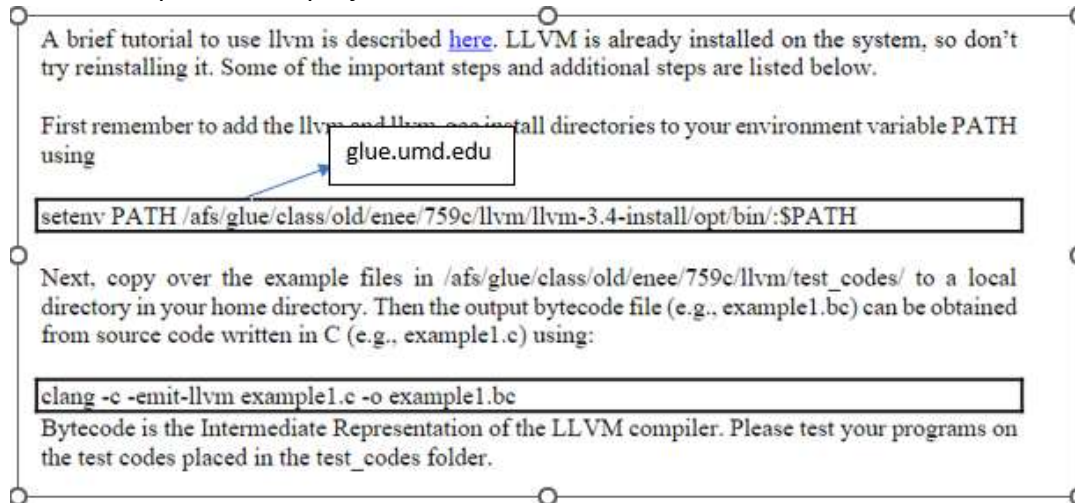
The above modified .ll file shows the cloned functions *pow21()* and *print\_int2()*. Since the *pow21()* function has a return value. The return value is stored inside the global variable (*store i32 %4, i32\* @g*). After the store instruction, we need to call the function *pop\_direct\_branch()* (*call void @pop\_direct\_branch()*). We intended to read the value from the global variable so load instruction is included in the main function(*%4 = Load i32\* @g*).

As the *print\_int2()* does not have any return value, no instruction has been inserted.

## CHAPTER 5: ISSUES

Below are the few issues we encountered during the execution of this project in the GLUE machine

- 1) The `setenv` path in the project document has to be modified



A brief tutorial to use llvm is described [here](#). LLVM is already installed on the system, so don't try reinstalling it. Some of the important steps and additional steps are listed below.

First remember to add the llvm and llc install directories to your environment variable PATH using

glue.umd.edu

```
setenv PATH /afs/glue/class/old/enee/759c/llvm/llvm-3.4-install/opt/bin:$PATH
```

Next, copy over the example files in /afs/glue/class/old/enee/759c/llvm/test\_codes/ to a local directory in your home directory. Then the output bytecode file (e.g., example1.bc) can be obtained from source code written in C (e.g., example1.c) using:

```
clang -c -emit-llvm example1.c -o example1.bc
```

Bytecode is the Intermediate Representation of the LLVM compiler. Please test your programs on the test codes placed in the test\_codes folder.

- 2) Since there is no Legacy pass manager in the installed LLVM in the GLUE machine. The iteration of instructions is performed by converting the iterator into instruction with the following command.

```
for(Module::iterator b = M.begin(); b != M.end(); ++b){
    for(Function::iterator f = b->begin(); f != b->end(); ++f){
        for(BasicBlock::iterator i = f->begin(); i != f->end(); ++i){
            Instruction *I = &*i;
```

The above iteration can be easily implemented in the latest llvm using **auto**.

- 3) The relevant doxygen files and class descriptions on the internet have been altered in accordance with LLVM16 and its numerous methods and constructors changed.

According to the doxygen file of `cloneFunction()` the description states that generated clone will be added to that function's module, LLVM3.4 does not add the clone function into the module instead we are required to add the function into the module manually. Since we are assuming obtaining the constructors according to the Doxygen files they are providing some misconceptions while implementing the program in the GLUE machine.

#### ◆ CloneFunction()

```
Function * llvm::CloneFunction ( Function *      F,  
                                ValueToValueMapTy & VMap,  
                                ClonedCodeInfo *   CodeInfo = nullptr  
                                )
```

Return a copy of the specified function and add it to that function's module.

Manual addition in LLVM 3.4 is done by following the below approach.

```
llvm::ValueToValueMapTy VMap;  
Function *clone = CloneFunction(fun,VMap,false);  
fun->getParent()->getFunctionList().push_back(clone);
```

## CHAPTER-6 CONCLUSION

The function cloning pass has been implemented for the appropriate target processors which do not have branch prediction hardware. The function cloning pass optimizes the execution of a program that has a return value, starting with the letter “p”.