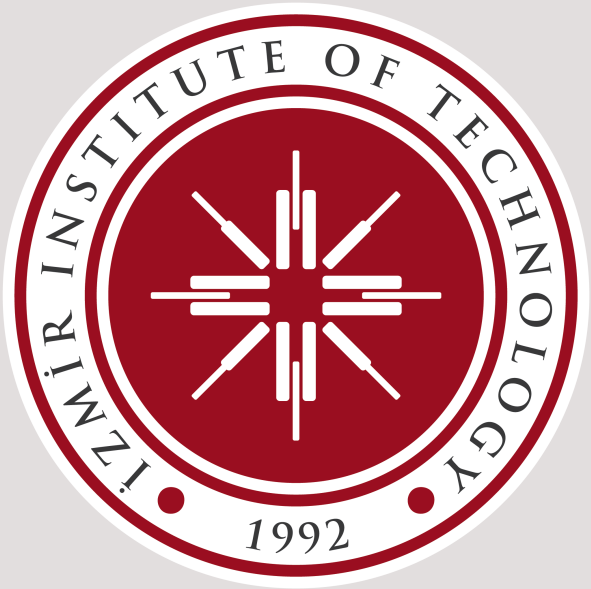
Izmir Institute of Technology
Computer Engineering Department
CENG513 Programming Assignment 3

Student Name: Gökay Gülsoy Student No: 270201072

May 18, 2024



List of Code

1 Implementation of MulToAdd.cpp 3
2 Updated CMakeLists.txt file located under llvm-tutor/lib 6
3 MulToAdd.h Header File 8
4 Initial IR File 9
5 Optimized IR File 12

1 LLVM Pass Implementation

In order to implement my own LLVM pass which replaces all integer multiplications by 2 with addition by itself, I have followed the steps in order which are indicated as given below [1] :

1. Adding new **MulToAdd.cpp** file under llvm-tutor/lib
2. Updating **CMakeLists.txt** file under the llvm-tutor/lib
3. Adding **MulToAdd.h** file under llvm-tutor/include
4. export LLVM_DIR=/home/gokay/llvm-project/build
5. $LLVM_DIR/bin/clang -emit-llvm -S llvm-tutor/inputs/input_for_mul_to_add.c -o input_for_mul_to_add.ll
6. $LLVM_DIR/bin/opt -load-pass-plugin=build/lib/libMultToAdd.so -passes="mul-to-add" -S input_for_mult_to_add.ll -o optimized_mul_to_add.ll

1.1 Adding MulToAdd.cpp File

First step is to write an actual LLVM pass code which is going to traverse all instructions in basic block. I added this file under **llvm-tutor/lib** directory [2]. Implementation of MultToAdd.cpp is given as follows:

```
#include "MulToAdd.h"

#include "llvm/ADT/Statistic.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/Value.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"
#include "llvm/Transforms/Utils/BasicBlockUtils.h"

#include <random>

using namespace llvm;

#define DEBUG_TYPE "mul-to-add"
STATISTIC(SubstCount, "The # of substituted instructions");

//-----------------------------------------------------------------
// MulToAdd implementation
//-----------------------------------------------------------------
```

```
20 bool MulToAdd::runOnBasicBlock(BasicBlock &BB) {
21     bool Changed = false;
22
23     // iterating over all instructions in basic block
24     for (auto Inst = BB.begin(), IE = BB.end(); Inst != IE; Inst++) {
25
26         // we skip non-binary instructions
27         auto *BinOp = dyn_cast<BinaryOperator>(Inst);
28         if (!BinOp) {
29             continue;
30         }
31
32         // skip instructions other than mul
33         unsigned OpCode = BinOp->getOpcode();
34         if (OpCode != Instruction::Mul || !BinOp->getType()->isIntegerTy())
               {
35
36             continue;
37         }
38
39         auto *Op0 = BinOp->getOperand(0);
40         auto *Op1 = BinOp->getOperand(1);
41
42         ConstantInt *CI;
43         Value *OperandToAddByItself;
44         // determining which operand is the constant integer 2
45         if ((CI = dyn_cast<ConstantInt>(Op0)) && CI->equalsInt(2)) {
46             OperandToAddByItself = Op1;
47         }
48
49         else if ((CI = dyn_cast<ConstantInt>(Op1)) && CI->equalsInt(2)) {
50             OperandToAddByItself = Op0;
51         }
52
53         else  {
54             continue;
55         }
56
57         // Creating new multiplication instruction
58         // and replacing it with previous instruction in basic block
```

```
59          IRBuilder<> Builder(BinOp);
60          Instruction *NewInst = BinaryOperator::CreateAdd(
61                                  OperandToAddByItself,
62                                  OperandToAddByItself);
63
64          ReplaceInstWithInst(&BB,Inst,NewInst);
65          Changed = true;
66
67          // updating the statistics
68          SubstCount++;
69      }
70
71      return Changed;
72  }
73
74  PreservedAnalyses MulToAdd::run(llvm::Function &F,
75                                  llvm::FunctionAnalysisManager &) {
76
77          bool Changed = false;
78
79          for (auto &BB : F) {
80              Changed |= runOnBasicBlock(BB);
81          }
82
83          return (Changed ? llvm::PreservedAnalyses::none() :
84                            llvm::PreservedAnalyses::all());
85  }
86
87  //-----------------------------------------------------------
88  // Registering our Pass
89  //-----------------------------------------------------------
90  llvm::PassPluginLibraryInfo getMulToAddPluginInfo() {
91    return {LLVM_PLUGIN_API_VERSION, "mul-to-add", LLVM_VERSION_STRING,
92            [](PassBuilder &PB) {
93              PB.registerPipelineParsingCallback(
94                  [](StringRef Name, FunctionPassManager &FPM,
95                     ArrayRef<PassBuilder::PipelineElement>) {
96                    if (Name == "mul-to-add") {
97                      FPM.addPass(MulToAdd());
98                      return true;
```

```
                    }
                    return false;
                  });
            }};
}

extern "C" LLVM_ATTRIBUTE_WEAK ::llvm::PassPluginLibraryInfo
llvmGetPassPluginInfo() {
  return getMulToAddPluginInfo();
}
```

Code 1: Implementation of MulToAdd.cpp

Implementation for replacing integer multiplication by 2 with addition by itself mainly consists of two parts. First part is the traversal of instructions inside the basic block and replacing desired instructions with new instruction. Second part is the registration of the pass. I give the name "mul-to-add" for my pass, it should be used with -passes option whenever pass is to applied over any .ll file

1.2 Updating CMakeLists.txt Under llvm-tutor/lib Directory

I had to update the CMakeLists.txt file located under the llvm-tutor/lib directory so that my new pass is build as a plugin when I make the project. I have updated two parts in the CMakeLists.txt first is adding MulToAdd under LLVM_TUTOR_PLUGINS and adding **set(MulToAdd_SOURCES MulToAdd.cpp)** line below other set commands. Updated CMakeLists.txt file is as follows:

```
# THE LIST OF PLUGINS AND THE CORRESPONDING SOURCE FILES
# =====================================================
set(LLVM_TUTOR_PLUGINS
    StaticCallCounter
    DynamicCallCounter
    FindFCmpEq
    ConvertFCmpEq
    InjectFuncCall
    MBAAdd
    MBASub
    RIV
    DuplicateBB
    OpcodeCounter
    MergeBB
    MulToAdd
    )
```

```
17
18 set(StaticCallCounter_SOURCES
19   StaticCallCounter.cpp)
20 set(DynamicCallCounter_SOURCES
21   DynamicCallCounter.cpp)
22 set(FindFCmpEq_SOURCES
23   FindFCmpEq.cpp)
24 set(ConvertFCmpEq_SOURCES
25   ConvertFCmpEq.cpp)
26 set(InjectFuncCall_SOURCES
27   InjectFuncCall.cpp)
28 set(MBAAdd_SOURCES
29   MBAAdd.cpp)
30 set(MBASub_SOURCES
31   MBASub.cpp)
32 set(RIV_SOURCES
33   RIV.cpp)
34 set(DuplicateBB_SOURCES
35   DuplicateBB.cpp)
36 set(OpcodeCounter_SOURCES
37   OpcodeCounter.cpp)
38 set(MergeBB_SOURCES
39   MergeBB.cpp)
40 set(MulToAdd_SOURCES
41    MulToAdd.cpp)
42
43 # CONFIGURE THE PLUGIN LIBRARIES
44 # ==============================
45 foreach( plugin ${LLVM_TUTOR_PLUGINS} )
46     # Create a library corresponding to 'plugin'
47     add_library(
48       ${plugin}
49       SHARED
50       ${${plugin}_SOURCES}
51       )
52
53     # Configure include directories for 'plugin'
54     target_include_directories(
55       ${plugin}
56       PRIVATE
```

```
      "${CMAKE_CURRENT_SOURCE_DIR}/../include"
    )

    # On Darwin (unlike on Linux), undefined symbols in shared objects are not
    # allowed at the end of the link-edit. The plugins defined here:
    #  - _are_ shared objects
    #  - reference symbols from LLVM shared libraries, i.e. symbols which are
    #    undefined until those shared objects are loaded in memory (and hence
    #    _undefined_ during static linking)
    # The build will fail with errors like this:
    #    "Undefined symbols for architecture x86_64"
    # with various LLVM symbols being undefined. Since those symbols are later
    # loaded and resolved at runtime, these errors are false positives.
    # This behaviour can be modified via the '-undefined' OS X linker flag as
    # follows.
    target_link_libraries(
      ${plugin}
      "$<$<PLATFORM_ID:Darwin>:-undefined dynamic_lookup>"
      )
endforeach()
```

Code 2: Updated CMakeLists.txt file located under llvm-tutor/lib

1.3 Adding MulToAdd.h file Under llvm-tutor/include Directory

I have added **MulToAdd.h** file under llvm-tutor/include directory. MulToAdd.h file contains MulToAdd struct along with required methods to run pass over the basic block. MulToAdd.h file is as follows:

```
//========================================================================
// FILE:
//    MulToAdd.h
//
// DESCRIPTION:
//    Declares the MulToAdd pass for pass manager.
//========================================================================
#ifndef LLVM_TUTOR_MUL_TO_ADD_H
```

```
9  #define LLVM_TUTOR_MUL_TO_ADD_H
10
11 #include "llvm/IR/PassManager.h"
12 #include "llvm/Pass.h"
13
14 struct MulToAdd : public llvm::PassInfoMixin<MulToAdd> {
15   llvm::PreservedAnalyses run(llvm::Function &F,
16                               llvm::FunctionAnalysisManager &);
17   bool runOnBasicBlock(llvm::BasicBlock &B);
18
19   // Without isRequired returning true, this pass will be skipped for
         functions
20   // decorated with the optnone LLVM attribute. Note that clang -O0
         decorates
21   // all functions with optnone.
22   static bool isRequired() { return true; }
23 };
24 #endif
```

Code 3: MulToAdd.h Header File

run function takes reference to Function type and reference to FunctionAnalysisManager as it's parameters. runOnBasicBlock function takes a reference to BasicBlock over which pass will be applied. isRequired method should return true, otherwise pass will be skipped for functions decorated with optnone LLVM attribute. By default when source code is compiled with clang -O0, all functions are decorated with optnone [3].

1.4 Generating IR file and Running Pass over the Input File

As llvm-tutor is an out-of-tree project, I had to specify path for my LLVM installation directory with the command **export LLVM_DIR=/home/gokay/llvm-project/build** so that LLVM_DIR environment variable holds the LLVM installation directory for the current shell session. Then I have generated IR for the input file with the command **$LLVM_DIR/bin/clang -emit-llvm -S llvm-tutor/inputs/input_for_mul_to_add.c -o input_for_mul_to_add.ll**. Finally, I run the pass over the IR with the command **$LLVM_DIR/bin/opt -load-pass-plugin=build/lib/libMulToAdd.so -passes="mul-to-add" -S input_for_mult_to_add.ll -o optimized_mul_to_add.ll** to generate the optimized IR file. As a result of running the pass, I got the optimized version of the IR file which is named as **optimized_mul_to_add.ll**. initial and optimized IR files are as follows:

```
1 ; ModuleID = '../inputs/input_for_mul_to_add.c'
```

```
source_filename = "../inputs/input_for_mul_to_add.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128
    :128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main(i32 noundef %0, ptr noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca ptr, align 8
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  %8 = alloca i32, align 4
  %9 = alloca i32, align 4
  %10 = alloca i32, align 4
  %11 = alloca i32, align 4
  %12 = alloca i32, align 4
  %13 = alloca i32, align 4
  store i32 0, ptr %3, align 4
  store i32 %0, ptr %4, align 4
  store ptr %1, ptr %5, align 8
  %14 = load ptr, ptr %5, align 8
  %15 = getelementptr inbounds ptr, ptr %14, i64 1
  %16 = load ptr, ptr %15, align 8
  %17 = call i32 @atoi(ptr noundef %16) #2
  store i32 %17, ptr %6, align 4
  %18 = load ptr, ptr %5, align 8
  %19 = getelementptr inbounds ptr, ptr %18, i64 2
  %20 = load ptr, ptr %19, align 8
  %21 = call i32 @atoi(ptr noundef %20) #2
  store i32 %21, ptr %7, align 4
  %22 = load ptr, ptr %5, align 8
  %23 = getelementptr inbounds ptr, ptr %22, i64 3
  %24 = load ptr, ptr %23, align 8
  %25 = call i32 @atoi(ptr noundef %24) #2
  store i32 %25, ptr %8, align 4
  %26 = load ptr, ptr %5, align 8
  %27 = getelementptr inbounds ptr, ptr %26, i64 4
  %28 = load ptr, ptr %27, align 8
  %29 = call i32 @atoi(ptr noundef %28) #2
```

```
  store i32 %29, ptr %9, align 4
  %30 = load i32, ptr %6, align 4
  %31 = mul nsw i32 2, %30
  store i32 %31, ptr %10, align 4
  %32 = load i32, ptr %7, align 4
  %33 = mul nsw i32 2, %32
  store i32 %33, ptr %11, align 4
  %34 = load i32, ptr %8, align 4
  %35 = mul nsw i32 2, %34
  store i32 %35, ptr %12, align 4
  %36 = load i32, ptr %9, align 4
  %37 = mul nsw i32 2, %36
  store i32 %37, ptr %13, align 4
  %38 = load i32, ptr %10, align 4
  %39 = load i32, ptr %11, align 4
  %40 = add nsw i32 %38, %39
  %41 = load i32, ptr %12, align 4
  %42 = add nsw i32 %40, %41
  %43 = load i32, ptr %13, align 4
  %44 = add nsw i32 %42, %43
  ret i32 %44
}

; Function Attrs: nounwind willreturn memory(read)
declare i32 @atoi(ptr noundef) #1

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "
    min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-
    buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+
    fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { nounwind willreturn memory(read) "frame-pointer"="all" "
    no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"=
    "x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune
    -cpu"="generic" }
attributes #2 = { nounwind willreturn memory(read) }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
```

```
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"clang version 19.0.0git (https://github.com/llvm/llvm-project.git
    790bcecce6c135476d2551805c09ed670b9f8418)"}
```

Code 4: Initial IR File

```
; ModuleID = 'input_for_mul_to_add.ll'
source_filename = "../inputs/input_for_mul_to_add.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128
    :128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main(i32 noundef %0, ptr noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca ptr, align 8
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  %8 = alloca i32, align 4
  %9 = alloca i32, align 4
  %10 = alloca i32, align 4
  %11 = alloca i32, align 4
  %12 = alloca i32, align 4
  %13 = alloca i32, align 4
  store i32 0, ptr %3, align 4
  store i32 %0, ptr %4, align 4
  store ptr %1, ptr %5, align 8
  %14 = load ptr, ptr %5, align 8
  %15 = getelementptr inbounds ptr, ptr %14, i64 1
  %16 = load ptr, ptr %15, align 8
  %17 = call i32 @atoi(ptr noundef %16) #2
  store i32 %17, ptr %6, align 4
  %18 = load ptr, ptr %5, align 8
  %19 = getelementptr inbounds ptr, ptr %18, i64 2
  %20 = load ptr, ptr %19, align 8
  %21 = call i32 @atoi(ptr noundef %20) #2
  store i32 %21, ptr %7, align 4
```

```
32    %22 = load ptr, ptr %5, align 8
33    %23 = getelementptr inbounds ptr, ptr %22, i64 3
34    %24 = load ptr, ptr %23, align 8
35    %25 = call i32 @atoi(ptr noundef %24) #2
36    store i32 %25, ptr %8, align 4
37    %26 = load ptr, ptr %5, align 8
38    %27 = getelementptr inbounds ptr, ptr %26, i64 4
39    %28 = load ptr, ptr %27, align 8
40    %29 = call i32 @atoi(ptr noundef %28) #2
41    store i32 %29, ptr %9, align 4
42    %30 = load i32, ptr %6, align 4
43    %31 = add i32 %30, %30
44    store i32 %31, ptr %10, align 4
45    %32 = load i32, ptr %7, align 4
46    %33 = add i32 %32, %32
47    store i32 %33, ptr %11, align 4
48    %34 = load i32, ptr %8, align 4
49    %35 = add i32 %34, %34
50    store i32 %35, ptr %12, align 4
51    %36 = load i32, ptr %9, align 4
52    %37 = add i32 %36, %36
53    store i32 %37, ptr %13, align 4
54    %38 = load i32, ptr %10, align 4
55    %39 = load i32, ptr %11, align 4
56    %40 = add nsw i32 %38, %39
57    %41 = load i32, ptr %12, align 4
58    %42 = add nsw i32 %40, %41
59    %43 = load i32, ptr %13, align 4
60    %44 = add nsw i32 %42, %43
61    ret i32 %44
62  }
63
64  ; Function Attrs: nounwind willreturn memory(read)
65  declare i32 @atoi(ptr noundef) #1
66
67  attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "
        min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-
        buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+
        fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
68  attributes #1 = { nounwind willreturn memory(read) "frame-pointer"="all" "
```

```
    no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"=
    "x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune
    -cpu"="generic" }
attributes #2 = { nounwind willreturn memory(read) }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"clang version 19.0.0git (https://github.com/llvm/llvm-project.git
    790bcecce6c135476d2551805c09ed670b9f8418)"}
```

Code 5: Optimized IR File

If we look at the initial and optimized IR files, specifically at lines **42-52** we can see that in initial IR file, mul instructions are used to multiply by two, whereas in optimized IR file those mul instructions are replaced by add instructions to perform addition of integer by itself.

1.5 Effectiveness of Pass, Potential Limitations and Improvements

In the current version my LLVM pass it can find all integer multiplications inside functions, and replace them with addition by itself. However it is restricted to functions, meaning that it can only work with integer variables inside functions not with variables defined in global scope, also it is restricted to integer type it can not perform replacement for other numeric data types (float, double, etc.) for now. Improvements that can be done to extend the functionality of the pass is to adding support for other numeric data types, and performing replacement even when variables are not defined inside functions (in global scope).

References

[1] *GitHub - banach-space/llvm-tutor: A collection of out-of-tree LLVM passes for teaching and learning — github.com*. `https://github.com/banach-space/llvm-tutor`. [Accessed 18-05-2024].

[2] *2019 LLVM Developers' Meeting: A. Warzynski "Writing an LLVM Pass: 101" — youtube.com*. `https://www.youtube.com/watch?v=ar7cJl2aBuU&t=1223s`. [Accessed 18-05-2024].

[3] *Writing an LLVM Pass &x2014; LLVM 19.0.0git documentation — llvm.org*. `https://llvm.org/docs/WritingAnLLVMNewPMPass.html`. [Accessed 18-05-2024].