REPORT ON LOOP DESTRUCTION AND CONTROL FLOW DESTRUCTION

- IN LLVM IR -

Project Report By

Thumati Ujjieve (CS16BTECH11039)

IIT HYDERABAD
Computer Science Engineering

LaTeX

Contents

1 How Loop are represented in LLVM IR **3**
1.1 Example 3

2 What are non natural loops **5**

3 Obfuscation **7**
3.1 What is Obfuscation 7
3.2 Why Obfuscation 7
3.3 How LoopFlattening and BranchFlattening can be considered as Obfuscation Pass ?. 7

4 Loop Destruction - LLVM Transformation Pass to Obfuscate Loops **9**
4.1 Which Pass is implemented and Why ? 9
4.2 What are the preRequired Passes? 10
4.3 What is the Transformation? 10
4.3.1 Addition of Swtich Block 10
4.3.2 Addition of Pre Blocks 10
4.3.3 Example of the LD Transformation 10
4.4 What are valid inputs to Our Pass? 13
4.5 The issue of Adding PHI Node 13
4.6 The Issue of Pre-existing PHI Node 13
4.7 The issue of "Instruction doesn′t dominate all uses" 14
4.8 Case where only Header Block is present 15

5 Control Flow Destruction - LLVM Transformation Pass to Obfuscate If-Else Branches **21**
5.1 How branching statements are handled in LLVM IR 21
5.2 commands to run Pass: 21
5.3 Control Flow transformation 22
5.4 cases handled in BranchFlattening 23

5.5 improving Control Flow Obfuscation to work on other kind of CFG's 23

6 Testing of the Transformation Passes 27

6.1 Testing of LoopFlattening 28

6.2 Benchmarks: Causing runtime errors 28

6.2.1 Reasons for Runtime failure 29

6.2.2 Idea to tackle this problem 29

6.3 Testing of BranchFlattening 30

7 Bibliography 31

7.1 References 31

Chapter 1

How Loop are represented in LLVM IR

The Loops in LLVM IR consists of Basic Blocks. Only natural loops are identifiable by the LLVM Analysis pass. Natural loops are the loops where there is a backedge from n -> h where h dominates n.

The Information about the Loop is store in LoopInfo Object which is populated by the LoopInfoWrapper pass. The Loop consists of a pre header, a header, multiple blocks of body, a latch, and an end block.

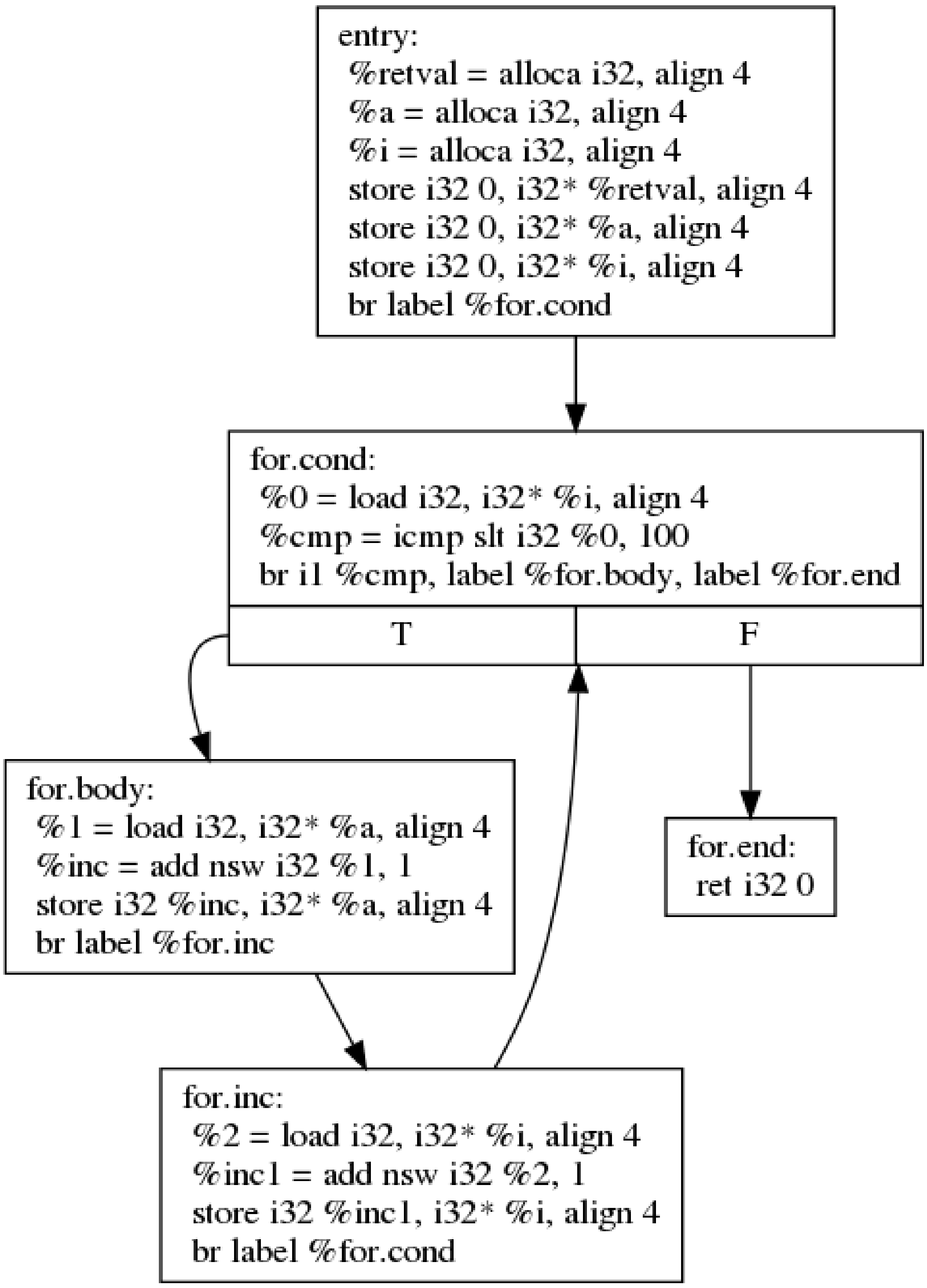
It's possible that anyone of the Latch, preHeader, endblock or body to be missing. But there must always be a header block.

1.1 Example

Example of Loop CFG in LLVM 4.3.

Example 1.1 (An Example to show the Loop CFG)
Simple C Code to show loop structure

```
    int main(){
int a= 0;
for (int i=0;i<100;i++){
a++;
}
return 0;
}
The generated CFG is:
```



```
entry:
 %retval = alloca i32, align 4
 %a = alloca i32, align 4
 %i = alloca i32, align 4
 store i32 0, i32* %retval, align 4
 store i32 0, i32* %a, align 4
 store i32 0, i32* %i, align 4
 br label %for.cond
```

```
for.cond:
 %0 = load i32, i32* %i, align 4
 %cmp = icmp slt i32 %0, 100
 br i1 %cmp, label %for.body, label %for.end
```

```
for.body:
 %1 = load i32, i32* %a, align 4
 %inc = add nsw i32 %1, 1
 store i32 %inc, i32* %a, align 4
 br label %for.inc
```

```
for.end:
 ret i32 0
```

```
for.inc:
 %2 = load i32, i32* %i, align 4
 %inc1 = add nsw i32 %2, 1
 store i32 %inc1, i32* %i, align 4
 br label %for.cond
```

CFG for 'main' function

Chapter 2

What are non natural loops

The Loops in which there is a cyclic graph but there exists no back edge from the block **n** to **h** where **h** dominates **n**. These blocks are not detectable by basic LLVM analysis pass. So The main objective of the project is to convert the Loop into such format .

An irreducible graph is also a non natural loop.

Example 2.1 (An Example to show Irreducible graphs and non natural Loops)

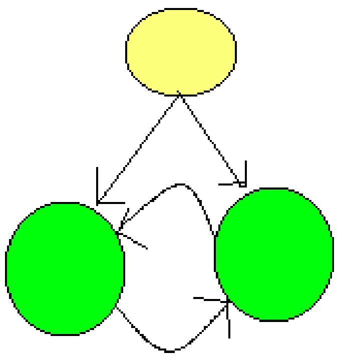


Figure 2.1: structure of irreducible graph

green block's don't Dominate each other , but forms a cycle

Chapter 3

Obfuscation

3.1 What is Obfuscation

Software obfuscation can be defined as a semantics-preserving code transformation of a program in an attempt to make the code as complex and confusing as possible.

.Obfuscation can be done manually or by using an automated tool .
there are different obfuscation techniques each aim for different things.but in general
a) data based Obfuscations :
eg: by reflecting of carry , using indirect addressing , use if register addressing , combining binary instructions etc
b) control flow based Obfuscations :
eg: loop-transformations , destructing branches ..etc

3.2 Why Obfuscation

Programming code is often obfuscated to protect intellectual property such as designs,concepts,trade secrets..as well as work of art and prevent an attacker from reverse engineering .
It will make much more difficult to reverse engineer , it will provide security against unauthorized access , bypassing licensing , vulnerability discovery etc

3.3 How LoopFlattening and BranchFlattening can be considered as Obfuscation Pass ?.

As mentioned above after Obfuscation Pass should not change before and after transformation .

a)**In Loop Destruction :**
we are successful in converting natural loops to irreducible which takes more time and effort to attack and the program after transformation provides same output as before.
we have run **-analyze -loops** before and after transformation and found that natural loops detected before transformation are not detected later . after running our pass we have ran **-O3** to check whether it re-transforms our pass results but it didn't
b)**In Control Flow Destruction:**
earlier the flow is determined by branching statements , after transformation we have changed the flow to be dependent on data .which makes harder to reverse engineer just by looking instructions and not data .

Chapter 4

Loop Destruction - LLVM Transformation Pass to Obfuscate Loops

The transformation pass is used to obfuscate the natural loops by altering the structure in ways that makes the analysis pass of the LLVM to not detect it.

To run the transformation pass in opt use the following command:

Example 4.1 (The command to run the pass in opt)

```
opt -load /build-folder/lib/ControlFlattening.so -flatten -S test.cpp
```

To run the transformation pass in clang use the following command:

Example 4.2 (The command to run the pass in clang)

```
clang -Xclang -load -Xclang /build-folder/lib/ControlFlattening.so -S test.cpp
```

4.1 Which Pass is implemented and Why ?

The transformation pass implements a Function Pass. Loop Pass was not used since for the loop pass the updation of the Dominator tree had to done manually which becomes very cumbersome since the our pass introduces too many new blocks. The function pass on the other hand does the updation automatically.

4.2 What are the preRequired Passes?

Our Pass requires one transformation pass named Loop Simplify pass which helps in bringing the CFG to natural loop structure.

One Analysis pass named Loop Info Wrapper pass to obtain the inforamtion about the loops present in the Module

4.3 What is the Transformation?

The main Idea is to transform the CFG such that there exists a subgraph which is irreducible and maintaining the semantic of the code.

4.3.1 Addition of Swtich Block

A new Block which contains Switch-case statements was introduced after the Pre-Header Block . The branch instruction in the Pre Header Block was modified to jump to this newly created Block.

This block contains instructions which generate a random number which is feeded in the switch-case instruction.
This ensures that the random number is generated during runtime.

4.3.2 Addition of Pre Blocks

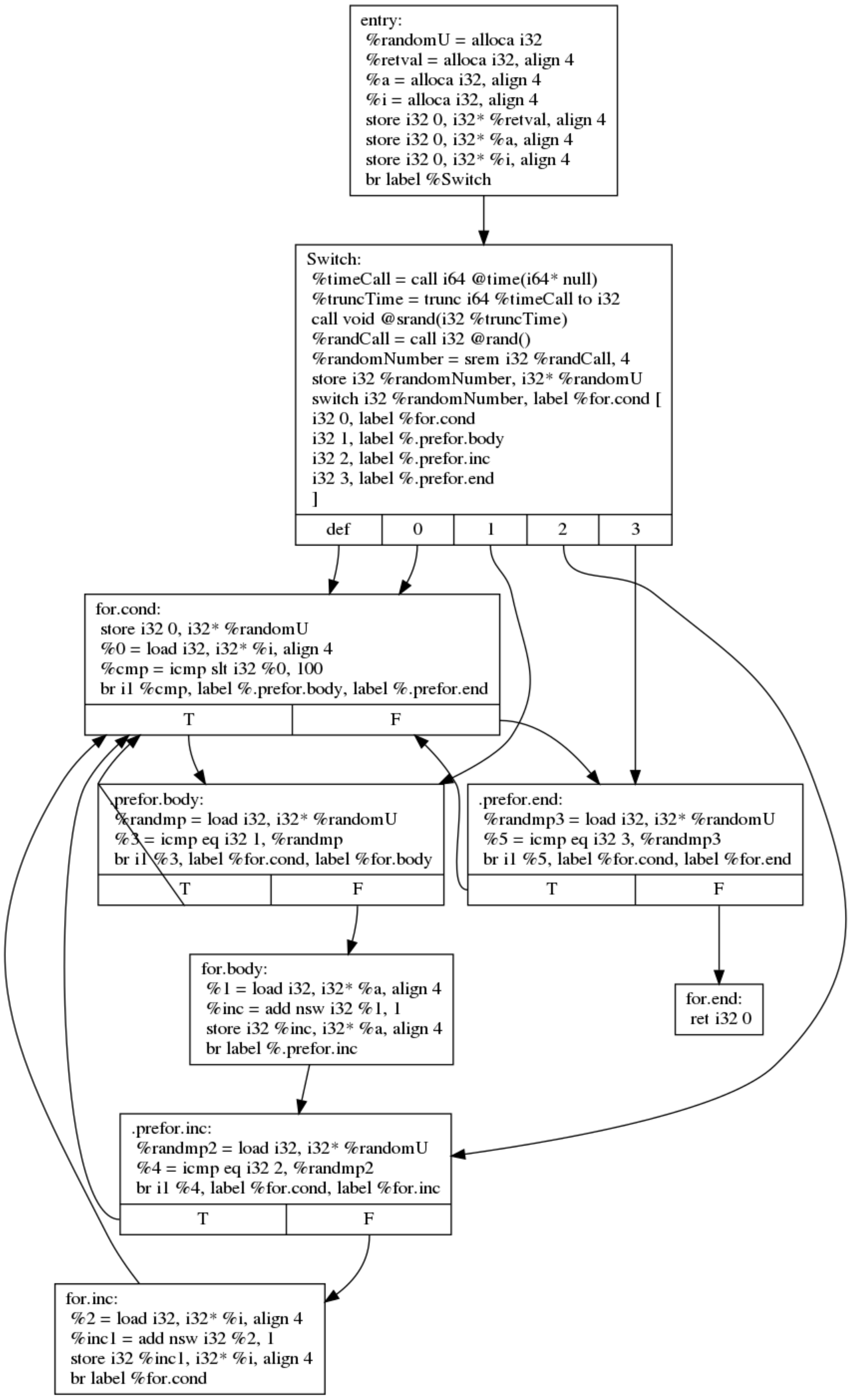
Not more than 3 Pre Blocks were created by the pass. Each one is associated with one of Latch, end block and body. If any one of these three block is missing then no pre block is created for it.

These Preblocks are added before the Latch, end block and body. The PreBlocks contain instructions which determine to which block to jump next. If the pre block was reached from the newly created switch block then it jumps to the header block otherwise it jumps to the associated block which could be either Latch, end block or body.

4.3.3 Example of the LD Transformation

Example 4.3 (The following C code for which the transformation is done)

```
int main(){
    int a=0;
    for (int i = 0 ; i < N ; i++){
        a++;
    }
    return 0;
}
```



```
entry:
 %randomU = alloca i32
 %retval = alloca i32, align 4
 %a = alloca i32, align 4
 %i = alloca i32, align 4
 store i32 0, i32* %retval, align 4
 store i32 0, i32* %a, align 4
 store i32 0, i32* %i, align 4
 br label %Switch
```

```
Switch:
 %timeCall = call i64 @time(i64* null)
 %truncTime = trunc i64 %timeCall to i32
 call void @srand(i32 %truncTime)
 %randCall = call i32 @rand()
 %randomNumber = srem i32 %randCall, 4
 store i32 %randomNumber, i32* %randomU
 switch i32 %randomNumber, label %for.cond [
 i32 0, label %for.cond
 i32 1, label %.prefor.body
 i32 2, label %.prefor.inc
 i32 3, label %.prefor.end
 ]
```

```
for.cond:
 store i32 0, i32* %randomU
 %0 = load i32, i32* %i, align 4
 %cmp = icmp slt i32 %0, 100
 br i1 %cmp, label %.prefor.body, label %.prefor.end
```

```
.prefor.body:
 %randmp = load i32, i32* %randomU
 %3 = icmp eq i32 1, %randmp
 br i1 %3, label %for.cond, label %for.body
```

```
.prefor.end:
 %randmp3 = load i32, i32* %randomU
 %5 = icmp eq i32 3, %randmp3
 br i1 %5, label %for.cond, label %for.end
```

```
for.body:
 %1 = load i32, i32* %a, align 4
 %inc = add nsw i32 %1, 1
 store i32 %inc, i32* %a, align 4
 br label %.prefor.inc
```

```
for.end:
 ret i32 0
```

```
.prefor.inc:
 %randmp2 = load i32, i32* %randomU
 %4 = icmp eq i32 2, %randmp2
 br i1 %4, label %for.cond, label %for.inc
```

```
for.inc:
 %2 = load i32, i32* %i, align 4
 %inc1 = add nsw i32 %2, 1
 store i32 %inc1, i32* %i, align 4
 br label %for.cond
```

CFG for 'main' function

One thing to notice is the presence of def in the switch statement. The def will point to a randomly chosen block. This is computed during compile time.

4.4 What are valid inputs to Our Pass?

Any LLVM IR containing a Natural Loop Structure is a valid input. The Loop Structure may or may not contain any of the Latch , Body or End Block.

The Pass is capable of handling cases with complex control flow involving presence of non induction phi nodes in Latch, Body or End Block. The handling of these cases is shown in the following sections.

4.5 The issue of Adding PHI Node

Lot of problems were faced when using PHI Nodes to determine from which block the preblock was reached. One of the issues was related to the transformation passes that ran during lnt-tests after our transformation was scheduled. These passes manipulated the PHI Node by adding or removing certain value,block pairs.

This issue was resolved by using load, store instructions instead. The alloca instructions storing the random number which decided where to jump to from the switch block was inserted in the entry block of the function. This alloca instruction is named as randomAlloca in the source code.

In each of the preBlocks the load instrution were inserted to which was used in the compare operation.

In the header block store instruction was inserted where it stores the number 0 in the randomAlloca location. This was done to reset the value to remove the infinite looping between header block and preblock.

4.6 The Issue of Pre-existing PHI Node

This was one of the major issues faced during handling complex control flow.
The issue was pertaining to the existance of a phi node in Body, Latch or endBlock. Even though mem2reg pass was not run during benchmark testing.

The issue was handled by creating a new PHI Node of type same as the original one in the newly created pre Block and then populating the PHI Node with the value to block pairs. And then the uses of the old phi node was replaced by the new one and then the old phi node was removed using eraseFromParent() function.

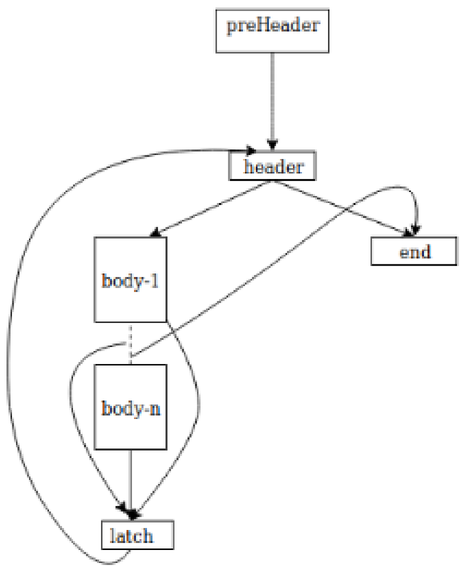


Figure 4.1: CFG before transformation

Care was taken that an extra pair of the required type was added that corresponds to the edge coming from the switch block.

4.7 The issue of "Instruction doesn't dominate all uses"

This issue was encountered in some specific benchmarks like 7zip, Misc-C++. The issue aroused because of the insertion of pre blocks

1. Between the Header Block and Body.
2. Between the Latch and Body.
3. Between the Header Block and end Block

Since by adding a these pre blocks caused a change in the dominators of the Latch, Body and end Block. The new dominator for these 3 blocks now became the newly created Switch Block. This caused the Instructions that were defined in Body, HeaderBlock to loose their dominance over Latch and (Body,endBlock) respectively.

This issue was fixed by adding a PHI Node in the preBlocks. These PHI Nodes

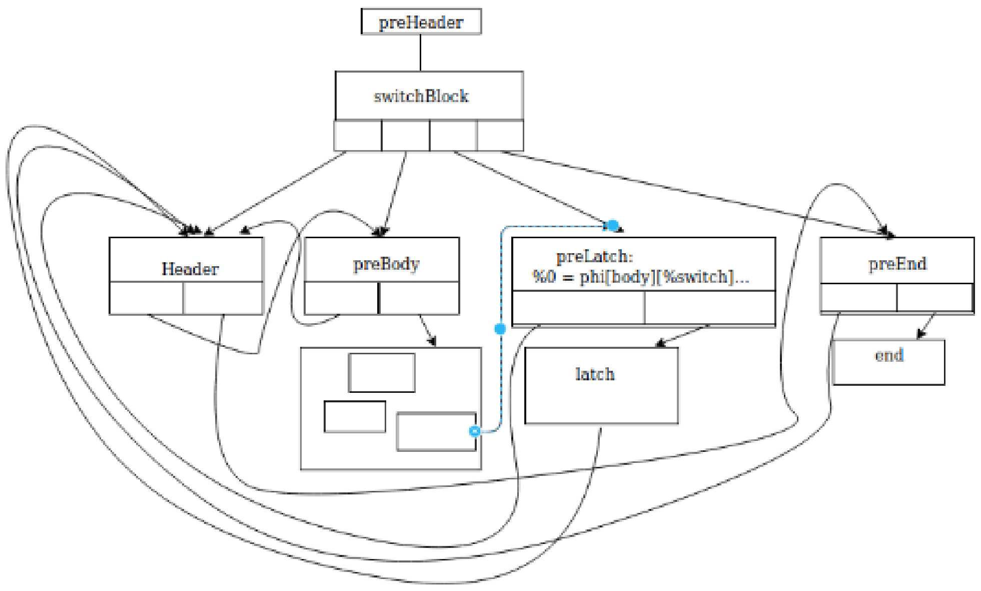


Figure 4.2: CFG after transformation

contained the value block pair for which there was dominance issues, for the rest of the incoming edges an default value was choosed.

4.8 Case where only Header Block is present

In this case the LoopInfo Object of LLVM was detecting the Header Block as Latch also. This caused wrong transformations in the begining . The handling of this case involved creating a new Basic Block similar to the header block.
This issue was found in Misc-C++ single source benchmark of lnt test suite.

The instructions were cloned and new block was populated from these new cloned instructions.Then a switch block was also creating and inserted between the pre header and header block. The graph was made similar to the one showed in figure 2.1 . This ensured that the loop was made undetectable. The transformation is showed in figure 4.5 and 4.6.

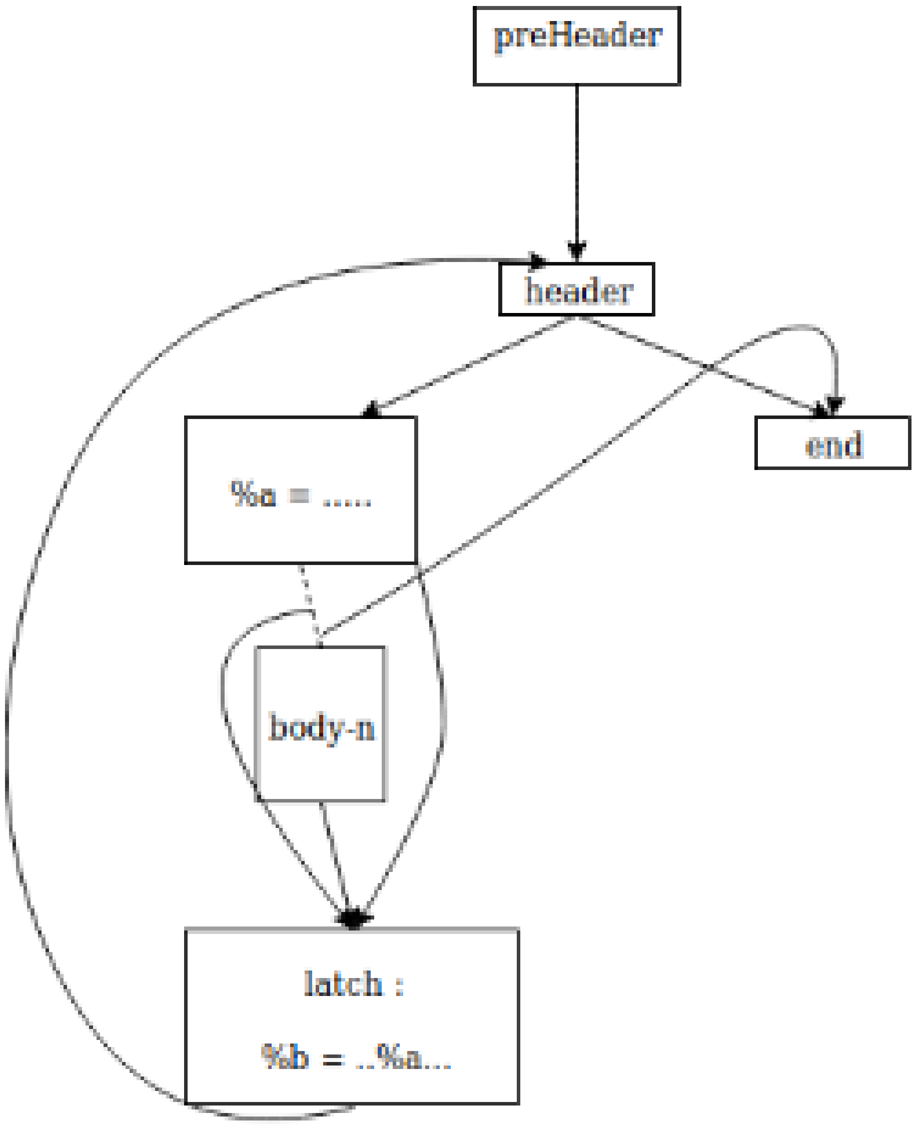


Figure 4.3: CFG before transformation

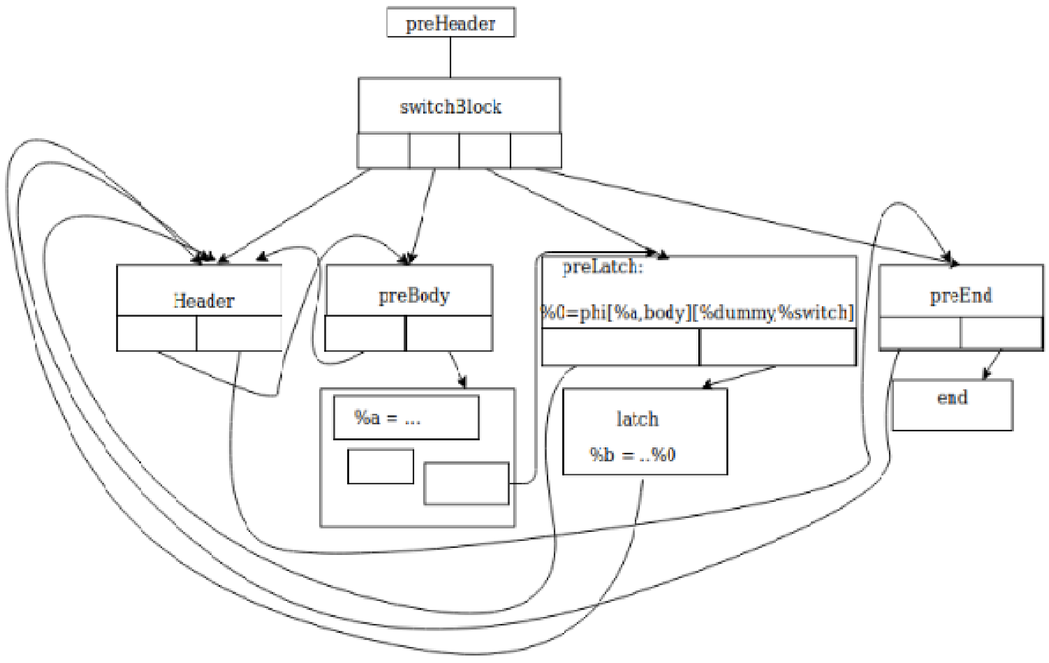
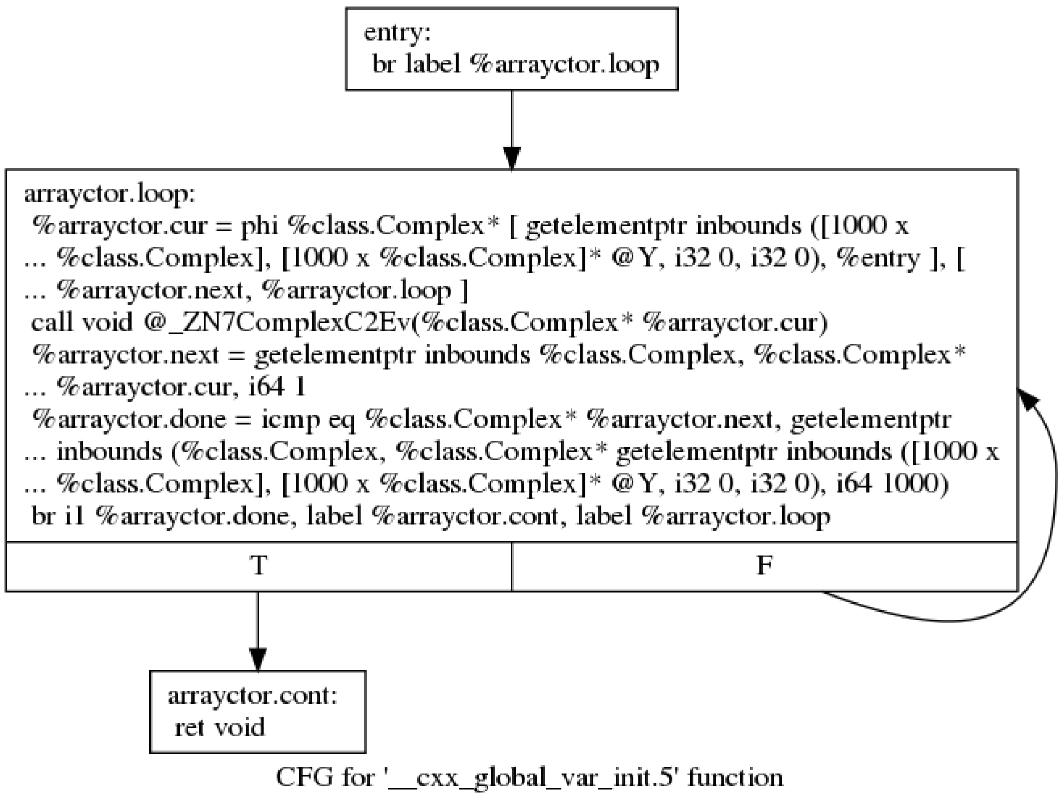


Figure 4.4: CFG after transformation

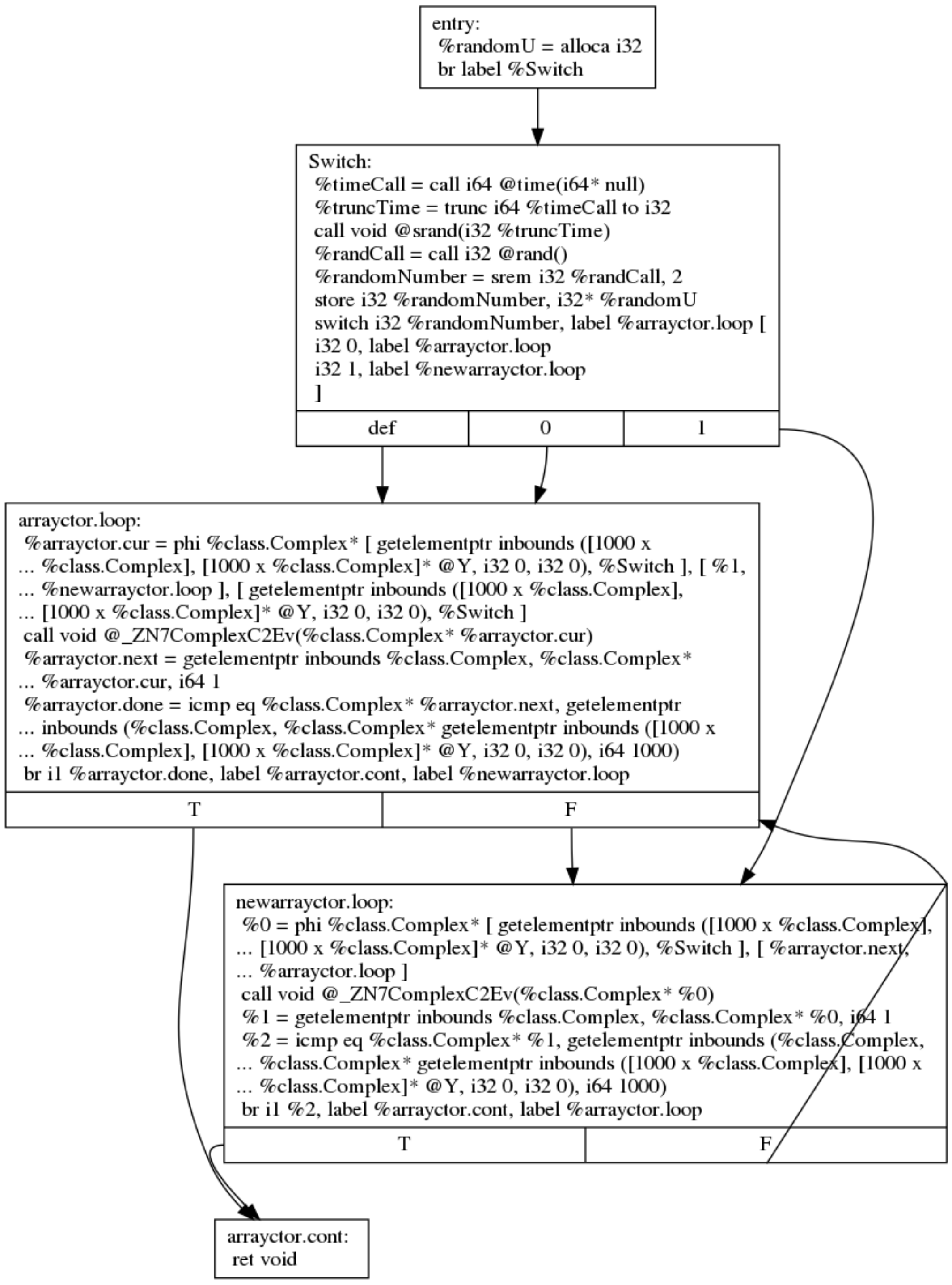


```
entry:
 br label %arrayctor.loop
```

```
arrayctor.loop:
 %arrayctor.cur = phi %class.Complex* [ getelementptr inbounds ([1000 x
... %class.Complex], [1000 x %class.Complex]* @Y, i32 0, i32 0), %entry ], [
... %arrayctor.next, %arrayctor.loop ]
 call void @_ZN7ComplexC2Ev(%class.Complex* %arrayctor.cur)
 %arrayctor.next = getelementptr inbounds %class.Complex, %class.Complex*
... %arrayctor.cur, i64 1
 %arrayctor.done = icmp eq %class.Complex* %arrayctor.next, getelementptr
... inbounds (%class.Complex, %class.Complex* getelementptr inbounds ([1000 x
... %class.Complex], [1000 x %class.Complex]* @Y, i32 0, i32 0), i64 1000)
 br i1 %arrayctor.done, label %arrayctor.cont, label %arrayctor.loop
```

```
arrayctor.cont:
 ret void
```

Figure 4.5: CFG showing of the case where only one header block present



```
entry:
 %randomU = alloca i32
 br label %Switch
```

```
Switch:
 %timeCall = call i64 @time(i64* null)
 %truncTime = trunc i64 %timeCall to i32
 call void @srand(i32 %truncTime)
 %randCall = call i32 @rand()
 %randomNumber = srem i32 %randCall, 2
 store i32 %randomNumber, i32* %randomU
 switch i32 %randomNumber, label %arrayctor.loop [
 i32 0, label %arrayctor.loop
 i32 1, label %newarrayctor.loop
 ]
```

```
arrayctor.loop:
 %arrayctor.cur = phi %class.Complex* [ getelementptr inbounds ([1000 x
... %class.Complex], [1000 x %class.Complex]* @Y, i32 0, i32 0), %Switch ], [ %1,
... %newarrayctor.loop ], [ getelementptr inbounds ([1000 x %class.Complex],
... [1000 x %class.Complex]* @Y, i32 0, i32 0), %Switch ]
 call void @_ZN7ComplexC2Ev(%class.Complex* %arrayctor.cur)
 %arrayctor.next = getelementptr inbounds %class.Complex, %class.Complex*
... %arrayctor.cur, i64 1
 %arrayctor.done = icmp eq %class.Complex* %arrayctor.next, getelementptr
... inbounds (%class.Complex, %class.Complex* getelementptr inbounds ([1000 x
... %class.Complex], [1000 x %class.Complex]* @Y, i32 0, i32 0), i64 1000)
 br i1 %arrayctor.done, label %arrayctor.cont, label %newarrayctor.loop
```

```
newarrayctor.loop:
 %0 = phi %class.Complex* [ getelementptr inbounds ([1000 x %class.Complex],
... [1000 x %class.Complex]* @Y, i32 0, i32 0), %Switch ], [ %arrayctor.next,
... %arrayctor.loop ]
 call void @_ZN7ComplexC2Ev(%class.Complex* %0)
 %1 = getelementptr inbounds %class.Complex, %class.Complex* %0, i64 1
 %2 = icmp eq %class.Complex* %1, getelementptr inbounds (%class.Complex,
... %class.Complex* getelementptr inbounds ([1000 x %class.Complex], [1000 x
... %class.Complex]* @Y, i32 0, i32 0), i64 1000)
 br i1 %2, label %arrayctor.cont, label %arrayctor.loop
```

```
arrayctor.cont:
 ret void
```

CFG for '__cxx_global_var_init.5' function

Figure 4.6: CFG showing the handling of only one header block

Chapter 5

Control Flow Destruction - LLVM Transformation Pass to Obfuscate If-Else Branches

The other important control-flow decision maker are branching statements

```
if(bool_value){
....
}
else{
....
}
```

5.1 How branching statements are handled in LLVM IR

llvm converts if-else to a branching instruction

```
%value␣=␣br␣i1␣%value␣label␣%if_true␣,␣label␣%if_false
```

where ifTrue and ifFalse are labels of Basic Blocks that control flow will go based on the %value

5.2 commands to run Pass:

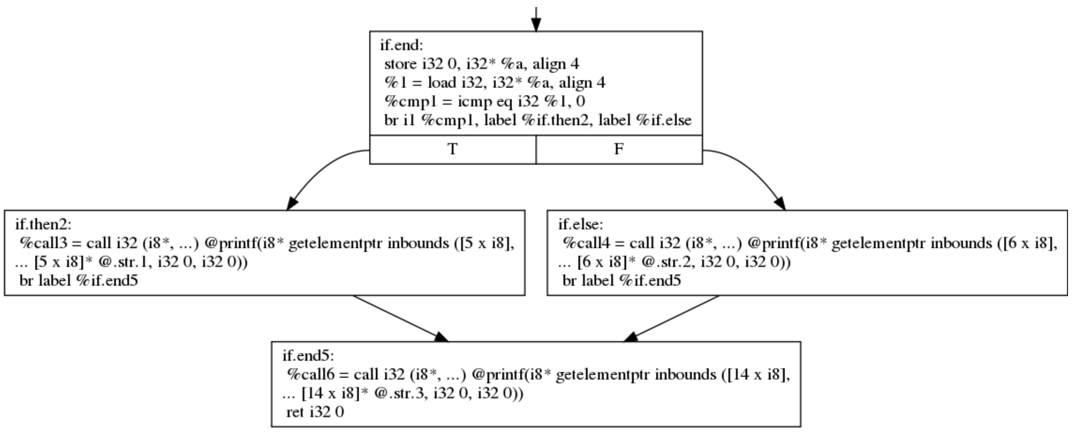
To run the Transformation pass with opt in llvm:

```
opt -load /build-folder/lib/BranchFlattening.so -bflatten -S test.cpp
```

To run the transformation pass in clang use the following command:

```
clang -Xclang -load -Xclang /build-folder/lib/BranchFlattening.so -S test.cpp
```

5.3 Control Flow transformation



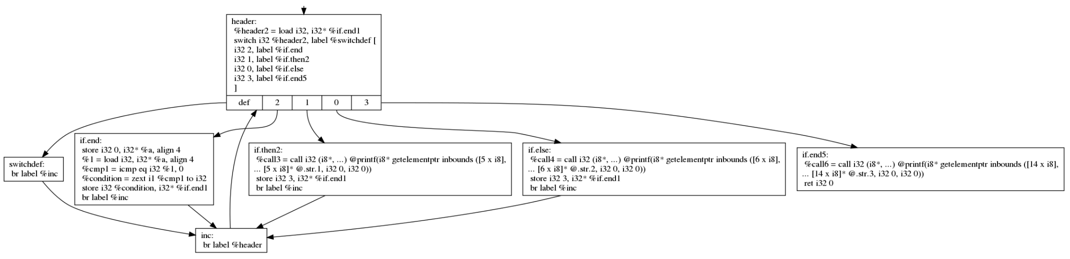
```
if.end:
 store i32 0, i32* %a, align 4
 %1 = load i32, i32* %a, align 4
 %cmp1 = icmp eq i32 %1, 0
 br i1 %cmp1, label %if.then2, label %if.else
```

```
if.then2:
 %call3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8],
... [5 x i8]* @.str.1, i32 0, i32 0))
 br label %if.end5
```

```
if.else:
 %call4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x i8],
... [6 x i8]* @.str.2, i32 0, i32 0))
 br label %if.end5
```

```
if.end5:
 %call6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8],
... [14 x i8]* @.str.3, i32 0, i32 0))
 ret i32 0
```

Figure 5.1: CFG before transformation



```
header:
 %header2 = load i32, i32* %if.end1
 switch i32 %header2, label %switchdef [
 i32 2, label %if.end
 i32 1, label %if.then2
 i32 0, label %if.else
 i32 3, label %if.end5
 ]
```

```
switchdef:
 br label %inc
```

```
if.end:
 store i32 0, i32* %a, align 4
 %1 = load i32, i32* %a, align 4
 %cmp1 = icmp eq i32 %1, 0
 %condition = zext i1 %cmp1 to i32
 store i32 %condition, i32* %if.end1
 br label %inc
```

```
if.then2:
 %call3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8],
... [5 x i8]* @.str.1, i32 0, i32 0))
 store i32 3, i32* %if.end1
 br label %inc
```

```
if.else:
 %call4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x i8],
... [6 x i8]* @.str.2, i32 0, i32 0))
 store i32 3, i32* %if.end1
 br label %inc
```

```
if.end5:
 %call6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8],
... [14 x i8]* @.str.3, i32 0, i32 0))
 ret i32 0
```

```
inc:
 br label %header
```

Figure 5.2: CFG after transformation

we are using a variable for each branching statement and switch instruction that directs to specific basic block based on value of the variable
2 - Block where branching instruction is present
1 - if true block
0 - if false block
3 - if end block
in each this block a new value for variable is stored such that it will go correctly to next block

5.4 cases handled in BranchFlattening

for now we have just handled very basic case i.e when a Basic Block has a conditional branching statement then it should have only one predecessor and the basic blocks that are available through the branching statement i,e **ifTrue** and **ifFlase** block should have unconditional branch to same BasicBlock.
but we can run some preProcessing to make other types of CFG's to transform through our BranchFlattening pass which are given below.

5.5 improving Control Flow Obfuscation to work on other kind of CFG's

a)creating a single Predecessor for BasicBlock that has conditional Branching statement in figure 5.3 and 5.4
b)adding a dummy BasicBlock in figure 5.5 and 5.6
c)joining all the Blocks that contains return value figure 5.7 and 5.8

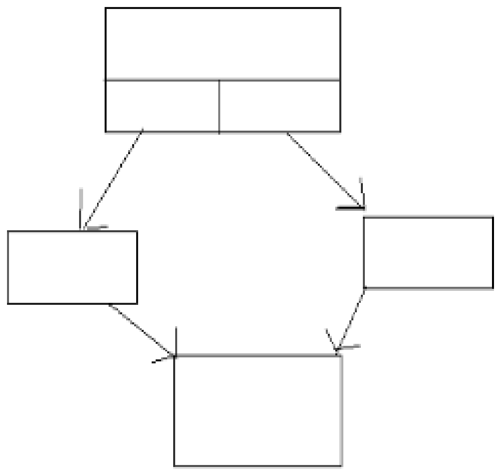


Figure 5.3: CFG befrore preProcessing

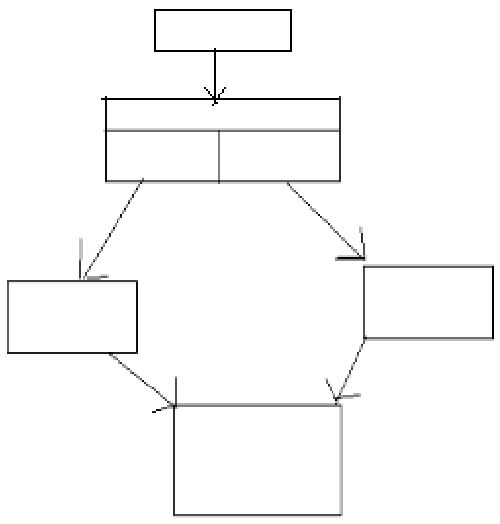


Figure 5.4: CFG after after preProcesing

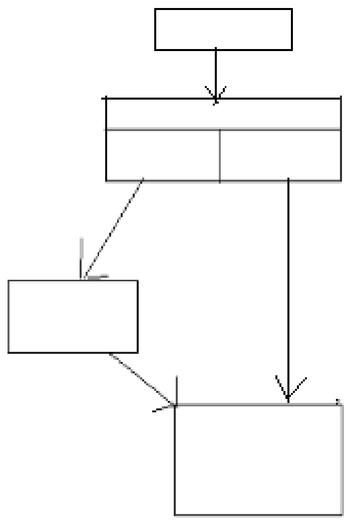


Figure 5.5: CFG befrore preProcessing

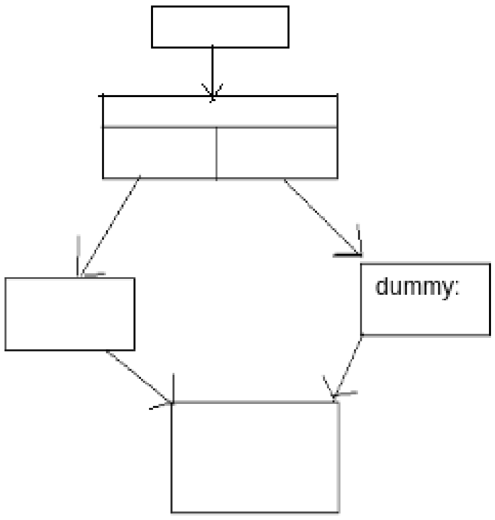


Figure 5.6: CFG after after preProcesing

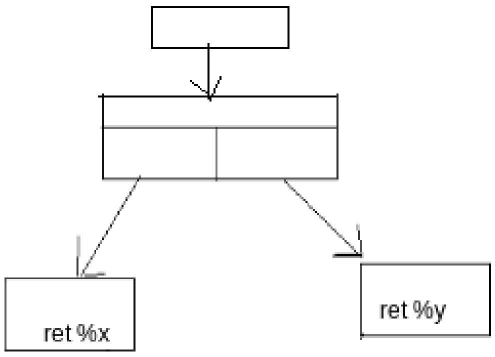


Figure 5.7: CFG befrore preProcessing

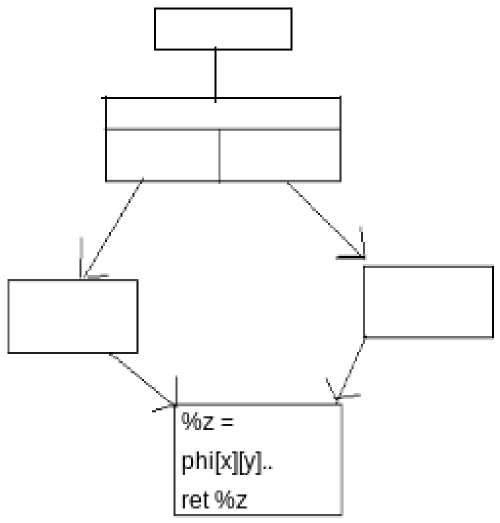


Figure 5.8: CFG after after preProcesing

Chapter 6

Testing of the Transformation Passes

we have used lnt to test Obfuscation passes . lnt is LLVM's performance tracking software , can be used to test . lnt itself provides a test suite nt but we can use any test suite to test Here we have used LLVM TEST SUITE.llvm test suite provides differnt bench marks to test as well as MultiSource tests as well .lnt provides us command line arguments to run pass with -cflags option ,it directly attach's when test file is compiling with clang option

we can run entire test-suite using the following command
lnt runtest nt –sandbox RESULTS –cc /llvm-build/bin/clang –cxx /llvm-build/bin/clang++ –test-suite /llvm-test-suite/ –test-externals /llvm-test-suite/ –cflag '-Xclang -load -Xclang /llvm-build/lib/ControlFlattening.so' -j22
where
–cc is clang path
–cxx is clang++ path
–test-suite for selecting testsuite
–test-only to test specific directory in test-suite –
some of the flags in testing are

```
Machine␣Info:{
hardware␣:␣x86_64
os␣:␣Linux␣4.15.0-36-generic
}
{
"ARCH":␣"x86_64"
"CC_UNDER_TEST_IS_CLANG":␣"1"
"CC_UNDER_TEST_TARGET_IS_X86_64":␣"1"
"DISABLE_JIT":␣"1"
"ENABLE_HASHED_PROGRAM_OUTPUT":␣"1"
"ENABLE_OPTIMIZED":␣"1"
"ENABLE_PARALLEL_REPORT":␣"1"
"LLC_OPTFLAGS":␣"-O3"
"LLI_OPTFLAGS":␣"-O3"
"OPTFLAGS":␣"-O3"
"TARGET_CC":␣"None"
"TARGET_CXX":␣"None"
"TARGET_FLAGS":␣"-Xclang␣-load␣-Xclang␣~/llvm-build/lib/ControlFlattening.so"
"TARGET_LLVMGCC":␣"/home/cs16btech11029/llvm-build/bin/clang"
"TARGET_LLVMGXX":␣"/home/cs16btech11029/llvm-build/bin/clang++"
"TEST":␣"simple"
"cc_build":␣"DEV"
"cc_dumpmachine":␣"x86_64-unknown-linux-gnu"
"cc_name":␣"clang"
"cc_src_branch":␣"tags/RELEASE_700/final"
"cc_target":␣"x86_64-unknown-linux-gnu"
"cc_version_number":␣"7.0.0"
"tag":␣"nts"
}
```

6.1 Testing of LoopFlattening

The Loop Flattening Algorithm was tested with the entire lnt-test suite. The total number of test cases were 2555 of which only 40 of the test cases were failing.

About 14 out of 36 were failures from Single Source Benchmarks. And The rest 22 were from MultiSource benchmarks.

There were no compile time failures and the rest 36 were runtime failures.

6.2 Benchmarks: Causing runtime errors

```
MultiSource/Applications/SPASS/SPASS.execution_time
MultiSource/Applications/aha/aha.execution_time
MultiSource/Applications/lambda-0_1_3/lambda.execution_time
MultiSource/Applications/lua/lua.execution_time
MultiSource/Applications/minisat/minisat.execution_time
MultiSource/Applications/obsequi/Obsequi.execution_time
MultiSource/Applications/siod/siod.execution_time
MultiSource/Applications/sqlite3/sqlite3.execution_time
MultiSource/Benchmarks/ASC_Sequoia/AMGmk/AMGmk.execution_time
MultiSource/Benchmarks/ASC_Sequoia/CrystalMk/CrystalMk.execution_time
MultiSource/Benchmarks/MallocBench/gs/gs.execution_time
```

```
MultiSource/Benchmarks/McCat/05-eks/eks.execution_time
MultiSource/Benchmarks/MiBench/automotive-bitcount/automotive-bitcount.
MultiSource/Benchmarks/MiBench/telecomm-FFT/telecomm-fft.execution_time
MultiSource/Benchmarks/PAQ8p/paq8p.execution_time
MultiSource/Benchmarks/Prolangs-C++/city/city.execution_time
MultiSource/Benchmarks/Ptrdist/ft/ft.execution_time
MultiSource/Benchmarks/Ptrdist/ks/ks.execution_time
MultiSource/Benchmarks/Ptrdist/yacr2/yacr2.execution_time
MultiSource/Benchmarks/SciMark2-C/scimark2.execution_time
MultiSource/Benchmarks/mediabench/g721/g721encode.execution_time
MultiSource/Benchmarks/sim/sim.execution_time
SingleSource/Benchmarks/BenchmarkGame/fannkuch.execution_time
SingleSource/Benchmarks/BenchmarkGame/puzzle.execution_time
SingleSource/Benchmarks/CoyoteBench/huffbench.execution_time
SingleSource/Benchmarks/Dhrystone/dry.execution_time
SingleSource/Benchmarks/Adobe-C++/functionobjects.execution_time
SingleSource/Benchmarks/Dhrystone/fldry.execution_time
SingleSource/Benchmarks/Misc-C++-EH/spirit.execution_time
SingleSource/Benchmarks/Shootout-C++/hash.execution_time
SingleSource/Benchmarks/Shootout-C++/hash2.execution_time
SingleSource/Benchmarks/Shootout-C++/nestedloop.execution_time
SingleSource/Benchmarks/Shootout/matrix.execution_time
SingleSource/Benchmarks/Shootout/nestedloop.execution_time
SingleSource/Benchmarks/Shootout/sieve.execution_time
SingleSource/Benchmarks/Misc/lowercase.execution_time
```

6.2.1 Reasons for Runtime failure

The reason behind the runtime failures were the srand() function. Due to the usage of srand in our pass for generating the random number during runtime caused the seed value of the RNG to become "off" the value expected by the test cases.

Because of this the data structures which used the RNG seeded by their expected value were becoming corrupt by the seeding of the RNG by our pass. This lead to failures in the expected outputs.

6.2.2 Idea to tackle this problem

One Idea is to manipulate the name of the srand function in LLVM IR by using setName() function. This piece of code is commented in the source code. To enable this function one can uncomment it.

6.3 Testing of BranchFlattening

The Branch Flattening was tested only for basic cases where the if-else structure is visible in the IR. The execution of the executable generated after transformation showed no difference.

Chapter 7

Bibliography

7.1 References

1. http://llvm.org/docs/ProgrammersManual.html#creating-and-inserting-new-instructions

2. https://link.springer.com/content/pdf/10.1007%2F978-3-540-87785-1_36.pdf

3.http://www2.cs.arizona.edu/~collberg/Teaching/553/2011/Resources/obfuscation.pdf

4.http://llvm.org/doxygen/

5.https://www.inf.u-szeged.hu/~akiss/pub/pdf/laszlo_obfuscating.pdf

Other sites like stackoverflow.com, stackexchange.com, wikipedia.com

6. http://llvm.org/docs/lnt/quickstart.html

7.https://media.readthedocs.org/pdf/lnt/latest/lnt.pdf