Compiler Construction CS 201 - Winter 2017
Project Proposal

Abdulrahman Aloraini
University of California Riverside
900 University Ave.
Riverside, CA 92521, USA
aalor001@ucr.edu

Dang Tu Nguyen
University of California Riverside
900 University Ave.
Riverside, CA 92521, USA
tnguy208@ucr.edu

1. INTRODUCTION

The LLVM (Low Level Virtual Machine) Project is a collection of modular and reusable compiler and toolchain technologies. LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of sub-projects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research [7].

LLVM comes in three pieces. The first piece is the LLVM suite. This contains all of the tools, libraries, and header files needed to use LLVM. It contains an assembler, disassembler, bitcode analyzer and bitcode optimizer. It also contains basic regression tests that can be used to test the LLVM tools and the Clang front end. The second piece is the Clang front end. This component compiles C, C++, Objective C, and Objective C++ code into LLVM bitcode. Once compiled into LLVM bitcode, a program can be manipulated with the LLVM tools from the LLVM suite. There is a third, optional piece called Test Suite. It is a suite of programs with a testing harness that can be used to further test LLVM's functionality and performance [5].

In this project, we will study two existing passes of LLVM: "*Canonicalize natural loops*" and "*Extract loops into new function*", and implement two new passes: "*function frequencies*" and "*call sequence*".

The paper is organized as follows: We will explain our motivation for choosing "*Canonicalize natural loops*" and "*Extract loops into new function*" and the new passes in Section 2. We will extensively study the two existing LLVM passes and its functions in Section 3. In Section 4, we will show the two existing performance on different benchmarks from LLVM library. Section 5 explain in details the new passes and verify them on benchmarks from Worst Case Execution Time (WCET).

Compiler Construction Winter 2017, UCR, USA

DOI: 10.475/123_4

2. MOTIVATION

Most execution time is spent on loops. Loops play an important role in every program and mainly responsible for performance efficiency. Therefore, we need to verify that every computation within loops is necessary. As shown in Figure 1, we see that computation $A = B + C$ is used by other computations inside the loop. However, the computation wastes resources because it remains unchanged during loop execution. Thus, it is better to move out the computation $A = B + C$ before the loop. The challenge is that the loop header is dominated by three predecessors, and copying the computation to all predecessor blocks is wasteful.

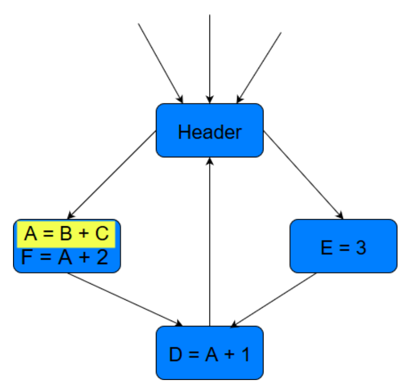

Figure 1: Bad Loop (1)

Another challenge is shown in Figure 2 where we have computation $C = A + 4$. The computation only needs to be updated only after the loop terminates. However, moving out the computation after the loop will change the program semantic.

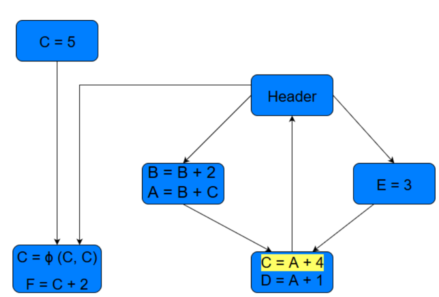

Figure 2: Bad Loop (2)

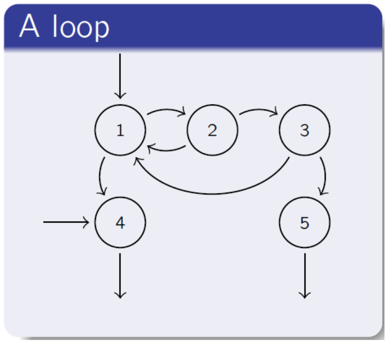

Figure 3: A loop example.

Every unnecessary computation inside loops could damage the performance dramatically. It is easy to manually optimize programs if they are relatively small, but it is extremely hard for large programs. Therefore, we decided to study passes related to loops because they are computationally interesting problems. LLVM handles several passes and one of the most important optimization module is *LoopPass* class. *LoopPass* has many passes that optimize loops. Among all loop passes within *LoopPass*, we decided to study "*Canonicalize natural loops*" and "*Extract loops into new function*". "*Canonicalize natural loops*" plays an important role in almost all loop passes because it handles natural loops. "*Extract loops into new function*" is the most useful pass for debugging because it extracts loops into new functions; thus, requires using a passes from *LoopPass* and *BlockPass* modules. With "*Extract loops into new function*", debugging process can be faster and more efficient because loops are extracted. So when an optimization crashes, we will be able to know where the crash happened. Both use Loop-invariant code motion (LICM) techniques to move computations outside of loops without affecting the loop semantics of the function.

To understand better LLVM environment, we decided to implement two new passes. The first pass is "*function frequencies*" which finds how many times each function has been called. The second pass is "*call sequence*" which shows the running time progress of a program. We believe these two new passes can help to understand and find which function consumes resources the most and how program works.

3. STUDY TWO EXISTING LLVM PASSES

Before discussing the two LLVM passes, we will briefly explain some loop terminology with an example in Figure 3 [11].

- **Back-edge**: edge entering loop header (*e.g.*, edge 3->1)
- **Header**: loop entry node (*e.g.*, node 1); or node that dominates all other nodes in a loop [2]
- **Body**: nodes that can reach back-edge source node (3) without passing from back-edge target node (1) plus back-edge target node (*e.g.*, the set of nodes {1, 2, 3})
- **Exiting**: nodes with a successor outside the loop (*e.g.*, the set of nodes {1, 3})
- **Exit**: nodes with a predecessor inside the loop (*e.g.*, the set of nodes {4, 5})
- **Natural loop** of a back edge: smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header [2]

Figure 4 is a good example to understand what a natural loop is.

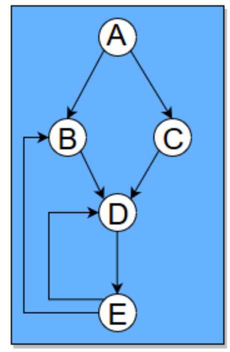

Figure 4: Only node D leads to Node E. Therefore, it is a natural loop. However, the loop between Node B and Node E is not natural because there is another path leads to Node E through Node C.

3.1 Canonicalize natural loops

This pass performs several transformations to transform natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective. Loop pre-header insertion guarantees that there is a single, non-critical entry edge from outside of the loop to the loop header. This simplifies a number of analyses and transformations, such as LICM. Loop exit-block insertion guarantees that all exit blocks from the loop (blocks which are outside of the loop that have predecessors inside of the loop) only have predecessors from inside of the loop (and are thus dominated by the loop header). This simplifies transformations such as store-sinking that are built into LICM. This pass also guarantees that loops will have exactly one back-edge. Note that the *simplifycfg* pass will clean up blocks which are split out but end up being unnecessary, so usage of this pass should not pessimize generated code. This pass obviously modifies the CFG, but updates loop information and dominator information [4].

In short, the *loop-simplify* pass normalize natural loops as following:

- **Pre-header**: the only predecessor of header node (always executed before entering the loop)
- **Latch**: the starting node of the only back-edge (always executed before starting a new iteration)
- **Exit-block**: ensures exist dominated by loop header (always executed after exiting the loop)

As shown in Figure 5, node 0 is the inserted pre-header one [11].

Figure 6 shows the insertion of a latch node, 6 [11].

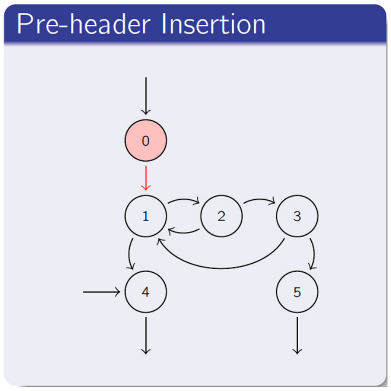

Figure 5: An example of pre-header insertion.

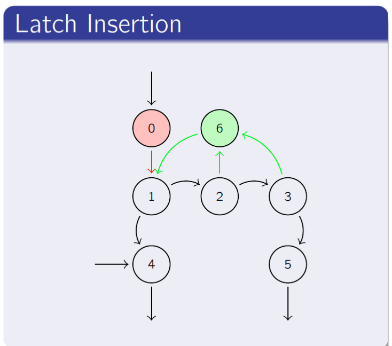

Figure 6: An example of latch insertion.

As shown in Figure 7, node 7 is the inserted exit-block [11].

One raising problem is that updating code in loop might require to update code outside loops for keeping SSA, which is expensive. This issue can be handled by the pass *lcssa*, which insert **phi** instruction at loop boundaries for variables defined inside the loop body and used outside. This guarantee isolation between optimization performed inside and outside the loop.

3.1.1 Implementation investigation (Canonicalize natural loops)

InsertPreheaderForLoop Method

Purpose: If a loop does not have a pre-header, this method is called to insert one. This method will make sure that loop's header has only one predecessor.

Parameters:

1. Loop **L*: The loop will be checked
2. DominatorTree **DT*: Dominator tree of the procedure containing the loop *L*
3. LoopInfo **LI*: Info of the loop *L*
4. bool *PreserveLCSSA*: To notify that if LCSSA needs to be preserved

Code flow's description:

1. *Header* <- Get loop's header

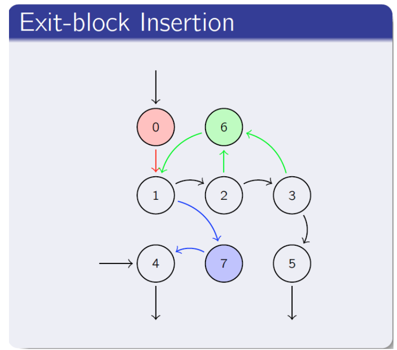

Figure 7: An example of exit-block insertion.

2. *OutsideBlocks* <- Compute the set of predecessors of *Header* that are not in the loop
3. *PreheaderBB* <- Create a new basic block and moves some of predecessors in the list *OutsideBlocks* to be predecessors of this new block. This new block will become the only predecessor of *Header*
4. Place the *PreheaderBB* after a block outside of the loop if it is not already in such a position
5. Return *PreheaderBB*

rewriteLoopExitBlock Method

Purpose: This method is used to split exit blocks that have predecessors outside of the loop. It will make sure that the loop pre-header dominates all exit blocks.

Parameters:

1. Loop **L*: The loop will be checked
2. BasicBlock **Exit*: The exit block that has predecessors outside of the loop
3. DominatorTree **DT*: Dominator tree of the procedure containing the loop *L*
4. LoopInfo **LI*: Info of the loop *L*
5. bool *PreserveLCSSA*: To notify that if LCSSA needs to be preserved

Code flow's description:

1. *LoopBlocks* <- Get the list of predecessors of *Exit* that are inside the loop
2. *NewExitBB* <- Create a new basic block and moves some of predecessors in the list *LoopBlocks* to be predecessors of this new block. This new block will become the predecessor of *Exit*
3. Return *NewExitBB*

separateNestedLoop Method

Purpose: If this loop has multiple back-edges, try to pull one of them out into a nested loop. This method will make sure that a basic block can be the header of only one loop.

Parameters:

1. Loop **L*: The loop will be checked
2. BasicBlock **Preheader*: The pre-header of the loop *L*

3. DominatorTree *DT: Dominator tree of the procedure containing the loop L

4. LoopInfo *LI: Info of the loop L

5. ScalarEvolution *SE: Scalar evolution info

6. bool *PreserveLCSSA*: To notify that if LCSSA needs to be preserved

7. AssumptionCache *AC: Assumption cache info

Code flow's description:

1. *Header* <- Get header of the loop L

2. *PN* <- Get the PHI node to partition. This PHI node must be in the header of the loop and has unchanging values on one back-edge correspond to values, which change in the "outer" loop, but not in the "inner" loop

3. *OuterLoopPreds* <- Pull out all predecessors that have varying values in the loop

4. Reset *SE* if it exists

5. *NewBB* <- Create a new basic block and moves some of predecessors in the list *OuterLoopPreds* to be predecessors of this new block. This new block will become the predecessor of *Header*

6. Make sure that NewBB is put someplace intelligent, which does not mess up code layout too horribly

7. *NewOuter* <- Create the new outer loop

8. Change the parent loop to use *NewOuter* as its child now

9. Make L become a sub-loop of *NewOuter*

10. Reset header of L to *Header*

11. *BlocksInL* <- Get blocks inside L that are dominated by *Header*

12. Scan all of the loop children of L, moving them to OuterLoop if they are not part of *BlocksInL*

13. Split edges to exit blocks from the inner loop L, if they emerged in the process of separating the outer one *NewOuter*

14. Return *NewOuter*

<u>*insertUniqueBackedgeBlock* Method</u>

Purpose: This method is called when the specified loop has more than one back-edge in it. If this occurs, re-vector all of these back-edges to target a new basic block and have that block branch to the loop header. This ensures that loops have exactly one back-edge.

Parameters:

1. Loop *L: The loop will be checked

2. BasicBlock **Preheader*: The pre-header of the loop L

3. DominatorTree *DT: Dominator tree of the procedure containing the loop L

4. LoopInfo *LI: Info of the loop L

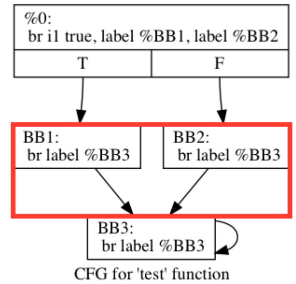

Figure 8: CFG of *basictest.ll* before applying loop-simplify.

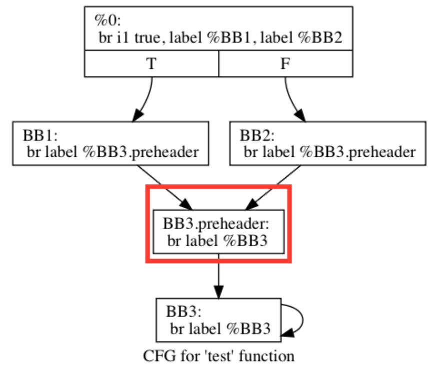

Figure 9: CFG of *basictest.ll* after applying loop-simplify.

Code flow's description:

1. *Header* <- Get header of the loop L

2. *BackedgeBlocks* <- Get basic blocks that contain back-edges to the *Header*

3. *BEBlock* <- Create a new black-edge block and make this new block jump to *Header*

4. Move *BEBlock* to right after the last element of *Backedge-Blocks*

5. For each PHINode *PN* in the *Header*: Create a new PHINode *NewPN*; Move all entries of *PN* except the one for the pre-header over to *NewPN*; Delete all of the incoming values from the old *PN* except the pre-header's; Add the newly constructed PHI node *NewPN* as the entry for the *BEBlock*; If all incoming values in *NewPN* (which is a subset of the incoming values of *PN*) have the same value, eliminate *NewPN* from *BEBlock*

6. Make all back-edge blocks in *BackedgeBlocks* to jump to the *BEBlock* instead of the *Header*. If one of the back-edges has llvm.loop metadata attached, remove it from the back-edge and add it to *BEBlock*

7. Update L and LI with the info of *BEBlock*

8. Update *DT*

9. Return *BEBlock*

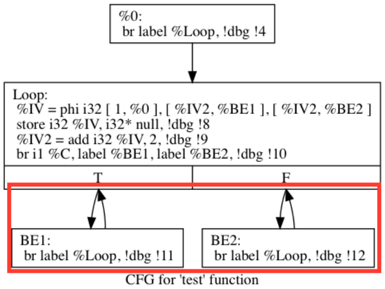

Figure 10: CFG of *single-backedge.ll* before applying loop-simplify.

simplifyOneLoop Method

Purpose: Simplify one loop and queue further loops for simplification.

Parameters:

1. Loop *L: The loop will be checked
2. SmallVectorImpl<Loop *> &$Worklist$: List of all loops
3. DominatorTree *DT: Dominator tree of the procedure containing the loop L
4. LoopInfo *LI: Info of the loop L
5. ScalarEvolution *SE: Scalar evolution info
6. bool *PreserveLCSSA*: To notify that if LCSSA needs to be preserved
7. AssumptionCache *AC: Assumption cache info

Code flow's description:

1. *BadPreds* <- Get list of blocks that are predecessors of blocks (other than the header) inside the loop
2. For each block in the list *BadPreds*: Delete the edges from this block.
3. If L does not have a pre-header, insert one using the method *InsertPreheaderForLoop*
4. If an exit block has predecessors that are not inside of the loop, split the edge using the method *rewriteLoopExitBlock*
5. If the header has more than two predecessors at this point (from the pre-header and from multiple back-edges), adjust the loop *separateNestedLoop*
6. Make sure the loop has only one back-edge by calling the method *insertUniqueBackedgeBlock*
7. Scan over the PHI nodes in the loop header. Since they now have only two incoming values (the loop is canonicalized), we may have simplified the PHI down to *X = phi [X, Y]*, which should be replaced with *Y*
8. If this loop has multiple exits and the exits all go to the same block, attempt to merge the exits

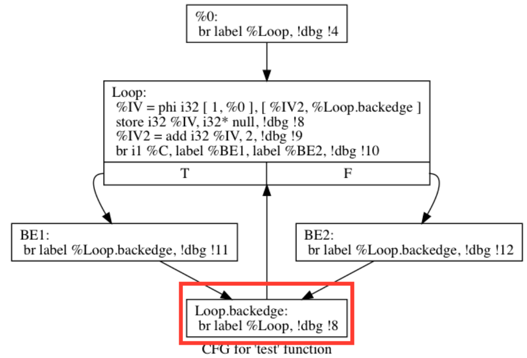

Figure 11: CFG of *single-backedge.ll* after applying loop-simplify.

3.2 Extract loops into new function

This pass transforms top-level loops into new functions. If a function has only one loop, the loop remains without any changes [4]. This pass is the most useful pass for debugging because it reduces test cases of a program more than the other passes [6]. Within the pass, there are four sub-passes:

- **createLoopExtractorPass**: This pass extracts all natural loops from the program into a function.
- **createSingleLoopExtractorPass**: This pass extracts one natural loop into a function if it is possible so it can be used by bug-point.
- **createBlockExtractorPass**: This pass extracts all basic blocks (unless those are in the argument list) from the functions [8].
- **BlockExtractorPass**: This pass is used by bug-point to extract all blocks from the module into their own functions except for those specified by the *BlocksToNotExtract list*.

3.2.1 Implementation investigation (Extract loops into new function)

createLoopExtractorPass Method

Purpose: If a program has more than one natural loop, this pass extract all of them into a function.

Parameters:

1. Loop *L: The loop will be checked
2. LPPassManager &: The pass manager executes *FunctionPass* and *PMDataManager.FunctionPass* implements global optimization on functions.*PMDataManager* is used to analyze data.

Code flow's description:

1. *skipOptnoneFunction(Loop *L)* <- Checks if the loop was optimized and have run transformation passes or not. If false, the method will quit, else will proceed.
2. *getParentLoop()* <- Checks if the specified loop is contained within in this loop. False if there is only one loop, and true if there is more.

3. *!isLoopSimplifyForm()* <- Checks if a loop in the simple form or not. Simple Form is when loops have a pre-header, a single back-edge, and all of their exits have all their predecessors inside the loop. If the loop is not simplified, the program will quit. Else, the program will proceed.

4. DominatorTree & DT <- Analyzes and saves the dominator tree of the program.

5. LoopInfo & LI <- Contains information of the processed loop.

6. *EntryTI* <- Extracts the loop only when the entry block does not branch to the header of the loop. Then checks to see if any exits within the loop are more than the blocks.

7. *ShouldExtractLoop* <- after replacing the loop with a function call, we should not try to run more loop passes.

8. Return *Changed* <- true only when createLoopExtractorPass method is performed.

createSingleLoopExtractorPass Method
Purpose: This method is used to extract only one natural loop from the program into a function to use it by bug-point if it is possible.
Code flow's description:

- Return *SingleLoopExtractor()* <- Returns the single extracted loop.

createBlockExtractorPass Method
Purpose: This method is used to extract all blocks from the functions in the module unless specified in the **argument list** not to be extracted.
Code flow's description:

- Return *BlockExtractorPass()* <- Returns the extracted blocks.

BlockExtractorPass Class
Purpose: This method is used to extract all blocks from the functions in the module unless specified in the **BlocksToNotExtract list** not to be extracted.
Code flow's description:

1. *LoadFile(const char *Filename)* Method: Load a block file which contains the blocks that we should **not** be extracted

2. *SplitLandingPadPreds(Function *F)* Method: The landing pad should be extracted when the invoke instruction is called. The critical edge breaker will refuse to break critical edges to a landing pad. So do them here. After running this method, all landing pads should have only one predecessor.

3. char *ID*: To idetify the the pass.

4. BasicBlock **BB* <- List of blocks not to be extracted.

5. Function **F* <- This function returns the parent of a block.

6. iterator *BBI* <- Finds the index of the basic block in its function.

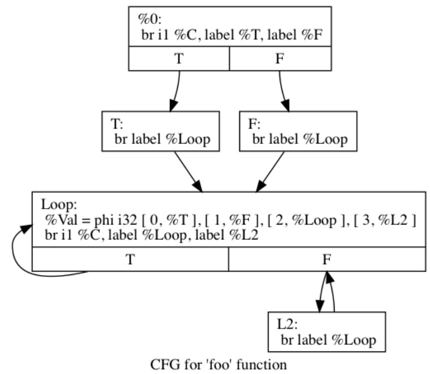

Figure 12: CFG of *hardertest.ll* before applying loop-simplify.

7. string & *FuncName* <- There is no way to find BBs so here we store the function name.

8. string & *BlockName* <- And here we store the name of the block.

9. Now that we know which blocks to not extract.

10. *BlocksToExtract* <- Adds blocks that are not specified to be extracted.

11. Return *!BlocksToExtract.empty()*

4. PERFORMANCE EVALUATION OF EXISTING PASSES

We use the benchmarks provided by LLVM in the directory /llvm/test/Transforms to verify our passes since they cover most of the corner cases. If we explain all benchmarks, the report will be very long. Therefore, we just explain the major ones which are: *basictest.ll*, *single-backedge.ll*, *hardertest.ll*, and *merge-exits.ll*.

4.1 Canonicalize natural loops

basictest.ll Benchmark: Before applying the pass, as shown in Figure 8, the loop header *BB3* has two predecessors: *BB1* and *BB2*. After applying the pass, as shown in Figure 9, we got the following results:

1. A new block *BB3.preheader* was inserted to the CFG. *BB3.preheader* became the only predecessor of the loop header *BB3*

2. The two predecessors *BB1* and *BB2* were updated to jump to the new block *BB3.preheader*, instead of *BB3*

single-backedge.ll Benchmark: Before applying the pass, as shown in Figure 10, the loop header *Loop* has two back-edges: from *BE1* and from *BE2*. After applying the pass, as shown in Figure 11, we got the following results:

1. A new block *Loop.backedge* was inserted to the CFG. *Loop* now has only one back-edge: from *Loop.backedge*

2. Changes in *BE1* and *BE2*: The back-edges to *Loop* were removed; A jump to *Loop.backedge* was added

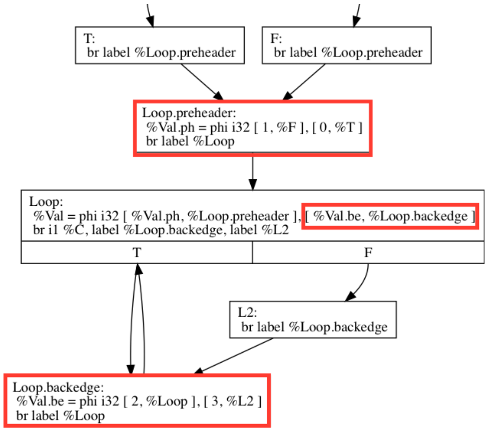

Figure 13: CFG of *hardertest.ll* after applying loop-simplify.

<u>*hardertest.ll* Benchmark</u>: As shown in Figure 12, before applying the pass, the loop header *Loop* has two predecessors: *T* and *F*, and two back-edges: from *Loop(T)* and from *L2*. After applying the pass, as shown in Figure 13, we got the following results:

1. A new block *Loop.preheader* was inserted to the CFG. *Loop.preheader* became the only predecessor of the loop header *Loop*

2. The two predecessors *T* and *F* were updated to jump to the new block *Loop.preheader*, instead of *Loop*

3. A new block *Loop.backedge* was inserted to the CFG. The loop header *Loop* now has only one back-edge: from *Loop.backedge*

4. Change in the content of the PHINode *Val* inside *Loop*: [0, %T] and [1, %F] were replaced with [%Val.ph, %Loop. pre-header]; [2, %Loop] and [3,%L2] were replaced with [%Val.be, %Loop.backedge]

5. Change in *L2*: The back-edge to *Loop* was removed; A jump to *Loop.backedge* was added

<u>*merge-exits.ll* Benchmark</u>: As shown in Figure 14, before applying the pass, the loop *bb1* has two exits that go to the same place: *bb1 -> bb3* and *bb1 -> bb2 -> bb3*. After applying the pass, as shown in Figure 15, we got the following results:

1. The loop-invariant instruction starting with %t11 in *bb2* was moved into the pre-header of *bb1*

2. The comparison instruction starting with %t12 in *bb2* was moved into the loop header *bb1*

3. A new instruction starting with %or.cond was added to *bb1*

4. The branch instruction of *bb1* was updated to jump directly to *bb* (instead of *bb1 -> bb2 -> bb*)

5. The block *bb2* was removed and the loop has only one exit: *bb1 -> bb3*

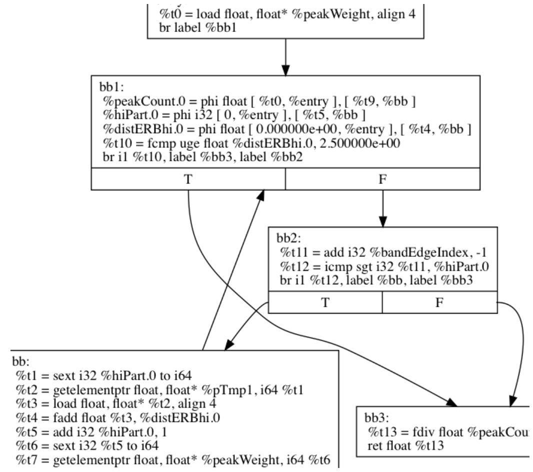

Figure 14: CFG of *merge-exits.ll* before applying loop-simplify.

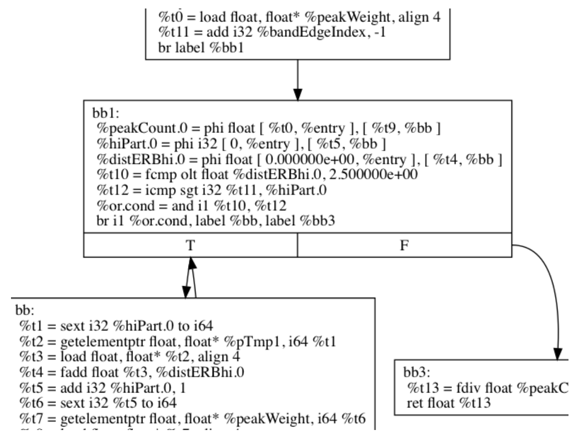

Figure 15: CFG of *merge-exits.ll* after applying loop-simplify.

4.2 Extract loops into new functions

<u>*basictest.ll* Benchmark</u>: Before applying the pass, as in Figure 8, BB3 contains a loop which was transofrmed into a new function as shown in Figure 16 and the extraced function in 17

1. A new block *BB3.preheader* was inserted to the CFG. *BB3.preheader* points to a new block that calls function *test_BB3*

2. The two predecessors *BB1* and *BB2* were updated to jump to the new block *BB3.preheader*, instead of *BB3*

3. The new block *BB3.preheader* calls the new function which contains the extracted loop.

<u>*single-backedge.ll* Benchmark</u>: Before applying extract loops pass, as shown in Figure 10, the loop header *Loop* has two back-edges: from *BE1* and from *BE2*. One of the methods in *Extract loops* is *loop simplify*. After applying the pass, as shown in Figure 11, we similarly had the same result as the previous pass because *Extract loops* pass has *loop simplify* method for unique loops. The steps are:

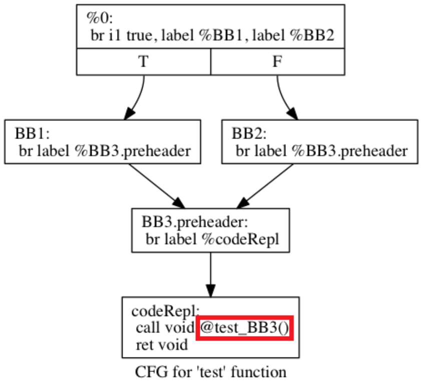

Figure 16: CFG of *basictest.ll* after applying extract-loops.

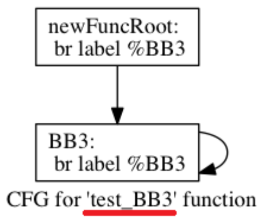

Figure 17: CFG of *basictest.ll* the new function after applying extract-loops.

1. A new block *Loop.backedge* was inserted to the CFG. *Loop* now has only one back-edge: from *Loop.backedge*
2. Changes in *BE1* and *BE2*: The back-edges to *Loop* were removed; A jump to *Loop.backedge* was added

hardertest.ll Benchmark: As shown in Figure 12, we have *Loop(T)* and *L2*. After applying the pass, as shown in Figure 18, we extracted the *Loop(T)* and *L2* into a new function called *foo_Loop*

1. The pass created *Loop.preheader* which calls the new function
2. Block *T* is turned to a critical edge block and points to a new block called *Loop.backedge* in 19
3. In 19, Block *F* still predecessor of *L2*. *L2* now has edge to *Loop.backedge*
4. Block *Loop.backedge* handles blocks *T* and *L2*. After finding the **phi** value, it loops back to *Loop* block as in 19
5. Proceed until the function completes

merge-exits.ll Benchmark: As shown in Figure 14, before applying the pass, the loop *bb1* has two exits that go to the same place: *bb1 -> bb3* and *bb1 -> bb2 -> bb3*. After applying the pass, as shown in Figure 20, we extracted block *bb1* into a new function as in 21

1. The pass extracted *bb1* into a new function including block textitbb2
2. Since there is a nautral loop between *bb1* and *bb*, both blocks are extracted to the new function *test1_bb1*

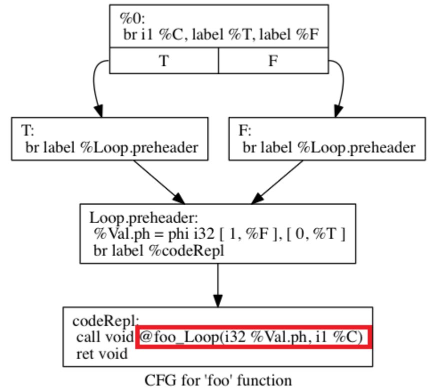

Figure 18: CFG of *hardertest.ll* after applying extract-loops.

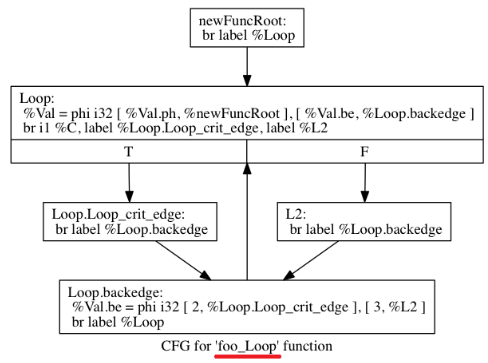

Figure 19: CFG of *hardertest.ll* the new function after applying extract-loops.

3. Two critical edge blocks are created. One responds to *bb1* and the other to *bb2* and both return to *bb3*
4. In the CGF, there are two new edge critical edge blocks that respond to the ones in the new function.
5. Both critical edge block proceed to *bb3*

5. DESIGN AND IMPLEMENT NEW LLVM PASSES

In this project, we have implemented two new passes: *call sequence* and *function frequencies*, which print out the execution sequence of functions and its call frequencies at each specific run.

5.1 Implementation of new passes

5.1.1 Call sequence

This pass inserts an instruction, which prints out the name of a function, to the entry block of a function. Since the entry block is always executed, this will make sure that the name of the function is always printed out once the function is executed.

Here are the steps to insert a printing instruction to a basic block:

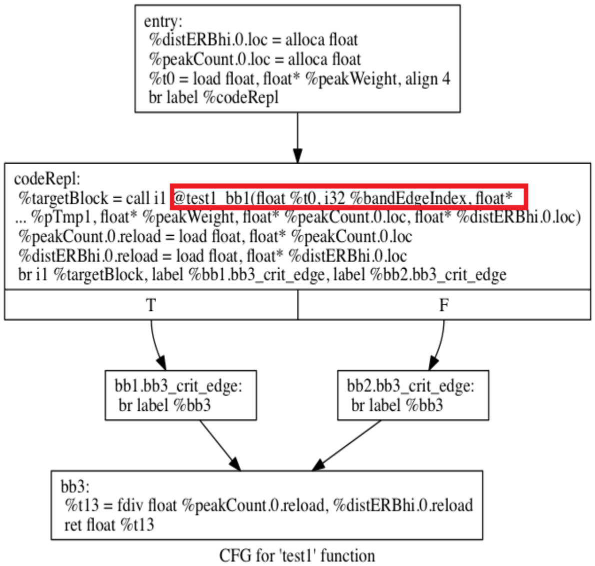

Figure 20: CFG of *merge-exits.ll* after applying extract-loops.

1. Create an IR Builder and specify the inserting location (before the first non-PHIorDbg instruction) using the following command:
   IRBuilder<> Builder(BB.getFirstNonPHIOrDbg());

2. Define the printf instruction using the following commands:
   vector<Type *> Args;
   Args.push_back(Type::getInt8PtrTy(*Context));
   FunctionType *PrintfType = FunctionType::get (Builder.getInt32Ty(), Args, true);
   Constant *PrintfFunc = CurrentModule -> getOrInsertFunction ("printf", PrintfType);
   Value *FormatStr = Builder.CreateGlobalStringPtr ("%s\n");
   Value *funcNamePtr = Builder.CreateGlobalStringPtr (funcName);

3. Create a call to printf function using the following command: vector<Value *> Values;
   Values.push_back (FormatStr);
   Values.push_back (funcNamePtr);
   CallInst *Call = Builder.CreateCall (PrintfFunc, Values);
   Call->setTailCall(false);

5.1.2 *Function frequencies*

To implement this pass, we need a global counter for each function. Thus, we use a dense map variable for this purpose: *DenseMap < Function*, GlobalVariable*> FunctionCounterMap;*. The counter of each function is initialized to 0 at function *doInitialization()*.

This pass inserts an instruction, which increase the counter of a function by 1, to the entry block of a function. Since the entry block is always executed, this will make sure that the counter of the function is always incremented once the function is executed. Moreover, this function also insert an instruction, which prints out the call frequencies of a function, to the block of function main that has a return instruction. This will make sure that the function frequencies are printed out before exit the program.

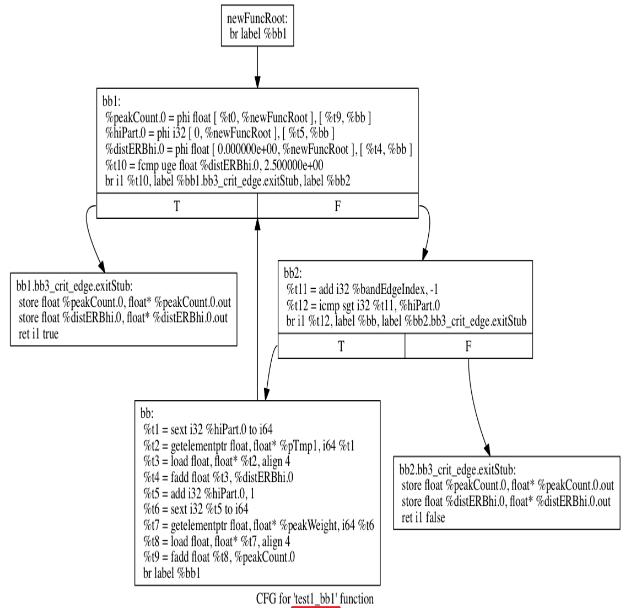

Figure 21: CFG of *merge-exits.ll* the new function after applying extract-loops.

Here are the steps to insert a incrementing instruction to a basic block:

1. Create an IR Builder and specify the inserting location (before the first non-PHIorDbg instruction) using the following command:
   IRBuilder<> Builder(BB.getFirstNonPHIOrDbg());

2. Load the counter of the function F into LoadAddr:
   Value *LoadAddr = Builder.CreateLoad (FunctionCounterMap[&F]);

3. Increment the counter by 1:
   Value *AddAddr = Builder.CreateAdd (ConstantInt::get (Type::getInt32Ty (*Context), 1), LoadAddr);

4. Store the counter back to the global variable:
   Builder.CreateStore (AddAddr, FunctionCounterMap[&F]);

5.2 Evaluation of new passes

First, we verified our passes with our own simple benchmark, which is shown in the Figure 22. As shown in Figure 23, our passes work properly.

Second, we verified our passes with different benchmarks from *http://www.mrtc.mdh.se/projects/wcet/benchmarks.html*. The result can be found at *CS201Group1/Benchmarks*.

5.3 How to compile and use our new passes?

Here are the steps to compile:

1. Copy the source code under *CS201Group1/Source Code/* and paste it under *.../llvm/projects/*

2. Open a terminal

```c
int multiply(int a, int x)
{
    return a*x;
}
int add(int a, int x)
{
    return a+x;
}
int foo(int a, int x)
{
    a += 10;
    return add(a,x);
}
int main() {
    int a = 10;

    for(int i = 0; i < 5; i++) {
        if (i%2 == 0)
            a = multiply(a,i+1);
        else
            a = foo(a,i+1);
    }
    multiply(5,10);

    return 0;
}
```

Figure 22: Simple benchmark used to verify new passes.

```
eduroam1-1-10-25-192-121:benchmarks tunguyen$ llc test_r.bc
eduroam1-1-10-25-192-121:benchmarks tunguyen$ gcc test_r.s -o test_r
eduroam1-1-10-25-192-121:benchmarks tunguyen$ ./test_r
****** Call sequence ******
main
multiply
foo
add
multiply
foo
add
multiply
multiply

****** Function frequencies ******
multiply: 4
foo: 2
main: 1
add: 2
eduroam1-1-10-25-192-121:benchmarks tunguyen$ 
```

Figure 23: Output when applying our new passes to the simple benchmark.

3. $ cd build
4. $ cmake -G “Unix Makefiles” ../llvm
5. $ make

Here are the steps to run the passes:

1. $ clang -emit-llvm -O0 <input.c> -c -o <input.bc>
2. $ opt -load myCS201lib.dylib -myCS201 <input.bc> > <result.bc>
3. $ llc result.bc
4. $ gcc result.s -o result
5. $ ./result

6. CONCLUSION

In this paper, we studied two existing passes “*Canonicalize natural loops*” and “*Extract loops into new function*”. “*Canonicalize natural loops*” basically optimizes loops by inserting *Pre-header*, *Latch*, and *Exit-Block* and the pass moves computations to these inserted blocks accordingly. “*Extract loops into new function*” extracts loops into new function using mainly for sub-passes: *createLoopExtractorPass*, *createSingleLoopExtractorPass*, *createBlockExtractorPass*, and *BlockExtractorPass* in order to facilitate debugging process. We analyzed “*Canonicalize natural loops*” and “*Extract loops into new function*” source code and test them on LLVM Transforms benchmarks. For new passes, we implemented two passes *call sequence* and *function frequencies*. *call sequence* which shows the running function sequence of a program. *function frequencies* which counts how many times each function has been called. We verified our new passes implementation on Worst Case Execution Time (WCET) benchmarks.

7. REFERENCES

[1] F. Allen. *Control flow analysis*. ACM SIGPLAN, IL, USA, 1970.

[2] S. Campanoni. *Code analysis and transformation - Loops*. Northwestern University, IL, USA, 2017.

[3] K. D. Charles Severance. *Timing and Profiling - Basic Block Profilers*. OpenStax-CNX, TX, USA, 2010.

[4] LLVM. Canonicalize natural loops and extract loops (http://llvm.org/docs/passes.html), January 2017.

[5] LLVM. Getting started with the llvm system (http://llvm.org/docs/gettingstarted.html), January 2017.

[6] LLVM. Llvm bugpoint tool: design and usage), January 2017.

[7] LLVM. Llvm overview (http://llvm.org/), January 2017.

[8] LLVM. Loopextractor.cpp (http://llvm.org), January 2017.

[9] LLVM. Source code of loop-simplify (http://llvm.org/docs/doxygen/html/loopsimplify8cpp.html), January 2017.

[10] LLVM. Writing an llvm pass (http://llvm.org/docs/writinganllvmpass.html), January 2017.

[11] M. Scandale. *LLVM Passes*. University of Politecnico di Milano, Italy, April 2012.