1 Introduction

I work solely on this project. The detected infeasible paths for the given benchmarks are appended with source code in the project deliverable zip. The IR (**.ll**) we used for detection are also appended in the zip. The LLVM build, IR generation, and pass run commands are available in the **Appendix**.

The report is structured into four parts. In the following section, we will briefly illustrate the problem and the solution presented in the reference paper to solve the shortest infeasible paths and correlated branches. We then illustrate the functionality of our works with some code sample and its CFG retrieved from LLVM. The general workflow of algorithm and some implementation details are also included in that section. Next, since LLVM utilizes the SSA form intermediate representations, there are some possible optimizations aside from the original algorithm illustrated in paper. Several implemented optimizations along with the corresponding code sample are presented in section 3. The experiment results of the three provided benchmarks along with some meta information are included in the last section.

2 Workflows & Implementation

In this section, we illustrate with code sample and the corresponding generated CFG to demonstrate the principles of the algorithm.

2.1 Algorithm Review

We have implemented the intraprocedural correlation analysis part and the shortest infeasible path detection based on the correlation analysis [1]. There are two steps in the branch correlation analysis. Starting from each candidate predicate, a *Query* ***q*** in the form of $\{var < c?\}$ are raised, where var is a variable and c is a constant.

1. The first step is to find the place of instruction where a answer can be retrieved for that q. The *Query* is propagated backward to the all predecessors. During the backward propagation, we will need to generate equivalent *Query* due to the predicate subsuming and assignment. And the initial *Query* might end up as several different ones, considering the loops inside the control flows. If the final assignment or there are some unpredictable results, such as arguments and function calls, the *Query* is resolved to be **UNDEF**. If the *Query* can be inferred and linked with some constants, it can be solved to either **TRUE** or **FALSE** at certain nodes.
2. The next step is a forward method which propagates the resolved results from predecessors to successors. It is an iterative method. Once there are new results available, the successors nodes will be reevaluated and propagate those results in further.
3. The final step is to find a path ending at the initial predicate node that the query and its equivalence along the path with the same resolved answers (either **TRUE** or **FALSE**). By doing so, the identified path can be used as the infeasible paths for the contrary result.

2.2 Sample Infeasible Paths

The infeasible paths refers to path from a node to the predicate where a answer of predicate is not possible. Due to the early assignments of some variables, it is common to see a longer path. This algorithm focus on the shortest infeasible paths. **Figure 1** is generated with the sample benchmark from paper. One of infeasible path for **TRUE** from our implementation. Combining with the original code, the predicate is to check $v == 5?$. By inspecting the actual instruction trips, we find that this is the result of assignment $v = w$, preventing the predicates to be **TRUE** and make the path infeasible.

```
Predicate: [ if.then5:   %1 = load i32, i32* %v, align 4  (Equal)  5 ]
TRUE Infeasible Path:
while.body, if.then5, if.end8, if.end11, if.then13, if.end17, while.body, if.then5,
```

Listing 1: Predicate & TRUE Infeasible Path

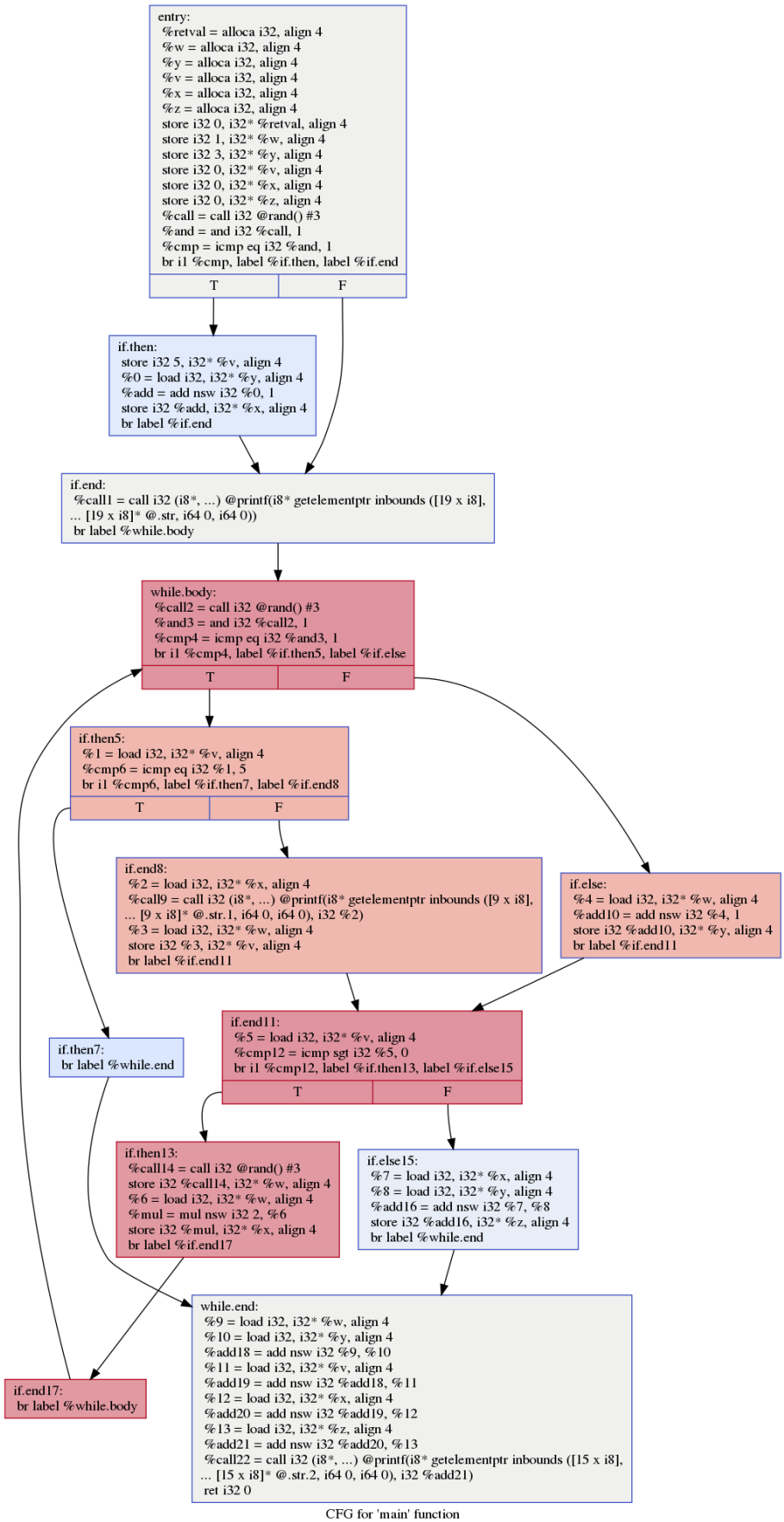

Figure 1: Sample Benchmark CFG

2.3 Implementation Details

We focus our implementation on the integer type. In general, a module pass is implemented to reflect the intraprocedural aspect of algorithm. For each function in the module, We first extract the candidate branches from By implementing a self-defined Query comparison and hash value (converting both to a signed 64 bit integer for comparisons), the natural conversion between the signed and unsigned are implemented. Several operators aside from $>$ are supported, including $==$ and $!=$. During step 1, we focus on the **LOAD** and **STORE**, and replace the target register. For each replacement or condition changes, we generate a equivalent *Query*. Two types of correlated branch detection are implemented. The first one is the constant and the second one is subsuming. As for subsuming, we merely implement very simple replace mechanism and remains as unsure for other conditions. The second step is an iterative method. We rewrite the step 3 to a while-loop dfs search rather than following the pseudocode from paper. The general logic is to detect the shortest path to the end node with single answer.

3 Optimizations

Given the SSA natures of LLVM, we implement two optimizations to improve the performances of original paper. These GVN-based optimizations are treated as addons inserted in the original workflows of detecting infeasible paths. We now illustrate the details with sample to show them.

3.1 Reconstruction of Lexical Order

We implement a predicate reordering mechanism, by swapping predicate operator, to unify the representation of *Query*. If the predicates are in the different order $\{c < var?\}$, we reorder the predicate. Also, this reorder are also used for the inference of binary operations.

3.2 Initial Constant Propagation

The original paper only handles the predicates with single variable. Once that single variable is the form of some binary operations, such as *Add* and *Sub*, it is hard to generate the equivalent query. However, given the easiness of constant detection in SSA form, we pre-check the variables and replace some of them to constant before feeding the predicates to the implemented algorithm for infeasible path detection.

3.3 Binary Operation Reflections

Given the format of $\{var < c?\}$, it is easy to have the eager to replace the *Query* where v is the result of some binary operators, saying **Add** and **Sub**. We implement this and test in our simple cases. However, it no longer works for the provided benchmarks, saying **SPEC** and **LULESH**. After inspecting the reasons, we found that it is the binary operator inside while loops, which results the queries keeping changes and thus an infinite loop. Since there is no way to infer the loop condition from analysis, the step 1 will run infinitely. We assume this is also the reason that this paper focus on constant, rather than extending to more complex cases.

4 Experiment Results

The experiments are conducted in **Ubuntu 18.04** with **LLVM 11.0.0**.

There are three benchmarks are tested for our program. Another module pass aside from the algorithm module pass for collecting the meta information, such as predicate number and function counts, are also implemented. The meta information of the benchmarks are presented as below. Total function number is in the first column. Since we only focus on the integer predicates, the total number of predicates and the number of int and target predicates are listed at the second and third columns.

Table 1: Meta Info

	Total Functions	*Total Predicates*	*Int Predicates*	*Target Predicates*
Infeasible Path	3 (random and print included)	4	4	4
LULESH	331	324	249	101
SPEC	165	1336	1336	827

We first generate single **.ll** IR file for each benchmark and we notice that the original IR has some phi representation, which is somehow hard for us to cover, since it is a branch command. Before feeding the IR to the implementation, we first pass the IR to a LLVM pass, reg2mem, to remove the phi.

A summary for the meta information of the correlated predicates are shown in following. The correlated predicate refers to a predicate with either **TRUE** or **FALSE** at some nodes after step 1.

Table 2: Predicate Correlation

	Target Predicates	*Correlated Predicates*	*Percentages*
Infeasible Path	4	2	50%
LULESH	101	29	28.71%
SPEC	827	99	1.12%

The initial detected shortest infeasible paths are in the representation of instructions (nodes in paper). We transform them to the name of basic block. The detailed paths are included in the zip file along with its corresponding predicate.

5 Appendix

The commands used for running LLVM pass are appended as below. The LLVM build instructions are also included.

```
$LLVM-Project-Dir:
cmake -DLLVM_ENABLE_PROJECTS=clang -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_ASSERTIONS=On -
    DLLVM_TARGETS_TO_BUILD=host ../llvm

$LLVM-Build-Dir:
bin/opt -load lib/LLVMInfeasiblePath.so -MetaInfo -disable-output LLVM-IR
bin/opt -load lib/LLVMInfeasiblePath.so -InfeasiblePath -disable-output LLVM-IR
```

Listing 2: LLVM Build & Pass Commands

References

[1] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In *Software Engineering—ESEC/FSE'97*, pages 361–377. Springer, 1997.