Shangru Yi
GT - syi73
2021/04/25

Project Part 2 Report
Compiler Designs

CS 6241

1 Introduction

I work solely on this project. The source code and the result of constant propagation along with the introduced code size increase by the implemented ***Split*** for the given benchmarks are appended in the project deliverable zip. The generated CFG of our implementation for the code sample included in the paper are also included in the zip. The IR (**.ll**) we used for detection are also available in the zip. The name convention represents the passes.

The report is structured into three parts. In the following section, we will briefly illustrate the problem and the solution presented in the reference paper for better data analysis with the CFG restructuring. 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. The experiment results of the 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.

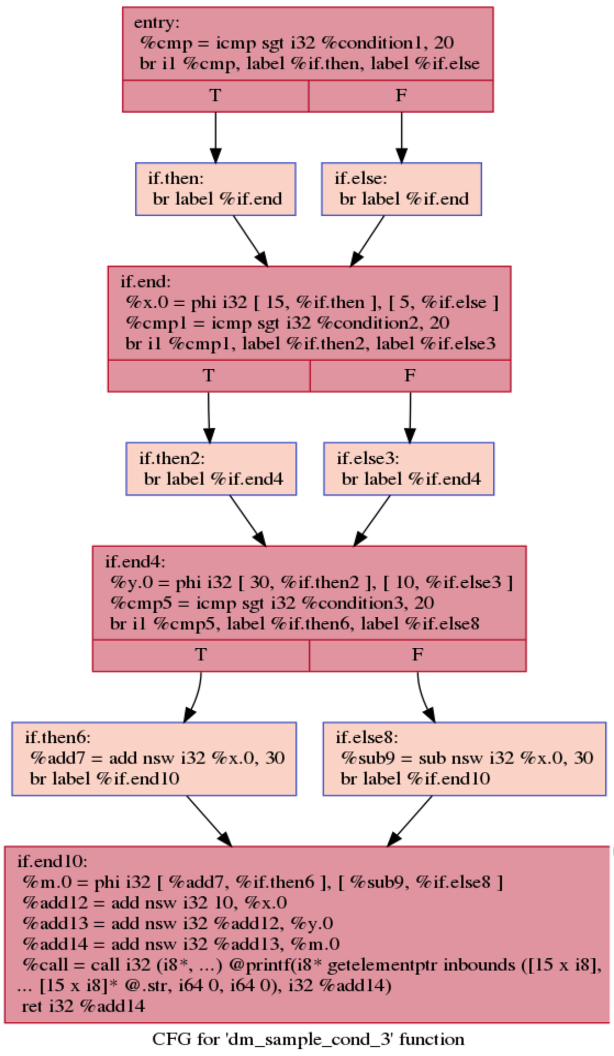
2.1 Algorithm Review

Within this paper, the idea of destructive merge, which prevents further constant propagation or other dataflow analysis are explored. The key idea is that there are some nodes (phi node in SSA form) will merge the results propagated from predecessors. Due to these merges, some certain values or useful dataflow analysis will be invalided. By splitting those merge back to separated path along with duplicated blocks, it is possible to recover those lost values and enable dataflow analysis performing separately along the split paths. There are several steps in the algorithm for the merge splits and block duplication for better dataflow propagation [1].

1. The first step is to find the influenced nodes and the region of influence (ROI) due to the destructive merges. The influenced nodes are defined as the nodes where there are optimizations available once the destroyed data facts are restored. Given the influenced nodes for a certain destructive merge, the ROI refers to the nodes that are reachable from the destructive merges with at least one path to some influenced nodes.
2. After finding the ROI of a destructive merge, we then detect the edges where the data facts are available (revival edges) and the edges where there are no more uses afterwards (kill edges). This will restrict the code size for duplication for data fact recovery. Then, each data fact will be associated with a state for the block duplication. When passing certain edges, once that edges belong to the revival edges of a certain data facts, there is a state change, indicating a duplication of the after block.
3. If we want to remove multiple destructive merges, we then chain above procedures and generate the ROI and influenced nodes in the generated CFG of previous destructive merge.

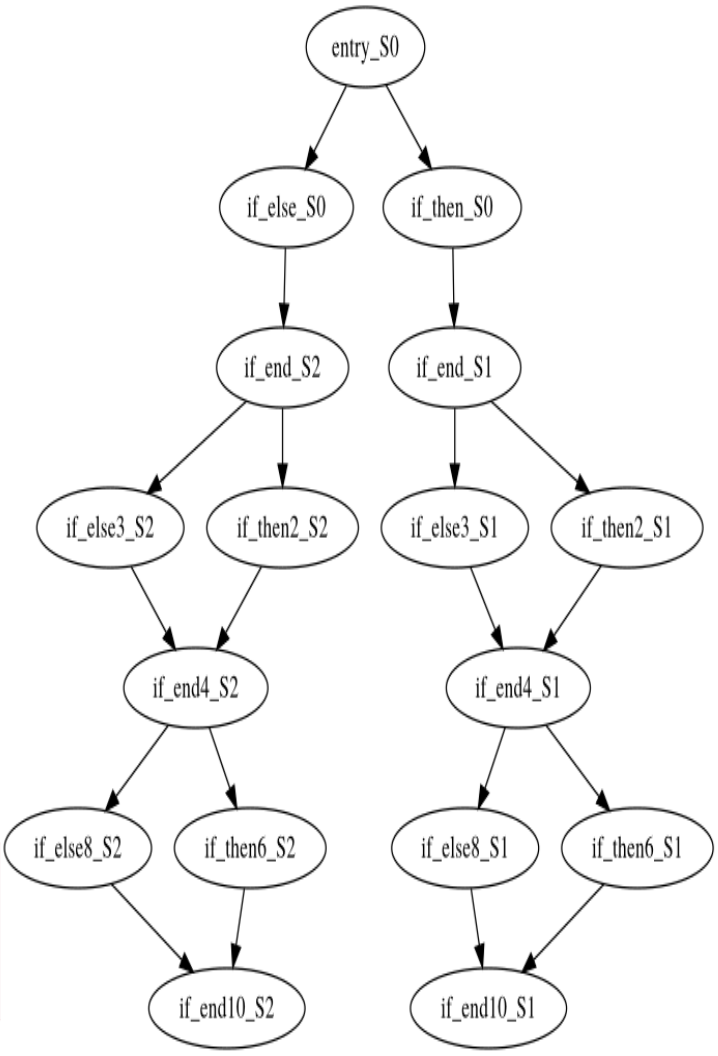
2.2 Generated CFG Sample

We generate some samples for illustrating the idea of block duplication. The generated code CFG are shown in **1a**. It is clear that there are several destructive merges shown in the CFG. For example, $\%x.0 = phi\ i32\ [\ 15,\ \%if.then\ ],\ [\ 5,\ \%if.else]$ in the $if.end$ block is a destructive merge. Before this merge, there are two constant value for $\%x.0$, i.e. 15 and 5. However, after the PHI merge, $\%x.0$ is no longer a constant and thus prevents some further optimization. We then perform the split and generate a symbolic representation of blocks shown in **1b**. The suffix represents the state of the constructed automata. It is clear that the CFG is now a clear two branch with each branch representing an unified state, i.e. a constant. Then, some further optimizations can be performed. Given this, it is clear that the split will recover some constants from the original destructive merges. Although the code size is increased, we can still expect some optimizations available.



```
entry:
 %cmp = icmp sgt i32 %condition1, 20
 br i1 %cmp, label %if.then, label %if.else
T | F

if.then:
 br label %if.end

if.else:
 br label %if.end

if.end:
 %x.0 = phi i32 [ 15, %if.then ], [ 5, %if.else ]
 %cmp1 = icmp sgt i32 %condition2, 20
 br i1 %cmp1, label %if.then2, label %if.else3
T | F

if.then2:
 br label %if.end4

if.else3:
 br label %if.end4

if.end4:
 %y.0 = phi i32 [ 30, %if.then2 ], [ 10, %if.else3 ]
 %cmp5 = icmp sgt i32 %condition3, 20
 br i1 %cmp5, label %if.then6, label %if.else8
T | F

if.then6:
 %add7 = add nsw i32 %x.0, 30
 br label %if.end10

if.else8:
 %sub9 = sub nsw i32 %x.0, 30
 br label %if.end10

if.end10:
 %m.0 = phi i32 [ %add7, %if.then6 ], [ %sub9, %if.else8 ]
 %add12 = add nsw i32 10, %x.0
 %add13 = add nsw i32 %add12, %y.0
 %add14 = add nsw i32 %add13, %m.0
 %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([15 x i8],
... [15 x i8]* @.str, i64 0, i64 0), i32 %add14)
 ret i32 %add14
```

(a) Sample CFG



(b) Destructive Merge Recovery - Symbolic CFG

2.3 Implementation Details

For a destructive merge, We first determine the reachable basic blocks using an iterative method. Once there are mode nodes reachable for a basic block, we add the predecessors of that block into the working set. We continue this behavior until the working set is empty. After determining the reachable blocks, the revival edges and kill edges can be inferred. Then the automata can be used to construct the split CFG. After we having the CFG for guides, we perform the actual code generation.

There are several stages for code generation. We first perform a simple transverse to prepare the data or register generated at each basic blocks. The terminators that branches to different blocks can be directly retrieved and updated from our generated CFG. After the initial data is prepared, we need one final iterative methods to perform the SSA transform. Due to the multi-splits, it is quite hard for this part. In general, we maintain a record of available data facts (not merely the data facts illustrated in the paper but also the register generated in each block. Once there are some data facts available at a certain basic block, we add all successors of that block to the working set. Once there are same data arriving at a common blocks, we then construct a Phi node for the data merges. We continue this manner until the working set is empty. The **invoke** instruction in LLVM with unwind for exception handling brings us some problems, since the target register will not be available once there is exception happens. Thus, we perform a special checks for this case and replace that register as Undef.

3 Experiment Results

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

We first pass the original IR to **mem2reg** to generate the corresponding SSA form. The SSA IR is then optimized with **sccp** pass to remove available constant propositions and merely leave those destructive merges for our pass to run. We set the cut off of running our pass as 0.2 for each function, i.e. we ignore that destructive merge if the fitness is below that number. This would save lots of useless works introduced by duplicating blocks for those lack of interests merges (since the cost of blocks duplication is much larger than the benefits of increased constant propagation). We identify the destructive merge as at least one of data in the original phi representation is constant and we focus on the integer destructive merges. There is one final **sccp** for checking the removed instructions. The **Possible Inst Removal** is inferred from the influenced nodes with the symbolic CFG.

There are two benchmarks (along with our samples) are tested for our program. We found that there is no destructive merge with fitness over 0.2 shown in the **SPEC-BZIP**, so we drop this benchmark. Three passes are implemented. The **FuncPhiInfo** collects the general meta for the destructive merges. **FuncCFGSplitInfo** pass are used for generating the symbolic CFG and count the possible instructions removal due to the split are also implemented. The **ModuleSplitMerge** performs the actual merge splits with block duplication.

Table 1: Meta Info

	Add Blocks	*Add Instructions*	*Possible Insts Removal*
Sample 1	14	36	6
Sample 2	4	9	2
Sample 3	2	6	1
LULESH	238	2423	170

A summary for the second **sccp** and the removed instructions are shown in follow tables. The detailed IR and out file are available in the appended zip file. The generated meta fitness information, generated code and symbolic CFG are ain the representation of instructions (nodes in paper). We transform them to the name of basic block.

Table 2: Second SCCP Remove

	SCCP
Sample 1	24
Sample 2	10
Sample 3	7
LULESH	1886

Since there is no destructive merge with fitness high enough in **SPEC-BZIP** and the previous samples are trivial, we merely run our implementation on **LULESH**. The first **SCCP + SSA** is the base input with SSA and SCCP pass. It is used as the input for our implementation for the following two. The **Split** is the raw output of our implementation and the **Split + SCCP** is the output with one extra SCCP pass.

Table 3: Benchmark Run

	SCCP + SSA	*Split*	*Split + SCCP (final)*
LULESH	25(s)	23(s)	22(s)

References

[1] Aditya Thakur and R Govindarajan. Comprehensive path-sensitive data-flow analysis. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 55–63, 2008.