Bachelor of Technology Project-1 Report

Towards validation of RTL passes of the GCC compiler

Eashan Gupta (160050045)

Guides: Prof. Amitabha Sanyal,
Prof. Supratik Chakraborty

1 Introduction

Translation validation establishes the semantic-equivalence of a source and the target program obtained by its transformation. In [4], the authors propose efficient techniques to use translation validation to validate the optimization passes of a compiler. This project is a part of the master project to implement these techniques to validate the optimization passes of the GCC-4.7.2 compiler. As part of the team working on this, while much work has been done and is being done, this is the first attempt to implement translation validation for the Register Transfer Language (RTL) optimization passes of GCC.

The techniques proposed in the paper are based on the observation that an optimization pass of a compiler often leaves large part of the code unchanged. This is particularly observed for the RTL optimization passes. Based on this, we further simplify the algorithm to validate the passes by doing a *block-by-block analysis* using the Z3 SMT solver. This report also contains some details of the implementation and the various supports added.

2 Problem Statement

Problem 2.1. *Given programs P and Q, prove if P and Q are semantically equivalent.*

Problem 2.2. *Given a program transformation $O : \mathbb{P}_1 \to \mathbb{P}_2$, validate O i.e. prove*

$$P = O(P), \forall P \in \mathbb{P}_1$$

where $p_1 = p_2$ implies that p_1 is semantically equivalent to p_2.

Problem 2.3. *Given a program P and a transformation O, prove if P and $O(P)$ are semantically equivalent.*

The final goal of the project is to validate the GCC-4.7.2 compiler. Here we observe the compiler as a transformation applied to the input C code which gives the Assembly code. The transformation also includes various optimizations done by the compiler during the course. Validation of compiler here means to prove the output program is semantically equivalent to the corresponding input for all input programs. This can be summarised in Problem 2.2 with compiler as the transformation O.

We know that Problem 2.1 is undecidable. Following this we can also show that Problem 2.2 and Problem 2.3 are undecidable. Thus, we cannot prove that the input code to a compiler and the output from it are semantically equivalent for all input programs. Instead, we can derive techniques to validate a compiler upto some particular benchmarks to observe its performance. In this project, we implement the techniques with the goal of validating the GCC-4.7.2 compiler to pass the SPEC CPU benchmarks. In addition, in the case it does not pass the test, we will have found an unreported bug of the compiler. Note that from now on, when referring to validation, we mean to validate for the SPEC CPU benchmark only.

To validate the compiler, we break the process into validating each optimization pass of the compiler separately. Next, we validate each function of a program independently and thus if all functions of the

program are validated, we can say that the program is validated. The aim of this project is for each program from the SPEC CPU benchmark, to validate each RTL optimization pass. Thus,

$$O = O_1 \cdot O_2 \cdot \cdots \cdot O_n$$

where O is the compiler transformation, and O_i are the optimization passes.

2.1 Overview

We implement plugins before and after each pass to implement this in the GCC-4.7.2 compiler. We also use the Z3 SMT solver [3] for the same. The input program to the optimization pass is referred to as the source program and output program to the optimization pass as the target program.

As described in [4], we can represent a synchronized execution of the source and the target program using a joint graph. To validate the semantic equivalence of the given programs using such a structure, we need to check for logical relations and assertions at the appropriate synchronization points. These assertions are the "obligations" which we have to follow to check for semantic equivalence and are given as follows:

1. **Return Obligation:** The return values of the function of source and target must be equal to prove program semantic equivalence.
2. **Heap Obligation:** The changes to the memory heap should also be same i.e. assuming that initial heap structure provided to the two programs is equivalent, the final structure should also be equivalent.
3. **Function call obligation:** The heap and the arguments given to a function before the function call are required to be equivalent as the function may require these values for its instructions.

Checking for these synchronization points is sufficient to conclude the equivalence. Detailed applications of these are covered in the following sections.

Definition 2.1. *A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.*

3 Analysis on RTL Passes

To validate the compiler, we need to validate each of its optimization passes. In this project, we only focus on the RTL optimization passes. A brief overview of all the RTL passes used in GCC-4.7.2 are given in the Appendix. A more detailed description about the various passes can be found in [1].

We have done various analysis on the control flow graphs of the input and output programs of the RTL passes after running the SPEC CPU benchmark. This includes checking for the number of basic blocks in the programs as well as if the source and target CFGs are isomorphic. We also compress the graph by merging as many basic blocks as possible and remove the dead blocks. The detailed statistics can be found in the Appendix. We also check for difference in the codes line by line ignoring variable names- register names in the case of RTL code. Some observations from these analysis are:

1. There are a total of 26 RTL passes which we tested
2. There are a total of 7 RTL passes out of 26 for which we find that the number of basic blocks changes when going from source CFG to target CFG.
3. There are 3 RTL passes out of 26 which maintain the number of blocks in the CFG but the structure changes so that the source and the target CFGs are not isomorphic. These 3 RTL passes are a subset of the above 7 i.e the other 19 passes do not change the CFG structurally at all.
4. There are also cases where there is a total reduction in the number of basic blocks so it is not possible to bunch passes together to run block by block analysis on the final output of the set of passes.
5. It may be possible to run block by block analysis for the other 19 RTL passes as they maintain the number of blocks as well as isomorphism among them.

6. Statistics based on differences line by line:
   Total functions in the SPEC CPU benchmark: 608
   Note that some files in the benchmark do not run and may have required some more packages in GCC. Such cases have been ignored for the analysis in this project.
   Number of evaluations: $608 \times 18 = 10944$
   Number of (pass, CFG) pairs where the CFG does not change: 6997
   Rest number of pairs: 3947
   Note that for each pass we store the pre pass CFG, so the last pass is unable to be validated in current version. This is why we have only 608×18 *evaluations and not* 608×19.

4 Modelling RTL Statements

Register Transfer Language

Register Transfer Language is a low-level intermediate representation on which the last part of the compiler work is done. The language is inspired by Lisp lists and mainly uses expressions as objects. The other objects used are integers, wide integers (internal object), strings and vectors. Thus, the instructions in RTL are described pretty much one by one, in an algebraic form that describes what the instruction does. The most basic unit of an expression, as in the name Register Transfer Language, are registers. More detailed information about RTL syntax can be found in [2].

To validate the programs using the Z3 SMT solver, each statement in the RTL program needs to be emulated in the Z3 solver context to write appropriate assertions. Note that for the RTL passes in consideration, both the input and the output code is in Register Transfer Language. We now first describe the methods used to write assertions from different RTL statements and then describe the algorithm used in detail.

4.1 General RTL statements

General RTL statements are handled recursively. As RTL is inspired from the Lisp lists, each statement is an expression with a head operator and arguments. All arguments are handled recursively using the function `process_use` which returns a Z3 expression. These arguments are then operated over as described by the operator definition. Finally, each instruction generates a Z3 assertion in the context. e.g.

```
rtx statement = (plus:DI (reg:DI 61) (reg:DI 66))
process_use(statement) = process_use((reg:DI 61)) + process_use((reg:DI 66))
```

4.2 Single Static Assignment for Registers

The registers in RTL replace all variables used in the original C code. To emulate them in Z3, we treat them as variables and to differentiate between the source and target variables, we add a suffix of `src` or `dest` accordingly. In order to write assertions in the Z3 context, we use the single static assignment. We maintain a version number for each variable and update it each time the value is modified. The initial version number is assigned as 0. e.g.

```
rtx statement = (set (reg:DI 66)
                     (and:DI (reg:DI 61)
                          (reg:DI 66)))
process_use(statement) = (process_def((reg:DI 66)) =
                            process_use((and:DI (reg:DI 61)
                                                (reg:DI 66))))
process_def((reg:DI 66)) = r_66_src_1
process_use((and:DI (reg:DI 61) (reg:DI 66))) = process_use((reg:DI 61)) &&
                                                    process_use((reg:DI 66))
```

As we handle the instructions recursively, every update of a variable is done using the function `process_def` which processes the LHS. It also updates the version number of the LHS. Finally, processing `statement` returns the following Z3 assertion:

```
process_use(statement) = (r_66_src_1 = r_61_src_0 && r_66_src_0)
```

Here, the variable name is delimited by "_". "`r`" is a marker for a register, next the register number, `src` as source CFG and the last number is the version number.

4.3 Emulation of Heap as Memory Array

The memory in a program is mainly accessed due to dynamically allocated memory (pointers on heap) or global variables (stack). For any of these accesses, it changes the state of the program which we need to keep track of. For this, we use Z3 arrays, one for each data type. Every access to memory in RTL is emulated to an access to the corresponding data type array. e.g. For pointer access:

```
rtx statement = (mem:SI (reg/v/f:DI 73))
process_use(statement) = (select src_int_memory_0 r_73_src_0)
```

Here the pointer is stored in register `73`. The data type of the statement is given by `SI` - single integer. So, we refer to the array representing integer objects of the source CFG. The naming of the array is also done accordingly i.e `src` for source (`dest` for target) and `int` for the integer objects. Similarly we have `src_float_memory_X` and `src_double_memory_X` arrays for the corresponding data types.

To access global variables, we again use the same array and a similar process. e.g.

```
rtx statement = (mem/c:SI (symbol_ref:DI ("x")))
process_use(statement) = (select src_int_memory_0 x)
```

The global variables like in the above example, are represented using their names "`x`". We access the variable using the names directly from the array.

The last number in the array name is the version number of the array. We again use single static assignment for the arrays in case they are updated. e.g.

```
rtx statement = (set
                    (mem/c:SI (symbol_ref:DI ("x")))
                    (const_int 3 [0x3]))
process_use(statement) = (src_int_memory_1 = (store src_int_memory_0 x 3))
```

Here the global variable "`x`" is updated to 3 i.e. the C code has an instruction `x = 3;`.

The data types of the Z3 arrays are (`Y` for source or target, `X` for versioning):

```
Y_int_memory_X (Array Int (_ BitVec 32))
Y_float_memory_X (Array Int (_ FloatingPoint 8 24))
Y_double_memory_X (Array Int (_ FloatingPoint 11 53))
```

The data types are defined as per GCC standards and may be changed according to the hardware version. The index data type is `Int` and value type in integer array is `BitVec 32`. This is done to manage the bits of the integers.

Using the above structures, we emulate a complete memory heap and stack, which can be tracked and used for assertions in validation.

Note that the array support for other data types like float and double do not completely work. It still gives errors in some cases of floats. The array support for integers is complete, so following the same ideas, the other types can also be completed.

4.4 Pointers

As described above, we use the arrays to emulate the heap structure of GCC. For pointers with higher order (e.g. `int** x;`), the accesses are only a recursive call on the memory array which is taken care of by RTL itself. But using this recursion requires special care for types as the index type of the arrays is `Int` while the value type is a `BitVec 32`.

4.5 Type Casting

The data types used in RTL are defined using the size of the object. As included in the above examples, `SI` for single integer, `DI` for double, `HI` for half integer(short) etc. More details for different modes of data are given in [2, Machine Modes]. RTL defines various operator heads like `sign_extend`, `zero_extend` etc. to take care of type casting. We use these operators and check their types before each call and change the types accordingly to manage data types. e.g.

C code:

```
int b;
short t;
t=3;
b=t;
```

RTL Code:

```
(set (reg/v:HI 60 [t])
    (const_int 3))
(set (reg:SI 68 [b])
    (sign_extend:SI (reg/v:HI 60 [t])))
```

Z3 assertions:

```
(declare-fun r_60_src_0 () (_ BitVec 16))
(declare-fun r_68_src_0 () Int)
(declare-fun r_60_src_1 () (_ BitVec 16))
(declare-fun r_68_src_1 () Int)
(r_60_src_1 = 3)
(r_68_src_1 = (ToInt (r_60_src_1)))
```

Here, from the original C code, the corresponding registers- `60` for `t` and `68` for `b`- are allocated in RTL. Also, the operator `sign_extend` is used to extend the bytes from `short` to `int`. In the final Z3 assertions of the above example, we manage the types using Z3's types casting.

Following this, we can use the Z3 expressions returned from the function `process_use`, check the types they should belong to using APIs in GCC, then type cast them in Z3 too. Additionally, as in the example in Section 4.1, the operands of `+` should be individually type checked.

Note that type casting has not been completely implemented yet and still requires work. But the above idea is clear and feasible and can be used for the implementation.

5 Block-by-Block Validation

Following the analysis in Section 3, we observe that majority RTL passes don't change the programs much. Additionally, as many as 19 RTL passes out of 26 do not change the structure of the graph. Thus, we can simplify the algorithm from validating each function independently to validating each basic block. This method is defined as the basic block-by-block analysis.

5.1 Validation Technique

Given source and target control-flow graphs, we first compress the two graphs independently to merge as many basic blocks as we can and remove the dead blocks. Then, the two CFGs are matched structurally to map the basic blocks from the source CFG to the target CFG. The corresponding basic blocks from both graphs are merged together to form a new control-flow graph. Each basic block in the new graph contains instructions from the corresponding source and target basic blocks. Note that we maintain the information to differentiate instruction statements from both initial CFGs. This new graph is now referred to as the merged CFG.

We also use equality propagation to propagate equality assertions between blocks. These equality assertions are propagated as sets of variables which are equal, also known as *equality sets*.

5.1.1 Forward Analysis

We use the standard Forward- fixed point- Analysis on the merged CFG to validate it. The value propagated between blocks of the merged CFG are the equality sets. Initially the entry at each block is kept as the top value. Start with the entry block of the CFG and analyse it. The equality sets generated at the end of the block are propagated to the successors which are queued to be analysed next. This goes on till there are no more changes to the values- equality sets.

5.1.2 Assertions at the entry of block

To validate a block, we assume that the initial memory stacks and heaps of the source and target program are equivalent. This assertion is added to the Z3 context.

```
(src_int_memory_0 = dest_int_memory_0)
```

This is also done for other data types.

Additionally, we use the equality sets from the predecessor blocks to set the initial assertions. In the case of multiple predecessors, we take an intersection of the equality sets from those propagated by the siblings. e.g.

```
(r_60_src_0 = r_61_dest_0)
```

if it is known that `r_60` of source is equal to `r_61` of target at the entry of a block.

5.1.3 Analyse basic block

In a basic block of the merged CFGs, all instructions are converted to assertions and added to the Z3 context as described in Section 4.

5.1.4 Equality Propagation

At the end of each block, equalities are generated using all variables pairwise. Then, by generating Z3 models for a case where they fail, we eliminate equalities. e.g. if we have live variables `p_src`, `t_src`, `p_dest` and `t_dest`, we put them all in a single equality set. We then check the assertion:

```
not ((p_src == t_src) && (t_src == p_dest) && (p_dest == t_dest))
```

i.e. atleast one of the variable pairs is not equal. We generate a satisfying model and in the case we get one, we can remove the unequal variables and create new equality sets based on the values from the model. Generating similar assertions for each of the sets, reducing the size of equality sets accordingly and repeating the process till there are no more changes, we get the final equality sets. These equality sets are then propagated.

5.2 Obligations and Correctness

Finally, at the exit of the block, we check for the following obligations:

5.2.1 Return Obligation

While analysing each block, we also maintain a check if the block has a return statement. Note that as it is a basic block, it may have atmost one return instruction. If it does, at the exit of the block, we check for equality between the registers of both the source and target that contain the return value. In the case that these are not equal, either we have found an unreported bug in the GCC compiler or we have failed in validation due to some implementation error which can be checked manually.

5.2.2 Heap Obligation

At the end of the block, we check for the equality of the final versions of the memory arrays of corresponding types from source and target. If this is true, we can continue assuming in the next block as well that the initial heaps were equal. Propagating it, we can show that heap obligation is satisfied throughout the program.

But this method of heap obligation is based on the assumption that the global variables and dynamically allocated memory are not eliminated or modified by the working of an optimization. Though the dynamic memory may be deleted in the program itself. e.g.

```
int x=5;
int f(){
    int a=2;
    x=6+a;
    return x+1;
}
```

No optimization pass should eliminate the use of `x`, as it is a global variable, though it may eliminate `a`, as it is locally defined.

5.2.3 Function Call Obligation

For this obligation, whenever there is a function call, we store the version number of memory arrays and also the arguments to the function. Then at the end of the basic block, these are tested to be equal correspondingly between the source and target.

Note that this has not been implemented yet and we skip function call instructions right now. The function calls require to fetch the function arguments which are not easily available in RTL.

Other Obligations

Note that there should be another obligation here to completely validate the program. This would be to validate the branches or to prove the mapping from source CFG basic blocks to target CFG basic blocks. Here, the program counter `pc` variable of the CFGs and the labels allocated in the RTL code could be helpful in adding useful assertions to validate the mapping. This obligation has not been taken care of yet in the current implementation.

5.3 Assumptions

Following are the assumptions for using the above block-by-block analysis as a validation technique for programs:

1. This technique only works for cases where the source and the target CFGs are isomorphic or do not change structurally.
2. It is assumed that the optimization pass does not eliminate the use of a global variable or modify them. This is also helpful as we emulate these in the memory array as shown in Section 4.3.

6 GCC Macros

In several cases, there are variables that are not defined. In such cases, the validation generally fails. e.g. C Code

```
int f(){
    int a, b;
```

```
    a=b+3;
    return a;
}
```

In this case, the previous validation technique will fail as `b` is not defined and the Z3 solver will find cases where the return value i.e. `a` may not be equal for the source and target CFGs. In such cases, we get the registers which are undefined using a liveness analysis on the target CFG only. This is a Backward- fixed point- analysis. Then we use GCC macros to get the registers which represented them in the source CFG. Using this mapping, we create additional equalities at the start of the block-by-block validation. e.g.

```
(r_63_src_0 = r_62_dest_0)
```

Here `r_62` is undefined in target CFG and it represents `b` from C in RTL. Using GCC macros we find that `r_63` represented the same variable `b` in the pre-pass (source) CFG. We can thus equate the two to get an additional equality and finally validate the above programs. This is a convention followed by us as there is no clear way of handling undefined variables in GCC.

This method is also used currently to get function arguments in RTL as we have not found a more direct way. The function arguments found are equated using this as they have to be equal before validation.

7 Implementation

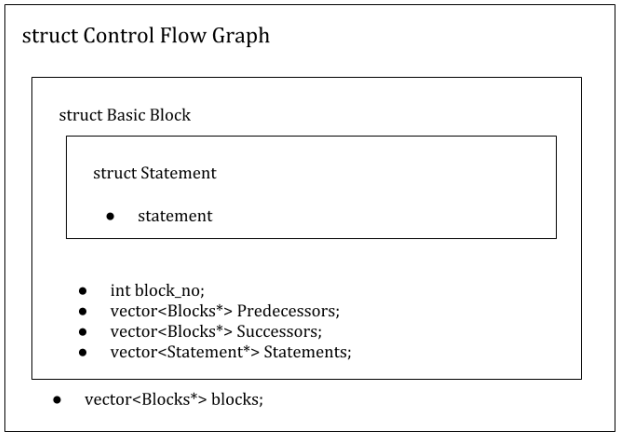

Figure 1: Code structure used

We have used the data structures as shown in Figure 1 to store and represent the CFGs in the plugins of GCC. The structures are all templated to suit any types of statments- RTL and Gimple. The code for various analysis are also similarly templated so that they can be used on the given graph structure. The skeleton design of templates was done by Mr. Ajeesh Kumar. I have implemented the specializations for these templates to support the RTL optimization passes. The code is further divided into a package called *Distillery* where the common templated code and framework is added. The specializations can be added by defining the methods of the framework. I have also contributed to the package *Distillery* by further refininng the design to support RTL optimization passes. Further, various methods in the classes were required to aid in the analysis out of which some have been implemented and refined by me.

8 Performance and Future Work

After running the above validation technique for the SPEC CPU benchmark, we get the following statistics.

1. Total number of running functions in the benchmark: 608

2. Number of RTL passes which maintain isomorphism: 18
   Unable to validate the last pass so 18 and not 19.

3. Number of cases to be validated = Number of (RTL pass, function) pairs = $608 \times 18 = 10944$

4. Number of (pass, function) pairs validated using the above techniques: 114

Following the above statistics, we see that we are only able to validate 114 runs out of a total of 10944 runs. This is because RTL has many instructions which have not all been implemented in our framework. Additionally, the data types and type casting have all not been supported yet. Still, from the statistics we can see that some simpler functions of the benchmark which rely on integers and its operations and void of function calls are being validated.

Currently, simple statements using integer variables have been supported and the structure is in place. The next work in completing this project would be to support the various data types, type casting and function calls and its obligations. Also, the various other instructions in RTL are also to be supported.

In the implementation, we have also added options to report the instructions at which our validator fails, in the case that the instruction is not currently supported. We have also added code scripts to run all the SPEC CPU benchmarks. As the benchmarks are run, the code reports all the instructions which have yet to be supported and then can be added to the implementation. Using this functionality and the description of the RTL instruction from [2], the various operators and instructions of RTL can be supported. As we know that the most basic objects used in RTL are expressions and registers, support for structures and other data structures should also not require different support structures and follow the current design.

From the above analysis and analysis in Section 3, we can see that checking using a simple `diff` command is much easier and shows better results as it clears 6997 cases as compared to 114 with the current validation technique implementation. So, we can add `diff` as a first level check in the implementation. Also, the current validation technique only works for 19 out of the 26 RTL passes which maintain isomorphism so the rest are yet to be validated and may require the complete implementation of techniques as described in [4] using the AJTGs. Further, the problem of translation validation is an open and undecidable problem so there can be many other algorithms which are more efficient and this requires continuous study and exploration.

References

[1] Gcc passes documentation:
`https://gcc.gnu.org/onlinedocs/gccint/Passes.html`.

[2] Rtl documentation:
`https://gcc.gnu.org/onlinedocs/gcc-4.6.4/gccint/RTL.html`.

[3] Z3 theorem prover:
`https://github.com/Z3Prover/z3`.

[4] A. Sanyal A. Sharma, N. George and S. Chakraborty. An abstraction based technique for translation validation. IIT Bombay.

Appendix

Analysis on RTL Passes

After running the SEC CPU benchmark for the RTL optimization passes, I was only able to run a total of 608 functions. Others might not be running due to them requiring some additional packages. These cases have been ignored currently. After running these 608 functions for the 26 passes, we analyse them for each RTL pass by counting the change in number of blocks, if the input and output CFGs are isomorphic and if the CFGs change at all. Following is a list of the RTL optimization passes in GCC. It also contains a brief description of each and details can be found on [1]. Further the statistics after the analysis are also include:

1. **sibling**:
   Pass number 151. Sibling call optimizations. Does not change the CFGs structurally i.e. maintains isomorphism. Only changes 1 CFG out of 608.

2. **rtl_eh**:
   Pass number 152. Does not change the CFGs structurally i.e. maintains isomorphism. Does not change any CFG out of the 608.

3. **initvals**:
   Pass number 153. Does not change the CFGs structurally i.e. maintains isomorphism. Does not change any CFG out of the 608.

4. **unshare**:
   Pass number 154. Does not change the CFGs structurally i.e. maintains isomorphism. Does not change any CFG out of the 608.

5. **vregs**:
   Pass number 155. Converts virtual registers to hard registers. Does not change the CFGs structurally i.e. maintains isomorphism. Changes all 608 CFGs.

6. **into_cfglayout**:
   Pass number 156. Merges blocks, removes dead blocks and merges blocks with unnecessary conditions. Changes the CFGs structurally by changing number of basic blocks. An observation that in all cases, the number of basic blocks decreases.

7. **jump**:
   Pass number 157. Solves simple if_then_else statements, only in old gcc compilers and patched in newer versions. Changes the CFGs structurally by changing number of basic blocks. There are also cases where number of blocks was same but the graphs were not isomorphic.

8. **reginfo**:
   Pass number 169. Changes the CFGs structurally by changing number of basic blocks. There are also cases where number of blocks was same but the graphs were not isomorphic.

9. **outof_cfglayout**:
   Pass number 189. Reverses the merging of into_cfglayout only in direct cases. Changes the CFGs structurally by changing number of basic blocks. An observation that in all cases, the number of basic blocks increases.

10. **split1**:
    Pass number 190. Instruction splitting. Does not change the CFGs structurally i.e. maintains isomorphism. Changes 61 CFG out of 608.

11. **modesw**:
    Pass number 193. Does not change the CFGs structurally i.e. maintains isomorphism. Does not change any CFG out of the 608.

12. **asmcons**:
    Pass number 194. Fixing rtl statements that have unsatisfied in/out constraints. Does not change the CFGs structurally i.e. maintains isomorphism. Changes all 608 CFGs.

13. **ira**:
    Pass number 197. Integrated Register Allocator- creates pseudos to rename. Changes the CFGs structurally by changing number of basic blocks. An observation that in all cases, the number of basic blocks increases.

14. **reload**:
    Pass number 198. Renumbers pseudo registers with the hardware registers numbers they were allocated. Changes the CFGs structurally by changing number of basic blocks. An observation that in all cases, the number of basic blocks decreases.

15. **split2**:
    Pass number 201. Instruction splitting. Does not change the CFGs structurally i.e. maintains isomorphism. Changes all 608 CFGs.

16. **pro_and_epilogue**:
    Pass number 205. Changes the CFGs structurally by changing number of basic blocks. There are also cases where number of blocks was same but the graphs were not isomorphic.

17. **stack**:
    Pass number 218. Does not change the CFGs structurally i.e. maintains isomorphism. Does not change any CFG out of the 608.

18. **alignments**:
    Pass number 219. Computes branch alignments. Does not change the CFGs structurally i.e. maintains isomorphism. Changes all 608 CFGs.

19. **mach**:
    Pass number 222. Performing the machine dependent reorganization pass, if that pass exists. Does not change the CFGs structurally i.e. maintains isomorphism. Changes all 608 CFGs.

20. **barriers**:
    Pass number 223. cleaning up the barrier instructions. Does not change the CFGs structurally i.e. maintains isomorphism. Only changes 2 CFG out of 608.

21. **eh_ranges**:
    Pass number 226. Does not change the CFGs structurally i.e. maintains isomorphism. Does not change any CFG out of the 608.

22. **shorten**:
    Pass number 227. Shortens branches. Does not change the CFGs structurally i.e. maintains isomorphism. Changes 71 CFG out of 608.

23. **nothrow**:
    Pass number 228. Does not change the CFGs structurally i.e. maintains isomorphism. Does not change any CFG out of the 608.

24. **dwarf2**:
    Pass number 229. Does not change the CFGs structurally i.e. maintains isomorphism. Changes 386 CFG out of 608.

25. **final**:
    Pass number 230. Does not change the CFGs structurally i.e. maintains isomorphism. Changes 386 CFG out of 608.

26. **dfinish**:
    Pass number 231. Not evaluated.

Other than the above RTL passes, there are two other passes which we have not tested- **expand** (the first RTL pass which converts Gimple to RTL) and **dfinit** (the last RTL pass). In the above RTL passes' list, the red coloured passes are ones which change the structure of CFGs in some cases- be it number of basic blocks or the isomorphism of graphs. The other passes have CFGs which maintain their structures when comparing input and the corresponding output for the SPEC CPU benchmark. The blue coloured passes may make some changes to some of the CFGs. The rest don't change anything at all atleast for the SPEC CPU benchmark.

Psuedo Code

Merge blocks and remove dead blocks

```
merge_blocks(V, E):
    merged={}
    remove={}
    result_E
    map m1                \\ vertex -> {}
    for i in E:
        if |E[i]|=1 and (number of predecessors of next node E[i][0]=1):
            if i not in merged:
                put E[i][0] in merged
                put E[i][0] in m1[i]
            else:
                put i in remove
                put E[i][0] in merged
                find j such that i is in m1[j]
                put E[i][0] in m1[j]
    for i in V - remove:
        if i not in merged:
            result_E[i]=E[i]
        else:
            find j such that i is in m1[j]
            result_E[j]=E[i]
    return result_E
```

Check Isomorphism

```
check_isomorphic(E1, E2):
    visit1[n]=-1, visit2[n]=-1
    return helper(E1, E2, visit1, visit2, 0, 0)

helper(E1, E2, visit1, visit2, n1, n2):
    if(visit1[st1]*visit2[st2]<0)
        return False
    if(visit1[st1]!=-1)
        return True
    visit1[st1]=st2
    visit2[st2]=st1
    if(|E1[st1]|!=|E2[st2]|)
        return False
    for i1 in E1[st1]:
        ch=0
```

```
    for i2 in E2[st2]:
        k, temp_vis1, temp_vis2 = helper(E1, E2, visit1, visit2,
                                         E1[st1][i1], E2[st2][i2])
        if(k):
            ch=1;
            visit1=temp_vis1
            visit2=temp_vis2
            break;
    if(!ch):
        return False
return (True, visit1, visit2)
```