DSnP Final Project：Functionally Reduced And-Inverter graph

Name：蔡松達

School ID：B05901040

Email address：B05901040@ntu.edu.tw

Ⅰ Design of Data Structure

1-1. Inheritance and Polymorphism of class CirGate

To apply different functionalities according to characteristics of different types of gates, I separate all the gates into five types and make them inherit class CirGate.

Five types of gates are: PI gates, PO gates, AIG gates, CONST gates, and UNDEF gates.

1-2. Data Structure in CirGate

All the gate object links to pointers of its fanin gates, fanouts gates, and stores its FEC group information (after simulation has done). If link between two gates has an inverter, add 1 to the corresponding pointer.

1-3. Four Gate Lists in class CirMgr

(1) Total_gate_IDList (vector): Stores gates' variables in sequence of they defined in aag file, including CONST gate, PI gates, PO gates and AIG gates. Only UNDEF gates are not included in this list.

(2) Undefined_gateIDList (vector): Stores variables of UNDEF gates. UNDEF gates are those gates with fanouts but not really defined in aag file.

(3) Gates_map (Unordered_map): Stores pointers of all gates. In this hash table, hash key is the variable of each gate, and hash value is the pointer of the gate.

(4) DFS_List (vector): Do depth first search from all the POs and record variables of gates along the path. Order of the list should follow the rules of DFS.

To access a gate, provides its variable and search Gates_map to get its pointer.

1-4. FEC_group in class CirMgr

Use two-dimension vector to store functionally equivalent candidate (FEC) information. An element in the first dimension represents a FEC group. Second dimension stores pointers of the gates which has the same simulation value so far.

1-5. Fail_patterns in class CirMgr

Some of the FEC pairs are not functionally equivalent. When we do fraig operation in this case, SATsolver would return result "SAT" and give the PI simulation patterns that lead to different simulation value between FEC pairs, and Fail_patterns (vector) stores this kind of patterns.

1-6 Hash table used in the program

Adopt C++ standard class "unordered_map", and set hash function to be:

$$\frac{\text{summation of pointers of two fanins in type size\_t}}{59}$$

in strash operation. Other hash table used in my program adopts default hash function.

Ⅱ Design of Algorithm

2-1. Sweep, Optimize, and Structure Hashing

Follow the instructions in class to implement these three operations. First construct two basic functions, which are used to update the connection with its own fanins and fanouts. If an AIG is to be removed, just remove both fanin gates' fanout which points to itself. Next, send pointer of the gate used to replace itself to all its fanout gates. Fanout gates should update its fanin from the replacing gate to the designated gate. Finally, delete this AIG gate and update this information in Gates_map and gateIDList.

Note that it's important to handle the inverting or noninverting phase. If the replacing AIG's fanouts has the same phase with the designated gate, just make them link together in noninverting phase. Otherwise, they should link in inverting phase.

These three operations follow the following process shown in Fig. 1, details are taught in class and detailed removing process is discussed above.

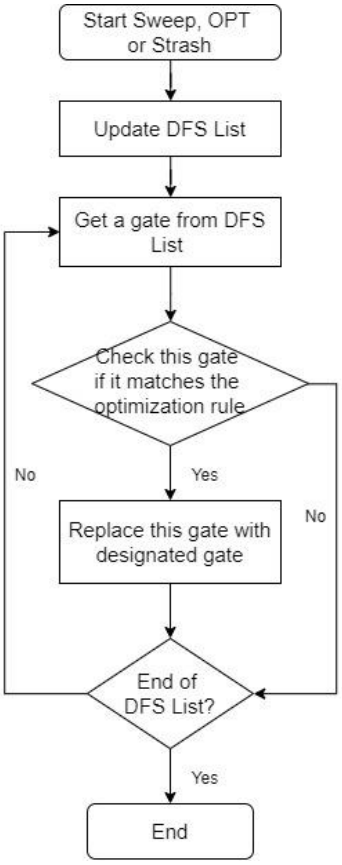

Fig. 1 Three optimization operation Flow

2-2. Simulation

(1) *Read or randomly generate patterns*

Fig. 2 shows the flow of simulation. First, read 64 patterns from the given pattern file or randomly generate 64 PI patterns. Since random function provided in C++ standard only ranges from 0 to 2^{32}-1, which means that we can only generate 32 bits. To achieve the goal of generating 64 bits, use random function to generate 2 numbers. Make the first generated number shift left 32 bits, and add the resulting number to the second generated number. Finally we can get a randomly generated number with 64 bits. Pack each 64 bits into C++ standard class "bitset<64>" as its simulation value, and store the corresponding simulation value in each PI.

(2) *Parallel Simulation*

Next step is to do parallel simulation. Starting from each PI, sends its simulation value to all of its fanout gates. Note that if an inverter occurs, simulation value sending to fanout should flip (1→0, 0→1).

If the fanout gate is AIG, check if this AIG gate have been visited in this simulation process. If this AIG gate has been visited, this gate must have gotten one value from one of its fanins. Adding the information of another fanin, this AIG gate can get both of its input value and do AND operation on both values. The resulting value would stores in the gate as its simulation value and sends to all its fanouts. However, if this AIG gate hasn't been visited, stores this value temporarily until another fanin gives it input value.

If the fanout gate is PO, stores the value as its simulation value.

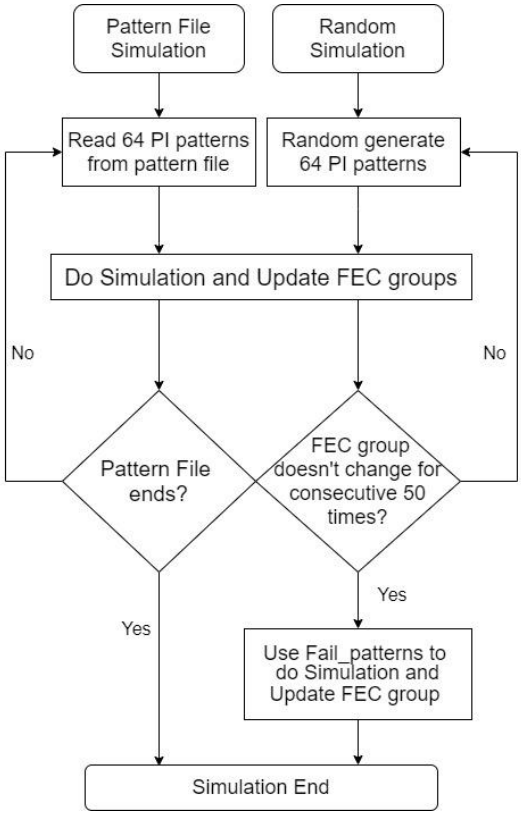

Fig. 2 Simulation Flow

(3) *Update FEC groups for the first time*

All gates including PIs, POs and AIGs should get a simulation value after simulation. Next step is to update FEC groups according to simulation value of each gate. All AIG gates and const 0 gate including its noninverting and inverting phase are in the same group initially,

where inverting phase refers to the inverse of the simulation value (or doing NOT operation). Keep this group in increasing order of variable of each gate, with noninverting gates putting before inverting gates. Construct a hash table, with simulation value as hash key and vector of pointers as hash value. After "first" 64 bits parallel simulation has done, check each gate in this initial FEC group if there's the same key as its own simulation value. If there's a group with the same simulation value, add itself into this group. Otherwise, insert a new element to this hash table with its simulation value as hash key, and also add itself into this group.

After checking all the gates, there would be many new FEC groups with both noninverting and inverting phase. To reduce memory usage and comparison, remove groups with the first gate is of inverting phase since noninverting groups could be representative of them. The remaining groups in hash table would be exactly half of original hash table. Note that we should check if some of the groups have only one gate. Those groups should also be removed since those gates can't form any FEC pairs with others. After doing the checking operation above, remove the first group from FEC group and put those remaining groups in hash table into FEC_group in class CirMgr.

(4) *Simulation again and update FEC groups afterwards*

After doing 64-bits simulation and updating FEC group for the first time, pack next 64 bits patterns and do simulation again. In the follow-up simulations, check all the existing FEC groups separately since there must be no FEC pairs between different FEC groups.

Simulation value	Gates in FEC group
XXXXXXXX (not yet simulated)	0 !0 11 !11 12 !12 13 !13 14 !14 15 !15 16 !16 17 !17 18 !18 19 !19 20 !20

(a) Before Simulation process

Old FEC groups	1st Simulation value	Gates in new FEC group
0 !0 11 !11 12 !12 13 !13 14 !14 15 !15 16 !16 17 !17 18 !18 19 !19 20 !20	00000000	0 11 12 !15 !20
	~~11111111~~	~~!0 !11 !12 15 20~~ (inverting group)
	~~01010101~~	~~13~~ (only one gate)
	~~10101010~~	~~!13~~ (inverting group, only one gate)
	11110001	14 18 !19
	~~00001110~~	~~!14 !18 19~~ (inverting group)
	11001101	15 !16
	~~00110010~~	~~!15 16~~ (inverting group)

(b) After First Simulation

Old FEC groups	2nd simulation value	Gates in new FEC group
0 11 12 !15 !20	00000000	0 11
	00011000	!15 20
	~~00111000~~	~~12~~ (only one gate)
14 18 !19	10101111	14 !19
	~~11101110~~	~~18~~ (only one gate)
15 !16	11000100	15 !16

(c) After Second Simulation

Table. 1 Example of updating FEC groups. To simplify the task, I use 8 bits parallel simulation as an example. (a) Before Simulation process: Put all AIG gates and CONST 0 gate with noninverting and inverting phase in the same group. (b) After First Simulation: Erase groups with leading inverting terms and with only one gate. (c) After Second Simulation: Erase groups with only one gate, and all follow-up simulations would update FEC groups with the same manner.

For each existing (old) FEC group, construct a hash table to separate all the gates into new groups by their simulation value. After an old FEC group has been checked, check all the new groups in hash table, and remove groups with only one gate in it. Construct a temporary vector to store those new groups and continue to check other old FEC groups until all old FEC groups have been checked. Finally replace all the old FEC groups with all the new groups.

(5) *Set limit for random simulation*

Doing the above operations repeatedly until all of the patterns in the pattern file has been simulated. In random simulation, check each parallel simulation if it leads to changes in FEC groups. Random simulation stops when there's no change in FEC groups for 50 times consecutively.

(6) *Use fail patterns to do random simulation*

If "FRAIG" operation has been called previously, it may record some of the patterns that are used to prove FEC pairs. Random simulation would use those patterns to do simulation and update FEC groups after "50 times limit" has reached. Table. 1 shows an example of constructing and updating FEC groups. Assume there are 30 gates in this circuit, and variable 11~20 are AIG gates.

2-3. FRAIG

Fig. 3 shows the flow of FRAIG operation. Use SAT solver to check each FEC pair if they are truly functionally equivalent. Since FEC group of CONST 0 gate may be large in a large circuit, proving all possible pairwise FEC pairs spends much time. To make CONST 0 gate dominates all the other possible FECs, prove CONST 0 gate with all the other gates in the same FEC group initially. If a gate is functionally equivalent to CONST 0 gate, remove this gate and add all its fanouts to CONST 0 gate (with inverting or noninverting phase), which is the same as what "strash" operation do. In addition, store all patterns that are used to prove inequivalent between CONST 0 gate and FECs. After proving all possible pairs related to CONST 0 gate, directly do simulation using those fail patterns to reduce FEC pairs. Those patterns can also be used in simulation afterwards.

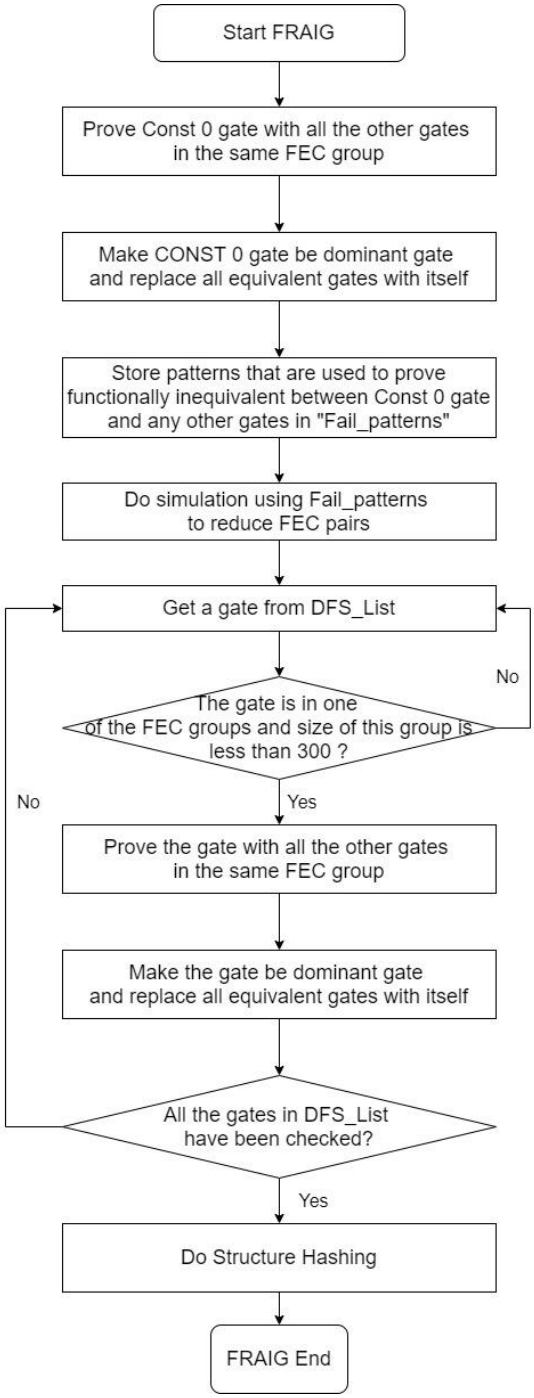

Fig. 3 FRAIG Flow

Next, go through DFS list and check each gate if it's in FEC group. Prove all possible FECs with itself and if they are proven to be functionally equivalent, replace those FECs with

itself (make itself be dominant gate). Remove gates from FEC group if they have been replaced or proven with all possible FECs. Note that to reduce proving effort, skip those FEC groups with over 300 gates in it.

After checking all gates in DFS List, call "strash" to further reduce circuit structure.

Ⅲ Result and analysis

3-1. Results of experiments

(1) *Sweep, Optimize, and Structure Hashing*: Results are all the same as reference program. Runtime and memory usage in the above three operations are also close to reference program.

(2) *Simulation*: Results of file simulation and the corresponding output pattern file are all the same as reference program. Table. 2 shows the experimental results of some aag files using dofile "do.fsim" and pattern files in "test.fraig" folder. My average simulation time is 1.43 times of reference program, and memory usage is 1.75 times of reference program. Memory usage performs worse than runtime in simulation, but all cases could get the correct answer.

(3) *FRAIG:* Results of random simulation and FRAIG operation are all correct using "cirmiter" operation to compare with results of reference program. Table. 3 shows the experimental results of some aag files using dofile "do.fraig". My average runtime is about 1.1 times of reference program, and memory usage is about 1.2 times of reference program. All cases can get correct results, and my average remaining AIGs is a little bit less than reference program. Results show that there are less remaining AIGs in my implementation, which means my program would optimize more than reference program. However, runtime performs poor in some cases, especially in sim12.aag.

aag files / results	My sim runtime (sec)	Ref sim runtime (sec)	My memory usage (MB)	Ref memory usage (MB)
sim09	0.05	0.04	1.586	1.297
sim10	0.02	0.02	0.3672	0.4258
sim12	0.6	0.38	4.055	2.691
sim13	14.46	10.16	34.51	19.07
sim14	0.02	0.01	0.7852	0.3398
sim15	0.06	0.04	0.7891	0.3164
Average	2.54	1.78	7.02	4.02
Normal-ize by Ref	1.43	1.00	1.75	1.00

Table. 2 Comparison of experiment results of my program with reference program in file simulation.

aag files / results	My FRAIG runtime (sec)	Ref FRAIG runtime (sec)	My memory usage (MB)	Ref memory usage (MB)	My remaining AIGs	Ref remaining AIGs
sim07	12.67	10.87	7.496	7.551	1175	1175
sim09	0.06	0.09	1.671	1.531	1003	1003
sim10	0.02	0.03	0.125	0.082	0	0
sim12	**10.13**	**3.51**	9.734	7.125	0	0
sim13	93.41	91.13	57.1	46.33	77261	77313
C499	0.06	0.06	0.7773	0.6719	87	109
C499_r	0.11	0.03	0.4648	0.7656	263	274
C880	0.02	0.02	0.4922	0.4844	314	314
C1355	0.02	0.04	0.4727	0.4375	431	439
C1908	0.03	0.05	0.6992	0.8203	115	121
C3540	0.06	0.05	1.328	0.9219	242	242
C5315	0.08	0.11	1.691	1.469	1003	1003
C6288	0.04	0.04	1.906	1.602	2291	2291
C7552	0.24	0.13	2.047	1.746	498	498
Average	8.3536	7.5829	6.1432	5.1098	6048.79	6055.86
Normalize by Ref	1.1016	1.0000	1.2022	1.0000	0.9988	1.0000

Table. 3 Comparison of experimental results of my program with reference program in random simulation and FRAIG operation. "CirFraig" operation is called in dofile *once* only.

3-2. Performance Analysis

(1) *Simulation*

I tried two different ways to implement the process of updating FEC group. Originally, I try to keep those FEC groups that don't change in the simulation. Just add all the new FEC groups behind the old FEC group. However, this way is more complex and runtime is about 1.4 times longer than the algorithm described in Section 2-2.

All of the aag files in "test.fraig" can get correct result in file simulation and pattern simulation. In small circuits, runtime and memory usage are close to reference program. However, both perform poor in sim13.aag, indicating memory usage might play an important role in big circuit.

(2) *FRAIG*

To further reduce the whole circuit, make CONST 0 be dominant gate is a good choice since trivial optimization would help remove or simplify gates related to CONST 0. Also proving from PI to PO helps reduce runtime since proving gates closer to PIs needs shorter path to reach than those closer to POs. This would reduce the possibility to prove gates closer to POs and reduce proving effort.

In Section 2-3, I store fail patterns only when proving CONST 0 with other gates since proving FEC group of CONST 0 gates can be more difficult than other groups. Those patterns would be store in CirMgr and use in random simulation next time. This is to balance between memory usage and runtime. Storing all fail patterns may cause large memory usage, while discarding them would cause long runtime in random simulation next time. So I decide to store part of fail patterns only.

During the process of FRAIG, doing simulation using fail patterns can *greatly reduce approximately 60% of runtime*. Since doing simulation is much easier and faster than doing FRAIG operation, use fail patterns to differentiate remaining FEC pairs is a good choice.

3-3. Bottleneck

Since this project takes much time to do, optimizing operations could be quite difficult in a few days. But there are some of the strategies that may or may not improve the performance:

(1) Apply better proving sequence

Runtime of FRAIG in sim12.aag performs much worse than other circuits. One of the reasons is that all of the AIGs could be simplified and removed in this circuit and its size is quite large. There may be some better proving sequence but I don't have enough time to implement. Also, random simulation limit performs poor in this case. It may be another reason that leads to poor result.

(2) Set proving limit in FRAIG: Since I don't set any proving limit in my program, it may lead to long runtime in FRAIG operation. I focus on optimizing circuit to make its size as small as possible.

3-4. Summary

Runtime of my program is about 3 to 4 times longer than reference program, and memory usage is about 1 to 2 times larger than reference program. In this project, I learned how to construct a circuit and handle the connection between gates. Applying parallel simulation and SAT solver in this project is also a good opportunity to check what I have learned in this semester. Using STL such as unordered_map and vector, and developing correct algorithms to solve complex problems are the most important things I learned in this project.