資料結構與程式設計

Functionally Reduced And-Inverter Graph (FRAIG)

電機二
B05901020 郭彥廷
E-mail: b05901020@ntu.edu.tw

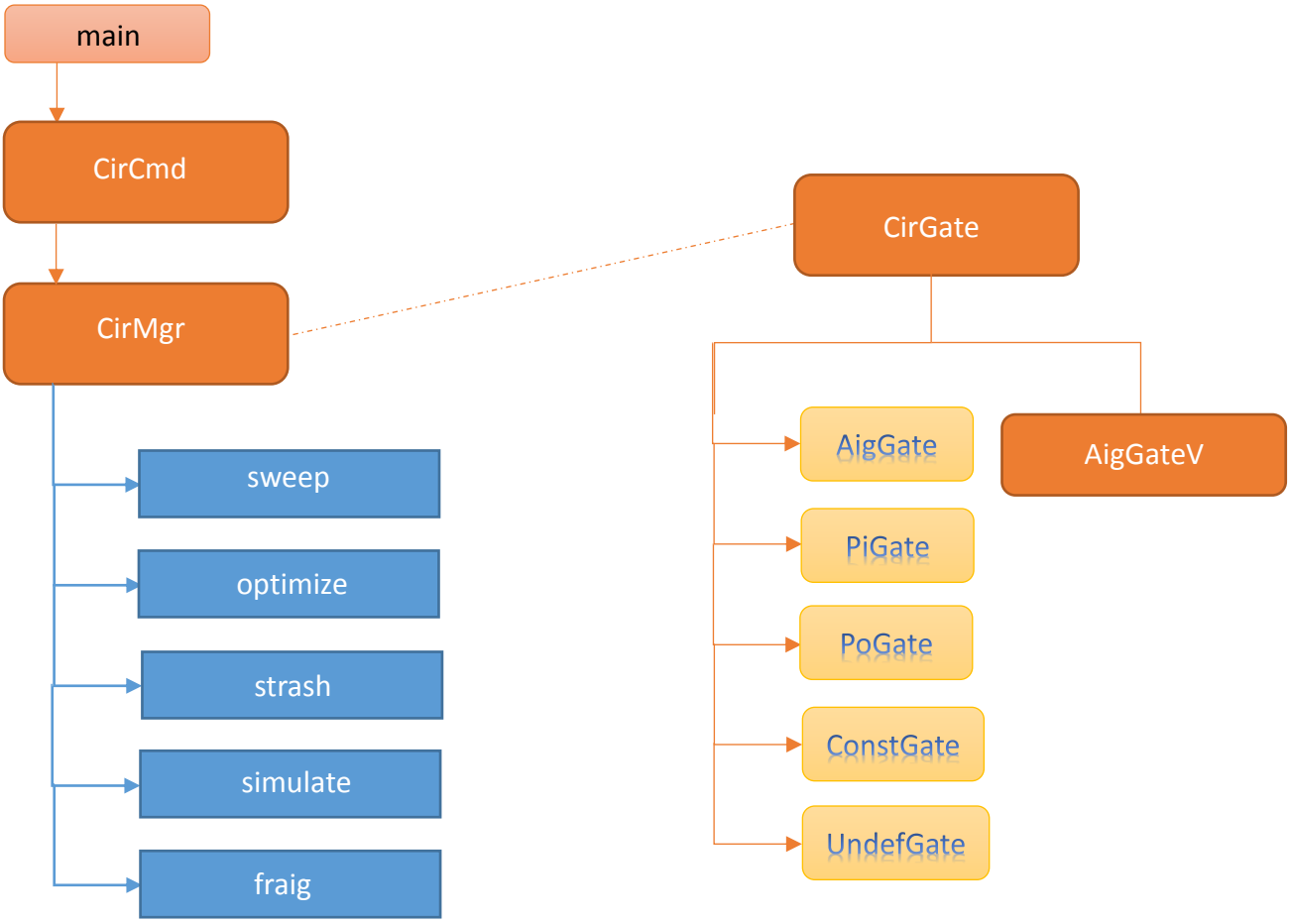
Abstract

In this project, we implement a special circuit representation, FRAIG (Functionally Reduced And-Inverter Graph), from a circuit description file. Then we can simply the graph by sweeping unused gate, using hash map to detect structurally equivalent signals, performing Boolean logic simulations to identify functionally equivalent candidate pairs in the circuit and calling Boolean Satisfiability (SAT) solver to prove the functional equivalence.

1 Data Structure

CirMgr contains a GateList used to store all gates. The pointer of each gate is stored in the slot corresponding to their IDs, so that accessing gates can be constant time. The DFS list is also stored in CirMgr as a GateList since it is heavily used in most of our functions. For each gate in CirGate, there are faninList to store the AigGateV of all inputs and fanoutList for outputs.

An overview of Data structure is drawn below.



2 Algorithm and Implementation

2.1 Sweep

Sweeping out those gates that are not reachable from POs, which means to remove them from their fanin gate's fanout list.

2.2 Optimize

If an AIG's fanin has constant 1 → merge it with the other fanin gate.

If an AIG's fanin has constant 0 → merge it with constant 0.

If an AIG's fanin has identical fanins → merge it with the fanin gate.

If an AIG's fanin has inverted fanins → merge it with constant 0.

When gate B is merged by gate A, the following three steps are executed. First, B's fanout gates shall change their input fanin IDs to the literal ID of A based on the sign of their connection. Second, B's fanin gates shall delete their fanout to B and reconnect to A. Lastly, B's fanout gates shall be pushed into A's fanout list.

2.3 Strash

The purpose of this function is to merge two gates with the same fanins. In this function, a HashMap is used to store the keys and the CirGate pointers. Since there are two fanins, the key is calculated by one multiplied by 2^5 plus another, so that the chance of collision can be decreased.

A strash algorithm is written below (from course note):

```
for_each_aiggate_from_pi_to_po(gate, hash)
    //Create the hash key by gate's fanins
    HashKey<...> k(fanin[0]<<5 + fanin[1]);
    size_t mergeGate;
    if (hash.query(k, mergeGate) == true)
        // mergeGate is set when found
        mergeGate.merge(gate);
    else hash.insert(k, gate);
```

2.4 Simulate

Parallel pattern encoding is used in this function. 32 patterns are stored inside an unsigned integer, and then we can use bitwise operation to calculate the simulation values for all gates. A HashMap is used to store the pairs. However, the insert function is different from the optimize function version; it won't allow collision to happen, and when a new node with different key value needs to be inserted, it will be stored into a separate vector instead. After that FEC group is simulated, the vector with collision gates will compare the key values of every possible pair to collect potentially equivalent pairs.

For the random simulation stopping criteria, if the FEC group size doesn't change after a determined maximum value, the random simulation is stopped. The maximum fail value I choose is estimated as the log of the total gate number multiplied by a constant.

```
for_each(fecGrp, fecGrps):
    Hash<SimValue, FECGroup> newFecGrps;
    for_each(gate, fecGrp)
        newFecGrps.query(gate, grp);
        if (grp != 0) // existed
            grp.add(gate);
        else newFecGrps.add(createNewGroup(gate));
CollectValidFecGrp(newFecGrps, fecGrp, fecGrps);
```

2.5 Fraig

The SAT solver is called to determine if the FEC pair is functionally equivalent. If the given function is reported to be unsatisfiable, one of them will be merged. Checking functional equivalence for AIG nodes n1 and n2 is performed as follows:

(1) assign each gate with a distinct SAT variable ID and record the mapping between gates and these IDs.

(2) call the interface function "addAigCNF" or "addXorCNF" to bulid the CNF formula for proof.

(3) specify the proof target by the function "assumeProperty". If necessary, release the previous proof target by the function "assumeRelease".

(4) prove it by "assumeSolve".

3 Experiment and Analysis

The output of all function mentioned above is mostly match to the reference program, however random simulation and fraig is slighty different because different amount of patterns are applied. But after fraig is done and sweep the unused gate, the outcome is still match the reference program.

The runtime of all functions except fraig is slightly slower yet acceptable, but my fraig function is much slower than the reference program while the correctness is still hold, it can reduce big circuit like sim12.aag to a constant 0 in 30 seconds.

3.1 Experiment of basic functions

In this experiment, basic function performance on larger circuits are estimated. I use do.opt that calls the three functions sweep, optimize and strash. All sim 01~15 files are performed, but only sim12 and sim13 are large enough to tell the difference.

File	sim12.aag	reference	sim13.aag	reference
Runtime(s)	0.09	0.05	0.72	0.55
Memory(MB)	3.523	2.262	29.14	17.52

The runtime is quite comparable, yet the memory usage can be around twice as large as the reference program. Sweep and optimize function doesn't need too much extra memory to perform, so I guess the reference program has a better control on the hashmap size.

3.2 Experiment on file simulation

The run.fsim script is used to estimate the performance of the file simulation. It simulates the circuit by patterns given for test, and print out the FEC groups to examine the correctness.

File	sim12.aag	reference	sim13.aag	reference
Runtime(s)	1.21	0.43	14.91	4.1
Memory(MB)	3.957	2.469	33.96	18.8

The runtime is about three times longer than reference program, and the memory usage is around twice as large as the reference program.

3.3 Experiment on fraig function

The run.fraig script is used to check performance of my fraig function. It contains random simulation and fraig function. The runtime may be significantly higher.

File	sim12.aag	reference	sim14.aag	reference
Runtime(s)	25.51	5.48	0.1	0.04
Memory(MB)	10.8	6.965	1.195	1.074

The runtime is much longer than reference program, I think it's because I don't have time to implement the simulation of the value SAT solver gives that distinguish the FEC pair.

4 Conclusion

The report shows that the AIGs powered by simulation and SAT lead to a representation called functionally reduced AIGs (FRAIG), in which each AND-node is functionally unique by construction. An implementation of FRAIGs is proposed, in that it detects and eliminates all functionally equivalent AIG nodes.

In the experiment, we can see the runtime and memory usage are not quite satisfying compared to reference program, so the next step should be the improvement of performance.

5 Acknowledge

經過一學期的~~疲勞轟炸~~知識薰陶，真心覺得自己對C++和資料結構掌握得更深了，每個作業和期末報告都相當優質，經過這樣的訓練，以後為了debug爆肝的機率一定降低了許多。除了感謝教授精闢的解說，還要謝謝辛苦的助教們7/24的待機為我們解決疑難雜症。