DsnP Final Project FRAIG – Report

電機二 B03901034 吳泓霖

E-mail: b03901034@ntu.edu.tw

Mobile: 0963723208

I. Introduction

The main idea of the project is to create a program to manage / optimize / simplify AIG circuits. The structure of the whole program mainly consists of two parts: the CirGate classes and the CirMgr class. The CirGate classes handles all per-gate functions and stores most of the information. Each object of CirGate can be consider as a component in the circuit. To manage all gates generally, we have to create a base class, and separately create different derived classes to implement different functions of the gate. On the other hand, the CirMgr can be consider as a circuit board, all gates are linked to the manager. We can do many structural functions in this class, so most of the optimization happens in this class.

II. Implementation

In this section I'll go through some special members and functions.

- CirGate

  Members:

  For all derived CirGate, each stores its own id, line number, current simulated value, and symbol. To link between gates, I used two pairs of vectors to record its fanins and fanouts. Each pair has a GateList (vector of CirGate pointer) and a bool vector to show if inverted. I also stored some other data, such as:

  bool _dfs: If true, this gate is in DFS list.

  bool _sim: Used in simulation, if true, the value changed in this simulation.

  UnsignedList* (unsigned vector pointer) _fec: Stores the pointer pointing to the gate's FEC group.

  unsigned _fecpos: The position of the gate in its FEC group.

  unsigned _gstat, _stat: _gstat is a static member, used to reset all markings at once. If _stat == _gstat, than the gate is marked.

  Unsigned _gfraigstat, _fraigstat: Same as above, used to mark gates done in fraig.

  Var _var: Used in SAT engine.

Functions:

void dfs(UnsignedList&): This function should only be called starting from PO. After calling this function, it will recursively call dfs to all its fanins before adding itself into _dfsList.

bool simulate(): This is a virtual function, which means different gate type has different functions. The return value shows whether the gate value has changed. This function will recursively simulate its fanins, check if its input has changed, update the value and return if the value has changed.

- CirMgr

Members:

GateList _idGlist: The only gate pointer source in the manager, all lists stores the id, and get its address through this list.

vector<UnsignedList*> _fecList: Store all FEC groups into a vector, each element points to a FEC group vector.

Functions:

void replace(Gate*& a, Gate* b, bool inv): This is the core function of all procedures. This function will replace 'a' with 'b', and invert it if inv = true. If b = 0, it will simply remove the gate from _idGlist and release memory. The procedure is:

1. Remove 'a' from all a's inputs
2. Connect a's outputs to b
3. Reconnect or remove (depends on b) a's outputs
4. Delete 'a' from _idGlist

Sweep function:

Scan through the whole _idList, if there is an AIG gate not in _dfsList, call replace function to remove it.

Optimize function:

Scan through the _dfsList, and check the following situation:

1. Const 1 fanin: Replace with the other fanin
2. Const 0 fanin: Replace with const 0.
3. Identical fanin: Replace with the fanin.
4. Inverted fanin: Replace with const 0.

Strash function:

I use HashMap in myHashMap.h to handle strash. The hash function in HashKey is:
For each fanin in AIG gate, calculate if invert: (id*2+1), else (id*2), then send the two values to the constructor. The id is unsigned (32 bit), size_t in 64 bit computer is 64 bit, so I shift the smaller one to the left, and store another one in the right. So in a size_t, we can store two unsigned in it. This method will create a distinct key for every different fanin.
Now we insert all AIG gates into the hash, replace all repeated gates.

Simulate functions:

The real simulation happens in function void sim(UnsignedList&) and simulation() in each gate. randomSim() and fileSim() only feeds the pattern to sim(). We use HashMap again to find collect FEC groups.

§ Input: Input the patterns to PIs (function setValue), if _fecList is empty, do initialize.

§ Initialize and first FECs: For the first time, it has to scan through all AIG gates and do once “all-gate-simulation” to find all initial values. The first FEC groups are special, because the first groups will contain the most pairs. Once the first pairs are determined, the pairs will only decrease. The initial FEC groups need to find out the inverted ones, so I created a class FecKey for hash function. It returns the smaller one of the signal or inverted signal. It means that no matter the signal is inverted or not, the hash function will always return the same value. Also, I added a function in HashMap --> checkInv. Send a key through this function, and it’ll check if there is an inverted key in the hash. This is used to determine whether the added gate is FEC or IFEC.

§ Not initialize: If FEC groups are initialized already, we simulate from PO to do “event-driven-simulation” (by using simulate in all gates, covered in CirGate). Also, we can consider the IFEC pairs as normal pairs, because it won’t increase anymore, so we can

treat inverted gates as normal gates. A new hash function class SimKey is used here. It is only a wrapper class of value. It will determine whether it is inverted or not at construction.

§ About random: For random, I used a formula $\log_{2.5}(dfsList.size)$ for max fail times. The fail time is the number of "FECGroup number hasn't changed" consecutively. For example, if max fail time is 3, and the total FECGroup number is 10, 9, 8, 8, 8 (terminates here). The number 2.5 is done by trial and error.

Fraig functions:

1. Set all gates with a Var, then construct the proof model.
2. Scan through AIGs in _dfsList, if it is in a FECGroup (fec != 0) and unfraiged (not visited by this fraig process before, this step is important, or it may form latches, which will totally break the circuit), access the FEC Group and start proving. The proving will have several situations:
   a. The first element of the FECGroup is const 0. This whole group can be proved directly (such as: If !G[10] is in the group, check if G[10] = 0 is SAT, if UNSAT, then G[10] = 1). Prove the the current gate with const 0 and continue.
   b. The first element is not const 0. Prove the current gate with other members in the group.
   c. For both situations above, if encounter UNSAT, save the pair to a vector (mergelist) for future merge and mark the gate.
   d. For both situations above, if encounters SAT, record the pattern and store it for future simulation
   e. If the stored pattern exceeds a limit (32 bit), do simulation and update the FECGroups
3. Merge all pairs in mergelist by using function replace.
4. Update FECGroups and dfslist, if there still exists FECGroups, restart the whole procedure from 1 until number of FECFroups is 0.

III. Performance and Discussion

Use the largest case as benchmark: sim13.aag

My program	Reference
fraig> cirr fraig> usage Period time used : 0.2 seconds Total time used : 0.2 seconds Total memory used: 28.06 M Bytes	fraig> cirr fraig> usage Period time used : 0.06 seconds Total time used : 0.06 seconds Total memory used: 11.99 M Bytes
fraig> cirsw fraig> usage Period time used : 0 seconds Total time used : 0.2 seconds Total memory used: 28.07 M Bytes	fraig> cirsw fraig> usage Period time used : 0.01 seconds Total time used : 0.07 seconds Total memory used: 11.99 M Bytes
fraig> ciropt fraig> usage Period time used : 0.01 seconds Total time used : 0.21 seconds Total memory used: 28.07 M Bytes	fraig> ciropt fraig> usage Period time used : 0 seconds Total time used : 0.07 seconds Total memory used: 12 M Bytes
fraig> cirstr fraig> usage Period time used : 0.04 seconds Total time used : 0.25 seconds Total memory used: 29.77 M Bytes	fraig> cirstr fraig> usage Period time used : 0.04 seconds Total time used : 0.11 seconds Total memory used: 15.9 M Bytes
fraig> cirsim –r 25024 patterns simulated. fraig> usage Period time used : 10.79 seconds Total time used : 11.04 seconds Total memory used: 32.73 M Bytes	fraig> cirsim –r 20224 patterns simulated. fraig> usage Period time used : 4.23 seconds Total time used : 4.34 seconds Total memory used: 18.69 M Bytes
fraig> cirf fraig> usage Period time used : 119.4 seconds Total time used : 130.5 seconds Total memory used: 55.45 M Bytes	fraig> cirf fraig> usage Period time used : 107.9 seconds Total time used : 112.3 seconds Total memory used: 44.2 M Bytes

In observation, although many functions are slightly slower than reference program, most of them are in the same order. CirSweep cause no time for me because I added a small check that if _unusedlist size = 0, then skip all steps. Unlike reference program's random simulate, my simulation effort depends on the random generated numbers, but the reference program always has the same simulation patterns.

The bottleneck in simulation seems to be the time spent on maintaining the feclists. I tested the running time required when I haven't handled feclist, it is almost 4 times faster than reference. But after I added the implementation, it causes the total running time to be twice longer than reference. So the main problem should be in the maintaining process.

In fraig, there may involve many problems. Once I didn't checked the order of the proving process, it generates latches at merging gates, which took me very long to find the problem. To be honest, there are still many problems in the code, not all tests can be done, but at least the largest case sim13, I optimized even further than the reference, and didn't took too much time, also proved correct in CirMiter.