DSnP Final Project

Functionally Reduced And-Invertor Graph (FRAIG)

Name : 陳昊

Student ID : b03901016

e-mail: b03901016@ntu.edu.tw

January 21, 2016

(Note : The Parsing part and the codes of #HW6 in this project is given by Terry Cheong - b03901175 .)

1. Introduction

We are going to implement a special circuit representation, FRAIG(Functionally Reduced And-Invertor Graph), from a circuit description file. In this project, we expand some new commands, which actually do the procedure to reduced the circuit to a more simple form.

2. Principles

This project should provide the following features:

- Parsing :

Parse the circuit description file (AIGER format). In this project we do not need to redo this part since it's already covered in #HW6.

- Sweeping :

Sweep the gates which cannot reach from POs (the gates that are not in the _dfsList).

After this operation, the gates that are originally “defined-but-not-used” will be detected and removed. Since these gates are removed, the UNDEF_GATEs whose fanout are connect to the removed gates should also be removed. However, although after this operation, some Pis might have empty fanout, they shouldn't be delete anyway. (Note : the PIs with empty fanout will be add to unused list and will be report by other command)

- Optimizing :

Provide some trivial Boolean operation, and simply optimize the circuit.

Including: (a) If both fanins of an AND are the same, merge it with it's fanin. (b) If one fanin is inverse to the other, replace the gate by 0. (c) If one of the fanin is 0, replace the gate by 0. (d) If one of the fanin is 1, replace the gate by the other fanin.

After executing the operation above, check and guarantee that there are no unused null pointers.

- Strashing :

Provide a structural hash map to store some structurally equivalent gates, and merge all the structurally equivalent gates (For example, AND(a, b) & AND(b, a)) together. This feature can let us keep simplifying the circuit when the circuit couldn't be optimized anymore.

-Simulating:

Some subsets in the whole circuit are actually same, but we cannot separate them by the reducing method above. For Example, ((C & A) & B) and ((C & B) & A) have different structure in circuit parsing, however, it's trivial to know that they are 100% equivalent by Boolean Algebra. To proof two circuit are absolutely same, one method is to try all the inputs and proof that the two circuit always have the same

outputs, but it's not a fine method since it cause a lot of time to simulate through all the gates. Why we still do simulating? It's because we can first roughly separate the gates into groups (FEC pairs, FEC groups) and can easily separate most of the different gates. So that if we need to use SAT engine to solve the problem, it will be more efficiently.

- Fraig :

Call the SAT engine to proof the rest different gates that still not separate yet. After this procedure, the whole circuit should theoretically be the simplest and cannot be simplified anymore.

3. Algorithms

For each functionally reduce principles, there are alternative ways to implement. The following is the algorithms and the ways I implement those principles in my program, and I would like to show some pseudo code of each function.

- Sweep :

In my program :

```
for all gate without PI and PO
    check_If_is_in_dfsList ( gate )?  continue : remove
```

Since I remove the gate while traveling through the netlist, it's necessary to link the all the fanout correctly, so I correct the fanout every time I removed a gate. However, it's a $O(n^2)$ Algorithm, so it's not a good way to implement this function, but I didn't change it to an alternative way since I have no time to rewrite this function.

Here's a better way to implement the function:

```
for each gate in _dfsList
    mark( gate );
for all gate without PI and PO
    gate.if_is_marked()? Continue :  remove
```

In this method, it's $O(n)$, much better.

- Optimize :

In my program :

```
for each gate in _dfsList from begin() to end()
    n1, n2 = gate->_fanins
    if ( n1 == n2)              return n1;
    if ( n1 == ~n2)     return 0;
    if ( n1 == const)  return 0 or n2;
```

```
if ( n2 == const)  return 0 or n1;
```

Here in the project I wrote a merge(gate1, gate2, size_t A) function to replace one gate to another, and input a flag that make sure the inverse-bit is correct (A XOR fanout_is_inverse()). Since the simplifying procedure operate from begin() to end(), it's a simple $O(n)$ function.

- Strash :

In my program :

```
HashMap<Key, gate> myMap(size)
for each gate in _dfsList from begin() to end()
        if (myMap.check(structurally equivalent gate))   merge();
        else        myMap.Insert();
rebuildDFSList();
```

This is a simple way to implement the function, however, the tricky part is how to decide the initialize size of the map. It may effect the performance a lot because of the feature of Hash. The value of size I decide to use in my program is 1.6*_dfsSize, it's because it's would be a Fibonacci Hash (Best balance in memory used and speed). Besides, the “Key” I use here is a array of size_t which has size = 2, note that the value in the array is the fanin-bit of a gate. In class Key, I design the operator () to return a value of “fanin[0] + fanin[1] + fanin[0] & fanin[1]”, it has better performance than just return a value of “fanin[0] + fanin[1]”, so that it could prevent countering the same bucket in HashMap all the time, and greatly speed-up the run time.

- Simulate :

Generate some values and assign to each PI, and use simple Boolean operation (AND, NOT) to set the simulate_value to all the gates .

Create a HashMap whose Key is the simulate_value, and store the gates with the same simulate_value together (a.k.a FEC_Group), at last put all the FEC_Group to a list that store all the FEC_Group and then collect the valid group (whose size > 1).

(Note : The size of the HashMap should be well designed, otherwise the run time might be hundreds times longer. In my implementation, I choose to use the function getHashSize(size_t) to initialize the size of the HashMap.)

The challenging part of simulation is “Random_Simulate”, it's hard to decide the MAX_FAILS. I thought that it couldn't be linear because it would be an $O(n)$ if so, and it will really cause a long time to reach the end if the net of a circuit is really huge. I couldn't actually figure it out, but I consider it's should be a $O(\log(n))$, so I roughly decide the MAX_FAILS of my program to be “log2(_dfsSize) + 4” ! Why +4 ? I guess so, because it's closer to the value of ref.

- Fraig :

For my method, I will like to link each gate to it's corresponding FEC_Group after simulation, so that I can follow the DFS traversal to use SAT engine to solve the group from PI to PO. It will be more efficiently doing in this way.

I only do the UNSAT part in Fraig, while I do not consider the SAT part yet.

However, I face a lot of Segmentation fault while coding the function Fraig() using the method I mentioned above, some smaller data for testing is OK for my program to execute, while the others might cause Segmentation fault. So, I choose to use the simplest method that's $O(n^2)$.

4. Experiments

The experiments are focus on the run time of the program, let us neglect the memory usage.

It's not very necessary to calculate the performace of Sweep() and Optimize() because they almost finish shortly at a time.

The value in the table is an average of 5 times. (without g++ -O3)

size(gates)	Sweep	Optimize	Strash	Simulate(-r)	Fraig
10^2	0.01	0.01	0.00	0.01	0.00
3×10^2	0.01	0.01	0.01	0.05	0.00
10^3	0.01	0.16	0.01	0.08	0.24
10^4	0.02	0.24	0.10	0.66	144.4
10^5	No case	No case	No case	12.47	Run time error

(seconds)

(Note : Although the complexity of my implementation of Sweep is $O(n^2)$, it's still run very quickly, maybe it's because there aren't so many gates to sweep.)

For Sweep, Optimize, Strash, because it's not every gate in the circuit can be swept, optimized or strashed, so for a 10000 gates circuit, it's still very fast, I think most of the time is spent on printing.

For Fraig, I've no time to improve the efficiency.

5. Reviews

(This part is not relative to above.)

After this semester, I feel my coding ability is better than before, but still have a lot to improve. I always spend a lot of time debugging, and often have some silly bugs and a lot of segmentation fault. Sometimes I feel that I know how to do it in my mind, I know the algorithm very well, but when I sit in front of my computer, I just couldn't convert what I thought in my mind to C++ language. I still need to learn a lot, especially that I fail to complete the whole final project, fail to handle a better way of Simulation and Fraig. At last,

I would like to thank professor a lot, going through all these challenging homework, I think I won't be afraid of coding like before. (Murmur: Final project is really too hard for me, I think it's my limit to reluctantly finish Simulation and Fraig)