DSNP Final Project Fraig Report

電機二 李高迪 B06901118

Email : b06901118@ntu.edu.tw
godiclili123456@gmail.com

Table of Contents

Part 1 Introduction

The main aim of this project is to write a C++ program to parse a circuit description file in AIGER format and to simplify the AIG circuit.

The circuit is read in by command CIRRead, which can report if there are any errors in the AIGER file and then construct the circuit netlist.
There are several query commands (CIRPrint, CIRGate, CIRWrite) to print circuit information. Five Commands are used to simplify the circuit. (CIRSweep, CIROptimize, CIRStrash, CirSimulate, CirFraig)
Most operations in the program are implemented in the class CirMgr, which is a circuit manager for circuit construction, reporting netlist, simulation and optimization. Every gate in the circuit is an instance of the class cirGate. In general, my program is able to simplify all AIG circuit to the simplest (given no gates with floating fanins)

In this report, I will first introduce some important data members in CirMgr and CirGate. Afterwards, for each of the five optimization commands, I will explain my implementation, algorithm and some of the problems I encountered in details. Finally, I will analysis the results of my program by doing experiments and comparing with the ref program. Bottlenecks in the runtime efficiency and memory usage, as well as room for improvement are included at the very end.

Before starting the report, I would like to thank professor Chung-Yang Huang (Ric) for his teaching and guiding this semester. I wouldn’t be able to finish this project without his help and encouragement.

Part 2 Implementation and Data Structure

1. Data Members in CirMgr

1. **map<int, CirGate*> gateList**
   Include all existing gates in the circuit. I use a map to store so that each gate can be queried in O(logn) time by its ID
2. **vector<CirGate*> piList, poList**
   Lists of all inputs and all outputs are included, so that there is no need to traverse the whole gateList when only inputs / outputs are needed
3. **vector<pair<CirGate*, vector<int>>> aigList**
   This list is only used when constructing circuit from file
4. **vector<CirGate*> dfsList**
   Store gates in depth-first-search order. It is frequently updated in optimizing functions. Every gate in dfsList has its data member “visited” set True.
5. **set<int> floatList / set<int> notusedList**
   Store float gates and gates that can’t be reached from output respectively
6. **deque<pair<int,map<int,pair<CirGate*, bool>>>> _fecList**
   Deque data structure is used because I have to maintain fecList by both pushing front and popping back. Each pair in the deque is a fecGroup. Every gate has a data member “gatefecList” pointing to “map”.
   Details of implementation are included in the section “Simulate”
7. **vector<pair<CirGate*, pair<CirGate*, bool>>> mergeList**
   Gates waiting for merge is put here temporary

2. Data Members in CirGate

Storing Basic Information :

1. int ID 2. int LINE 3. char TYPE 4. string NAME
5. int INDEX (used in the circuit construction process)
6. bool visited (used when constructing circuit’s Depth-First-Search List)
7. bool fraiged (used in function CirFraig)
8. bool removed (used in function CirFraig)
9. size_t value (used in function CirSimulate to store input pattern)
10. Var var (used in function CirFraig to interact with SAT solver)
11. pair<CirGate*, bool> fanin1, fanin2 (store inputs of gate)

12 **multimap<int, pair<CirGate*, bool>> _fanoutList** (store outputs of gate)

13. **map<int, pair<CirGate*, bool>>* _gatefecList** (store fecList of gate)

3. Sweep

Overview

The command CIRSWeep removes unused AIG gates that can't be reached from POs. I scan through the whole gateList to search for gates that are "not visited" by depth-first-search before, removing all unused AIG gates and UNDEF gates. PI gates which have no fan-outs will be added into unusedList.

Time Complexity: Linear to size of gateList

Simplified Pseudo Code

Function sweep in CirMgr

```
1   for ( each Gate in gateList ) :
2       if ( Gate is Visited in DFS ) :    Skip to next Gate
3       if ( Gate is AIG ) :  Remove the Gate and update floatList / notusedList
4       if ( Gate is UNDEF) :  Remove the Gate
```

4. Optimize

The command CIROPTimize performs trivial optimizations on AIG gates in the dfsList, there are generally four cases.

1. If one of the fanins is zero, merge the gate with CONST 0
2. If one of the fanins is CONST 1, merge the gate with other fanin
3. If both fanins are the same, merge the gate with fanin
4. If one of the fanins is inverse to other fanin, merge the gate with CONST 0

When scanning through the dfsList, I classify the gate to be deleted into two general categories, then merge and connect the circuit. dfsList and notusedList are reconstructed afterwards if there is any removal of gates.

Category 1: The gate is replaced by CONST 0

Category 2: The gate is replaced by one of its fanins

Time Complexity : Linear to size of dfsList

Note : I didn't think thoroughly before coding, so the code in this function is quite messy and I spent quite a lot of time in debugging. From this experience I leant that it is sometimes very useful to devise the algorithm on paper before starting to code.

5. Strash

Overview

The command CIRSTRash performs structural hash on the circuit's DFS List and merges structurally equivalent gates.

I implement Hash by STL unordered_map, with size_t as Hash key and CirGate* as Hash value. A function "Strashkey" is used to map each gate into a unique integer.

Time Complexity

This operation has linear time complexity to size of dfsList
Traverse and rebuild dfsList once O(n), hash insertion and query O(1)

Although Fraig can find all functionally equivalent gates, including those which can be found by Strash, it is sometimes desirable to perform Strash to remove structurally equivalent gates before performing Simulate and Fraig since Strash is much less time-consuming.

Implementation of Strash Key in Hash

I implement a function called StrashKey, which takes in a pointer to CirGate as the argument and return f(a,b), as calculated below.
Let a = Gate_Fanin1_ID x2 (+1 if inverted)
Let b = Gate_Fanin2_ID x2 (+1 if inverted)

$$f(a,b) = \frac{\max(a,b)(\max(a,b)+1)}{2} + \min(a,b).$$

This function provides a unique mapping from 2 non-negative integers to another integer. Also, f(a,b) = f(b,a). Therefore, structurally equivalent gates can be easily distinguished by using this Strash Key.

Simplified Pseudo Code

Function strash in CirMgr

```
1   Initialize Hash with [Key = Integer] [Value = CirGate*]
2   for ( each AIG Gate in dfsList ) :
3       if ( Hash_Find StrashKey(Gate)) : Merge Gate with Hash_Value
4       else : Hash_Insert [Key = StrashKey(Gate)][Value = Gate]
5   Update floatList and dfsList if necessary
```

6. Stimulate

Overview

The command CirSimulate performs Boolean logic simulation on the circuit to distinguish potentially functionally equivalence gates. It can read in a pattern file to simulate or generate random simulate patterns by calling rnGen. Parallel simulation (64bits) is utilized to enhance performance.

When the command CirSimulate is first called, function “add_first_FEC” is called to initialize the fecList with a fecGroup containing all gates in the dfsList. Afterwards, function “Simulate” is called to do gate simulation in the dfsList and FEC groups are recombined. (See Pseudo Code for details). Finally, function “store_FEC” is called to sort fecList and store pointer of fecGroup to each gate.

How to Maintain fecList

It is quite tricky to maintain the circuit fecList. While traversing the whole fecList, new fecGroups may need to be added to the fecList, and the original fecGroup is deleted. A queue data structure is well suited for this purpose. Because all fecGroups originally in fecList need to be deleted, I traverse the fecList for n times, where n is original number of fecGroups in fecList. In each traversal, I check the first fecGroup in the queue, using a function “Simkey” as Hash key to check whether two gates have the same simulation pattern (or inverse simulation pattern). Afterwards, if more than 2 gates have same simulation pattern, new fecGroup is pushed into the queue. Finally, the original fecList is pop from the front of queue.

(Note: I use STL deque instead of queue, so that it can be accessed like a STL vector afterwards)

Effort of Random Sim

I set a few criteria to determine how many patterns to simulate.

- If already fraiged, don’t stimulate again
- Max number of simulations: 262144
- Min number of simulations: 4
- Stop Stimulate after a certain number of simulations with same number of FEC groups
  (This number ranges from 3 to 100 according to numbers of inputs)

Simplified Pseudo Code

Function Simulate in CirMgr

```
1  for ( each AIG or PO Gates in dfsList ) :
2          Get Pattern in each of its Fanin(s)
3          Perform Bitwise "NOT" ( if inverted ) / "AND" and Store Pattern
4
5  Size = Initial Number of FEC groups in fecList
6  for ( Repeat "Size" times ) :
7          Initialize Hash with [Key = Pattern] [Value = FEC group]
8          for ( Gates in First FEC group in fecList ) :
9                  if ( Hash_Find Gate's Pattern ) :
10                         Insert Gate into Hash_Value (FEC group)
11                 else  Hash_Insert [Key = Pattern][Value = new FEC group]
12         for ( Each FEC group in Hash having more than 1 Gate ) :
13                 Insert FEC group to Back of fecList
14         Pop First FEC group in fecList
```

7. Fraig

Overview

The command CirFraig calls oolean Satisfiability solver (SAT) to determine whether each gate pair in fecList is functionally equivalent (called UNSAT) or functionally inequivalent (called SAT). If gate pair is proven SAT, the pattern leading to SAT is reused to simulate the circuit again because it may be useful to distinguish other FEC pairs. If gate pair is proven UNSAT, two gates are merged together.

In my implementation, I skip to next FEC group whenever a FEC pair in the group is proven SAT. If a FEC pair is proven UNSAT, it is temporary stored in a "mergeList". If 64 patterns are collected, the circuit is simulated again to reconstruct FEC group. The whole procedure is repeated after no fecGroup is left in fecList. Finally, gate pairs in mergeList are merged together and dfsList is reconstructed.

Comparing different implementation's efficiency

1st attempt: Do not restimulate patterns from SAT

The performance is acceptable when number of input gates are small; however, it takes really long time to fraig sim13.aag using this method.

2nd attempt: Skip to next FEC group if a FEC pair in the group is proven SAT, after all FEC group is traversed once, stimulate the circuit, and restart the whole process if there are still FEC group in FEC List

If any FEC pair is proven SAT, it is of utmost urgent NOT to continue proof on same FEC group since the pattern retrieved from SAT is most useful in the same FEC group. I assumed that the pattern from SAT can only distinguish not equivalent pairs in the same FEC group in this attempt. The performance is much better than the first attempt (Reduce time usage by a half)

3rd attempt: Similar to attempt 2, resimulate when 64 patterns are collected

It turns out my assumption was wrong. Pattern from SAT can actually distinguish not equivalent pairs outside of the original FEC group. Though, it is only 10% faster than the 2nd attempt. I choose to use this implementation in my program.

When to Merge Equivalent Gates?

At first, I merge equivalent gates in the mergeList before stimulate on patterns from SAT; however, I found it time-consuming to reconstruct dfsList, and discovered a serious problem for this implementation:

- **Gate Proved to be SAT not in dfsList anymore after removing UNSAT gate**
  After removing UNSAT gate, dfsList is changed. However, sometimes the gate proved to be UNSAT is the only path SAT gates can reach outputs, so the SAT gates are no longer in dfsList. Therefore, when reperforming stimulation after the input pattern is collected, SAT gates can't get the pattern that can distinguish them from each other.

Thereafter, I only merge equivalent gates after number of FEC groups turns zero. It saves around 3% time and successfullly avoids the above problem since dfsList is only reconstructed just before leaving the "fraig" function.

Special Case: Floating Gates

It is quite panic to handle floating gates that can be reached from output. If floating fanin is viewed as Const 0, that floating gate (and all gates above) will merge with Const 0. I tried deeming each floating gate to be an extra input; however, a lot of bugs are generated and I failed to debug. Therefore, I retreated to view floating fanin as Const 0. (RIP)

Special Case: Merging gate with its fanin

Sometimes, two gates in the same FEC group are directly connected. Since the order of FEC group is sorted by gate ID, not by dfsList, the merging process may result in a gate which fanins are the gate itself, leading to incorrect circuit structure. Whenever a gate pair is proven UNSAT, I check if the above situation occur. If so, I would reverse the order of merging.

Safety Measure

A variable “effort” is added to avoid falling into infinite loops. Although I have already handled some special cases stated above to avoid infinite loops, I am not sure if there is still any unknown bug in the program (due to some special circuit structure that is not presented in the testcases)

If a pair of gates in the FEC List is proven SAT, “effort” is incremented; if proven UNSAT, “effort” is reinitialized to zero. If “effort” is larger than 2000, fraig function is forced to terminate. Whether the circuit is left correctly connected at this moment is a riddle: it may still function correctly, or it crashes. This safety measure may lead to disaster only if a nomral circuit has gate pairs proven SAT for 2000 consecutive times, which is a trivial case.

Simplified Pseudo Code

Function fraig in CirMgr

```
1   if ( fecList is empty ) :               Quit Function
2   if ( Already Performed Fraig ) :        Quit Function
3   Initialize SatSolver and Generate Proof Model
4
5   while ( fecList not Empty ) :
6       FLAG   Skip_Zero = False
7       FLAG   Skip_Other = False
8       for ( each AIG Gate in dfsList) : Unset Fraiged
9       for ( each AIG Gate in dfsList ) :
10          if ( Gate not in any FEC groups) :  Skip to next Gate
11          if ( Gate is set Skipped )          :  Skip to next Gate
11          if ( Gate is potentially CONST 0 ) :
12              if ( Skip_Zero is True ) : Skip to next Gate
13              Result = Proof ( CONST 0, Gate)
14              if ( Result is UNSAT ) :   add gate pair to mergeList
15              if ( Result is SAT) :
16                  set Skip_Zero to True
17                  store input pattern for stimulation
18                  if ( 64 patterns collected ) :     Jump to Line 31
19              Skip to next Gate
20          for ( other_gate in Gate's FEC group ) :
21              if ( Skip_Other is True ) : Skip to next Gate and set Gate Skipped
22              Result = Proof ( Gate, other_gate )
23              if ( Result is UNSAT ) :   add gate pair to mergeList
24              if ( Result is SAT ) :
25                  set Skip_Other to True
26                  set Gate Skipped
27                  set other_gate Skipped
28                  store input pattern for simulation
29                  if ( 64 patterns collected ) :     Jump to Line 31
30          set Skip_Other to False
31      if ( there exists gate pairs proven to be SAT ) :
32          Use stored input pattern to stimulate the circuit and update fecList
33  Merge gates in mergeList and Update dfsList
34  Update notusedList
```

Part 3 Results and analysis of performance

Experiment

For all test cases, I test my program with commands in the following order :
cirr xxx.aag → cirSimulate –r → cirFraig → cirSweep → cirOptimize → cirprint → cirprint –netlist → cirwrite
For reference program, I repeat the four optimization command if the circuit isn't optimized to the simplest (e.g. sim13.aag need 3 repeats)

Correctness

I compared my program with the reference program by "diff" result of cirprint, cirprint –netlist and cirwrite. For most of the testcases, my output is exactly the same as the reference. However, I am not able to correctly handle the case with floating input in the dfsList (e.g. strash05.aag) because I simply treat UNDEF gate as CONST0, thus optimizing all AIG gates away.
In addition, my program optimize away a few gates than the reference program in two of the testcases (C499_r.aag, C1908.aag). I am not sure if I accidentally optimized away functionally unequivalent gates or not, since the addtionally removed gates are proved UNSAT in the debug message.

My program may still fall into infinite loops in Fraig. I brutely quit the loop by an effort counter. The result afterwards may still be correct.

Usage

Since the largest testcase (sim13.aag) consume significantly long running time, so I split the experiment on usage into two, one with sim13.aag, the other one with all other testcases except sim13.aag.

sim13.aag

	Ref Program	My program
Time used	67.52s	58.17s
Memory used	47.2M	68.3M

Other test cases

	Ref Program	My program
Time used	6.06s	5.91s
Memory used	8.31M	11.48M

It turns out my program has comparable runtime with the refernce program overall, while memory usage is significantly higher.

Bottlenecks in runtime efficiency

My program is slower than the reference program in most commands apart from fraig. For instance, with the same number of pattern simulated, my program is almost two times slower than the reference. For sweep, optimize and strash, my program is even three times slower.

I believe that my program still has a linear time complexity in these commands; however, my program has a much higher constant factor. Here are some issues that may lead to the result.

- **Complicated Data Structure**
  Because my program has some "complicated" data member
  e.g. deque<pair<int,map<int,pair<CirGate*, bool>>>>
  It results in a bunch of dereference and pointer access
  e.g. it.second.first->_fanin1.first->_value
  Although pointer access takes almost no time, it may still contribute to the runtime difference (i.e. 0.1s vs 0.3s)

- **Some Reduntant Safety Measure**
  When debugging, I added quite a lot of "safety measures" to figure out the problem, such as traversing the whole gateList to set a variable back to its orignial state; however, I don't know whether some codes can be deleted afterwards.

Bottlenecks in memory usage

My program has a memory usage of about 1.5 times higher than the reference. The main reason is that I failed to implement polymorphism and spent an extra bit to store bool for every fanin / fanout.

For each gate, the memory usage is 55bytes + sizeof(Var) +
number of fanouts * 13 (multimap<int, pair<CirGate*, bool>>).
I realized that it is not a must to store the mapping between gateid and gate in fanoutList; however, it is too late for me to change the implementation because I have finished most of the code at that time.

I am quite confident that the memory usage can be reduced significantly if the above issues are addressed.