# DSNP

## Final Project Report

電機三

B04901020 解正平

Email：b0491020@ntu.edu.tw

Phone：0932329263

I. Data Structure

II. Algorithm

III. Experiments

IV. Discussion

# I. Data Structure

## 1. CirGate

```
protected:
  vector<CirGateV>  _fanin;
  vector<CirGateV>  _fanout;
  IdList _faninID;
  unsigned  _linenum;
  unsigned  _id;
  GateType _type;
  static  unsigned _globalRef;
  mutable unsigned _ref;
  int fecpos;  //sim
  size_t _sim;  //sim
  Var _var; //fraig
  static  unsigned _globalRef2; //fraig
  mutable unsigned _ref2; //fraig
```

I create **_faninID** to record all gates fanin ID for readparse. More specialy, the **CirgateV** fanin and fanout is a class that can contain the invertor information. The **fecpos** is used to record the gate's fecpairs and the **_ref2**, **_globalRef2** can be found in Cirfraig.cpp using for traverse between group gates.

## 2. CirMgr

```
private:
   //sim
   ofstream           *_simLog;
   vector<FecGrp>     _FecGrps;
   IdList _pilist;
   IdList _polist;
   GateList _totallist;
   mutable GateList dfslist;
   vector<bool> booldfs; // for sweep
   int M,I,L,O,A;
```

**_pilsit** and **_polist** are created when reading files, it won't change until read files replace. **_totallist** is an arrary to store every gates information, using its ID to place the gate into the arrary position. **dfslist** is an list that run dfstravel from PO without the floating gates and undefined gates. The **booldfs** vector is use for Cirsweep while **_FecGrps** is to record the sorted fecpairs using for Cirsimulation and Cirfraig. **M I L O A** are the parameters from reading files **A** would change while merging and sweeping.

## 3. CirSweep

```
for(size_t i=1;i<_totallist.size();i++){
    if(_totallist[i]!=0){
        if(_totallist[i]->getTypeStr()!="PI"){
            if(booldfs[i]!=true){
                if(_totallist[i]->getTypeStr()=="AIG")A--;
                cout<<"Sweeping: "<<_totallist[i]->getTypeStr()<<"("<<_totallist[i]->getID()<<")";
                cout<<" removed..."<<endl;
                delete _totallist[i];
                _totallist[i] =NULL;
for(size_t i=1;i<_totallist.size();i++){
    if(_totallist[i]!=0){
        for(size_t j=0;j<_totallist[i]->_fanout.size();j++){
            if(_totallist[(_totallist[i]->_fanout[j].gate()->getID())]==NULL){
                _totallist[i]->_fanout.erase(_totallist[i]->_fanout.begin()+j);
```

I run two for to deal with sweep. The first loop **eliminates the floating gates and undef gates** while the second loop **eliminates the fanouts of the gates** with floating gates or undef gates. Pay attention acces the Cirgate pointer from causing core dumped.

## 4. CirOptimize

```
if(f0.gate()->getID()==0)
    if(f0.isInv())
        oneinput(dfslist[i]);
    else
        zeroinput(dfslist[i]);
else if(f1.gate()->getID()==0)
    if(f1.isInv())
        oneinput(dfslist[i]);
    else
        zeroinput(dfslist[i]);
else if(f0.gate()->getID()==f1.gate()->getID())
    dirty = true;
    if(f0.isInv()==f1.isInv())
        oneinput(dfslist[i]);
    else
        zeroinput(dfslist[i]);
```

Distinguish the optimize problems in three cases for fanin0 is 0 or 1, fanin1 is 0 or 1 and two fanin is the same but with invertor or not. And there are **two method** to deal with cases, one is **merged with the fanin gate** and the other is **merged with the const0 gate**. I would introduce in Algorithm part more detailed. In the last steps we need to recreate the dfslist due to the merges and changes in gates.

# 5. CirStrash

```
size_t operator() () const
{
 //return _faninhash[0]+_faninhash[1]+_faninhash[0]%256*_faninhash[1]%256;
 //return _faninhash[0]+_faninhash[1]<<5;
 return (_faninhash[0]>>6&0xfff)*(_faninhash[1]>>6&0xfff);
}
```

```
HashMap<HashKey,unsigned> _CirHash(dfslist.size());
for(size_t i=0; i < dfslist.size();i++)
    if(dfslist[i]->getTypeStr()=="AIG")
        HashKey k(dfslist[i]);
        unsigned id;
        if(_CirHash.query(k,id))
            merge(_totallist[id],dfslist[i]);
            delete _totallist[dfslist[i]->getID()];
            _totallist[dfslist[i]->getID()]=NULL;
            A--;
        else
        _CirHash.insert(k, dfslist[i]->getID());
genDFS();
```

I try many methods to create a hashkey in order to do strash more efficiently；finally, I choose the **multiplication** of two fanins.to form the hashkey. In strash function, there are two results. If we can find it in the hashmap, we **merge the gate** with the gate in the hashmap. If can't, we i**nsert a new hashnode** into the hashmap. In the last steps we need to recreate the dfslist due to the merges and changes in gates.

# 6. CirSimulation

```
size_t operator() () const
    return (_sim > (~_sim))?(_sim):(~_sim);

bool operator == (const simKey& k) const
    return (_sim==k._sim||(~(_sim)==k._sim);
```

```
bool checkpattern(string &);//readparse
void setallsim();//set simValue
void setFecGrps();//set fecpairs
void updateFecGrps();//set every gates' Fecpos
void sortFecGrps();//sort pairs
void sortFecGrp(FecGrp&);//sort gates
vector<FecGrp> getGrps(){return _FecGrps;}
FecGrp getFecGrp(unsigned num) const {return _FecGrps[num];}
void simlog();
```

I create many functions for simulation. First in order to use hashmap, I create simkey that the if simValue is **the same or invert** they may in the same buckets because it may become fecpairs or Ifecpairs. About the functions, there are read part, create part and set part. We may read input pattern or random generate first and then **set all gates' simValue** according to dfslist steps by steps.

```
HashMap<simkey,FecGrp*> _FecMap(s);
for(int j=0; j<_FecGrps[i].size(); j++)
    simkey k(_totallist[_FecGrps[i][j]]->_sim);
    FecGrp *gp;
    if(_FecMap.query(k,gp))
        gp->push_back(_FecGrps[i][j]);  // set old grp
    else
        FecGrp* temp = new FecGrp;
        temp->push_back(_FecGrps[i][j]);
        _FecMap.insert(k,temp);          // set new grp

HashMap<simkey,FecGrp*>::iterator it = _FecMap.begin();
for(;it != _FecMap.end(); it++)
    if(it!=0)
        if((*it)->size()>1)
            sortFecGrp(*(*it));
            temp.push_back(*(*it)); //put hash grp into _Fecgrps
    delete *it;
```

Next, we create fecpairs using hashmap and the new pairs need to update to the **_Fecgrps.** In hash, if we can find the same simvalue then **add to the buckets** while can't find we **insert a new pair** into hash. We use **iterator** to push back the hash into groups, and sort the gates at the same time. Finally we create all the groups, we sort the groups in ascending.

# 7. Cirfraig

```
void runfraig(SatSolver&);
void fraigInit(SatSolver&); //init var
bool SATsolve(SatSolver&, CirGate*, CirGate*, bool);//solve sat
void fraigmerge(vector<mergeGate>);//merge gate in pairs
void storeCEX(SatSolver&);//store counterexample
```

I create many function to do different things. First we need to **initialize** the satsolver and every gate' varvalue and then we runfraig in special steps with two gates in the same fecgroup we **satsolve** the two gates if true we **merge** if false we **store the CEX** for simulate and recreate the dfslist. We always run the dfslist from the beginning after merging. If we run all the dfslist we stop fraig and clear the _Fecgrps.

***For(dfslist)***
***If(gate's pair is 0)***
***If(fanin is const 0)***
***gate merged with const0***
***else If(satsolve is 0)***
***gate merged with const0***
***Else***
***StoreCEX***
***Else***
***For(fecgroup)***
***If(satsolve is 0)***
***Groupgate merged with gate***
***Else***
***StoreCEX***

# II. Algorithm

## 1. CirSweep

```
for(unsigned i=0;i<_fanin.size();i++)
  if(!_fanin[i].gate()->isGlobalRef())
    _fanin[i].gate()->setToGlobalRef();
    _fanin[i].gate()->dfsTraversal(dfsList,dfs);
dfsList.push_back(this);
dfs[this->getID()]=true;
```

I create a booldfs with size the same the totallist to record whether the gate is in the dfslist when doing dfstraversal. And then we just run the totallist with booldfs steps by steps and eliminates the gates which need to be swept. The time complexity is approximately $O(n)$.

## 2. CirOptimize

```
for(size_t i=0; i < gate->_fanin.size();i++)
    CirGate* Ingate = gate->_fanin[i].gate();
    for(size_t j=0; j< Ingate->_fanout.size();j++)
        if(Ingate->_fanout[j].gate()->getID()==gate->getID())
            Ingate->_fanout.erase(Ingate->_fanout.begin()+j);
            break;
```

it is a remove fanin function two for loop to assure the fanin don't have the fanout to this gate.

```
for(size_t i=0; i < gate->_fanout.size();i++)
    CirGateV out = gate->_fanout[i];
    CirGateV temp = CirGateV(out.gate(),(out.isInv()!=fin.isInv()));
    fin.gate()->_fanout.push_back(temp);// in_fanout
    for(size_t j=0;j<out.gate()->_fanin.size();j++)
        if(out.gate()->_fanin[j].gate()->getID()==gate->getID())
            CirGateV temp2 = CirGateV(fin.gate(),(out.isInv()!=fin.isInv()));
            out.gate()->_fanin[j] = temp2;//out_fanin
```

It is a function to merge the fain gate to this gate using while a gate fanin is 1 or the same no invert fanin. I use two for loop to do two things. One is fanin gate push back new gate in its fanout; another is fanout gate reconnect its fanin gate to the fanin gate. Because it is so complexed, I set a function to merge gate. It is the same when merging with const0 gate. The optimize is just run all the gates in dfslist, so I think the time complexity is approximately $O(n)$.

# 3. CirStrash

Check all the dfslist gates in hashmap, if in the map we merge the gate; if not, we insert to the map. At the end, we recreate a new dfslist. The time complexity is approximately $O(n)$.

# 4. CirSimulation

```
unsigned maxfail = size_t(pow(I,0.5)*15);
do
    for(int i=0; i<I; i++)
        _totallist[_pilist[i]]->setsim((size_t)(rnGen(1)<<31)+(rnGen(INT_MAX)));
    setallsim();
    simlog();
    setFecGrps();
    //double time = use.clk(1);
    //double standard = (abs(presize - _FecGrps.size())/time);
    //if(standard < 100.0) ++fail;
    if((presize==_FecGrps.size()))++fail;
    presize = _FecGrps.size();
    num++;
while(fail<maxfail);
cout<<"\r\033[K"<<num*64<<" patterns simulated."<<endl;
updateFecGrps();
```

I use 64 bit to store pattern simulate in parallel due to faster than 32bit. When random simulation, I use PI numbers to set the max fail, stalling the simulation. I create the random pattern with rnGen() and I find if the pattern is random enough, it is helpful for the performance in simulation. Most important, I create to methods to calculate fail; one is when the oldGrpsize the same as the currentGrpsize, the other is the change velocicy which I use usage function to record time and set a standard to generate a fail. The above is about the pattern algorithm.

```
for(int i=_FecGrps.size()-1;i>=0;--i)
{
    int s = _FecGrps[i].size();
    HashMap<simkey,FecGrp*> _FecMap(s);
```

About checking whether it is the same in the fecgroup, I use the hashmap. Due to the group will create the max groups number is the gate size, I create a hashmap that buckets number the same gate size. At the end of the simulations I sort the fecgrps and set the fecpos number to every gate.

# 5. Cirfraig

I run the steps according to the dfslist, due to I think the pair gate will close to the gate. When checking whether the gate is need to be solved or merged, I set two flags to record, solveflag and mergeflag. First I run the dfslist only for AIG gate, when the gate is in the zero group it didn't

need to check the solveflag. However in other gates, when the gate see the gate first, we do solve and set the solvegate so that next time we cannot access the gate. We won't solve the gates again. In other words, If we need to merge the gate, we need to set the specified gate ergeflag to insure no gate will merge it and no access to it .

| | |
|---|---|
| ***Group 0 gates*** | ***other gates*** |
| ***If(fanin has mergeflag)*** | ***Else*** |
| ***gate merged with const0*** | ***For(fecgroup)*** |
| ***else If(satsolve is 0)*** | ***If(Groupgate no mergeflag and no solveflag)*** |
| ***gate merged with const0*** | ***If(satsolve is 0)*** |
| ***gate set mergeflag*** | ***Groupgate merged with gate*** |
| ***Else*** | ***Groupgate set mergeflag*** |
| ***StoreCEX*** | ***Else*** |
| | ***StoreCEX*** |
| | ***gate set solveflag*** |

Besides the flags I set, I will do simulation while CEX number==64, to set the difference between FecGroup gates, it may decrease the grps size and run the fraig more efficient. Also, I set a recycle can to accumulate the merge gates pair when some time I will merge them all. It is a method not always merge the gates due to the less performace while if we merge at the end of the fraig it may waste time to run dfslist. As a result, it is important to figure out a tradeoff to have the high performance.

```
typedef pair<CirGate*, CirGate*> mergeGate;
mergeGate temp(_totallist[0],dfslist[i]);
vector<mergeGate> recycle;
recycle.push_back(temp);
    if(res.size()>500)
    {
        fraigmerge(res);
        genDFS();
        i=0;
        res.clear();
        fraigInit(s);
        CirGate::setGlobalRef();
    }
```

```
if(countCEX==64)
{
    fraigmerge(res);
    genDFS();
    setallsim();
    setFecGrps();
    updateFecGrps();
    i=0;
    res.clear();
    countCEX=0;
    fraigInit(s);
    CirGate::setGlobalRef();
    CirGate::setGlobalRef2();
}
```

I merge all the functions and objects above. We can find they do the same things except for the simulation. And the number 500 was from experiment, I will introduce later.

# III. Experiments

1. Compare with ref

| Sim.09 | Ref fraig | | My fraig | |
|---|---|---|---|---|
| | Time | RAM | TIME | RAM |
| Ciropt | 0 | 0.7578M | 0 | 2.168M |
| cirstrash | 0 | 1.117M | 0 | 2.168M |

| Sim.13 | Ref fraig | | | | My fraig | | | |
|---|---|---|---|---|---|---|---|---|
| | pattern | grps | time | ram | pattern | grps | time | ram |
| Cirsim –r | 135816 | 3366 | 11.2 | 17.59 | 49664 | 3468 | 8.79 | 28.71 |
| Cirsim –r | 95488 | 3260 | 7.05 | 17.58 | 24768 | 3344 | 3.71 | 28.71 |
| Cirsim -r | 90624 | 3186 | 6.29 | 17.58 | 22848 | 3246 | 3.45 | 28.71 |

| Sim.13 | Ref fraig | | My fraig | |
|---|---|---|---|---|
| | time | RAM | time | RAM |
| Cirsim –r | 10.95 | 17.72 | 8.45 | 28.18 |
| Cirfraig | 101.6 | 46.77 | 88.72 | 48.5 |

2. Calculate the optimization of maxfail
   Maxfail= size_t(pow(I,0.5)*15)
3. Calculate the optimization of merge time
   if(res.size()>500)
4. Difference of the simulation methods
   Number > Velocity

## Results

In the above I discover that the ref use less memory but in doing fraig and doing simulation my fraig has higher performance.
When calculating the parameters which to set, I use many sim.aag files to decide. And discover that this case has a higher performace, it is a great tradeoff, not more and more bigger or not more and more smaller it has a optimization.
According to considering the time or not, I find the performance is nearly. However there will be a more and more parameters to decide. So I don’t choose the velocity in my fraig

# IV. Discussion

## 1. doing fraig often have a infinite loop

The problem have many cases. Most of the common one is that there are some gates in the dfslist but not in the _FecGrps. They may affect we redo solve in the same gates. Maybe check the gates In small files and print their information in gdb is helpful.

## 2. core dumped

I always encounter this problem due to my less use of the assert. It can help you whether the pointer is exist. The most common case is that the gate is merged but you still use it in _FecGrps, it is due to that the _FecGrps have to realtime with the dfslist and vice versa. Remember set assert !

## 3. double free

My first version is when the gate will be merge, I eliminate it from its group. But I may face the double free because re merge the same gate. As a result, I use two flags rather than just erase it from Grps; I do erase and merge at the same time to ensure preventing from the problems.

## 4. performance

When doing simulation, I firstly use update to change the grp in hashmap but it may seem too waste time. Another problem, is that I use erase the grp we implement in the hashmap. It is too slow to simulate. So I change to recreate the _FecGrps which has a high performance.

When doing fraig, I decrease the SATsolve frequency by checking zero groups. If the group’s fanin is const0, we don’t need to prove the gates but just merged with const0 gate. I think because there are so much gates in group 0, so it may be useful.

## 5. improvement

maybe the fraig can improve by the steps we do. Because the _FecGrps is sorted not just the dfslist sequence, we may pay more time on the SATsolve, so it is a method to do my work.