DSnP: Functional Reduced AIG (FRAIG)

Table of Contents

1. Structure and difference between Hw6

`class CirMgr`

```cpp
class CirMgr
{
private:
   size_t            _M;
   size_t            _A;
   ofstream*         _simLog;
   GateList          _gateList;         // gate pointer
   IdList            _PIList;           // literal ID
   IdList            _POList;           // literal ID
   IdList            _floatFaninList;   // gate ID
   IdList            _unusedList;       // gate ID
   IdList            _DFSList;          // gate ID
/************  newly added data member in fraig  ************/
   bool              _isSimulated;      // false if not simulated yet
   bool              _isFraiged;
   vector<size_t>    _inputs;           // inputs used in fileSim
   vector<size_t>    _outputs;          // used in writeLog
   vector<FecGrp*>   _fecGrps;          // stores FecGrp pointers
   FecHash           _fecFriends;       // (gate ID, FecGrp*)
};
```

In Hw6, the above members were stored in CirMgr.
To determine the difference between inverted and non-inverted, I stored the gates in literal IDs, which are the IDs in the AIGER files. One of the benefits is that there is no need to store an additional boolean vector or store gates as `CirGate*, bool` pairs. Literal IDs alone can indicate the ID of gates and whether they are inverted.
For `_floatFaninList`, `_unusedList`, and `_DFSList`, I decided to store them in gate IDs because there's no need to know whether they are inverted.
The commented lines are members newly added. They will be indicated later in the report.

`class CirGate` and its derived class

C++
```cpp
class CirGate
{
public:
   virtual void simulate() {}
    // several set, get, print functions
protected:
   int                 _gid;               // gate ID
   size_t              lineNum;
   static CirMgr*      _mgr;               // gates connected to current Mgr
   mutable size_t      _ref;               // make setting, resetting O(1)
   static size_t       _globalRef;
   IdList              _faninList;
   IdList              _fanoutList;
/************  newly added data member in fraig  ************/
   unsigned            _pos;               // position in DFSList
   SimValue            _val;               // used in simulating circuit
};
```

Class `CirGate` is implemented nearly the same as in Hw6.
Same as the reasons above, `_faninList` and `_fanoutList` store literal IDs due to its convenience.
However, since storing literal IDs can't gain access to other gates, there is a `static CirMgr* _mgr` stored in `CirGate`. Every gate share the common `_mgr` so declaring it as static should save memory.
Setting and resetting flags by utilizing `_ref` and `_globalRef` can be done in constant time.

There are five derived class from class `CirGate` :
`CirUNDEFGate` , `CirPIGate` , `CirPOGate` , `CirAIGGate` , `CirCONSTGate`
Each of them contains members which are not necessarily needed by others.
They will also be mentioned in detail below.

2. Trivial optimizations

`CirMgr::sweep()`

```cpp
CirMgr::sweep()
{
    if (_unusedList.empty()) return;
    setAllRef();
    for (auto& i : _POList)
        AIGGate->sweep();    // recursive calls CirGate::sweep()
    resetRef();
    resetLists();
};

CirGate::sweep()
{
    if (isActive()) return;
    for (auto& i : _faninList)
        faninGate->sweep();
    reconnectGate();
    delete this;
}
```

Sweep the gates in `_unusedList`.
My approach is to first mark all the gates in `_DFSList`, and then DFS from the unused gates. If the fanin of the current gate is in `_DFSList`, the sweeping operation should be stopped by there.
One of the benefits is that in some cases if there is only a small part of gates to be swept, the operation can be done slightly quicker than iterating through all the gates twice. There is a more efficient approach, but because the time they spent don't differ much, I decide not to implement it. It will be mentioned in the last part. There is no need to update `_DFSList` because it won't change.

`CirMgr::optimize()`

```cpp
CirMgr::optimize()
{
    for (auto& i : _DFSList)
        currentGate->optimize();    // merge if correspond to 1 of the cases
    resetLists();
};
```

Optimize all the gates in `_DFSList`.
There are four cases to be optimize: `a & a = a`, `a & !a = 0`, `a & 0 = 0`, `a & 1 = a`
I determine the cases in the order above. If literal IDs are identical, it belongs to the first case. Otherwise, if gate IDs (literal ID / 2) are the same but the literal IDs are different, it belongs to the second case. The operations are trivial and a bit complicated; therefore, it will not be included in this report.

`CirMgr::strash()`

C++

```cpp
CirMgr::strash()
{
    unordered_map<size_t, CirGate*> hash;
    for (auto& i : _DFSList){
        size_t k = genKey(currentGate);
        if (find k in hash) currentGate.strashMerge(hash[k]);
        else hash[k] = currentGate;
    }
    resetLists();
}

size_t genKey(CirGate* g)
{
    // e.g. fanin of g: 33 24 --> return (24 << 32 + 33)
    size_t smaller = g->smallerFanin, larger = g->larger;
    return (smaller << 32 + larger);
}
```

Strashing is fairly simple. The feature of `CirMgr::strash()` is the key of hash. Instead of making two literal IDs as key, I combine the two into one `size_t` since the fanins are `unsigned`. Taking advantage of this property, there is no need to create another class `Fanin` as key.

Overview of fraig algorithm

The algorithm of fraig is based on the thought that the time complexity of the proving process is proportional to the gate number under a certain gate. To be more precise, the more inputs in the fanin cone of the gate, the longer it takes to prove if it is satisfiable. Therefore, the first we need to do is reduce the gate numbers under a certain gate as small as possible. We can iterate through `_DFSList` and try to prove and merge if the current gate is in a FecGrp. However, if we prove the gate once we find that it is in a fecgroup, things won't be that good. If one gate is at the bottom and its fecparter is at the top, the algorithm won't work, since the satsolver still needs to add the fanin cone of the gate at the top. Accordingly, the order should be reversed.
Therefore, the rule should be like this:

> **When the gate is not the base of its fecgroup, prove and merge it. If it is, skip and continue.**

3. Simulation

`class FecGrp`

C++
```cpp
class FecGrp
{
public:
   // basic set, get functions just like vector
   bool isSingleton();     // return true if _data.size() == 1
   void setBase();         // set the first gate searched in DFSList as base
   void setFECs(FecHash& h);   // insert all (gateID, FecGrp*) into hash
private:
   IdList   _data;
   unsigned _base;
   static CirMgr* _mgr;
}
```

`Class FecGrp` is basically a vector storing literal IDs, however, it includes an additional data member `_base`. This `_base` member will be set as the gate first searched in `_DFSList`. The reason to have this member is that we have to ensure the gate we are merging is at the bottommost of the circuit. In order to get the base of a FecGrp, I choose to store another member in `class CirGate`, which is `_pos`. `_pos` will be updated every time after the simulation process or during fraig. Details will be explained below.

`class SimValue`

C++
```cpp
class SimValue
{
public:
   #define INV 0xFFFFFFFFFFFFFFFF
   size_t operator () () const { return _v; }
   bool     operator == (const SimValue& sv) const
   { return _v == sv._v; }
   SimValue operator & (const SimValue& sv) const
   { return SimValue(_v & sv._v); }
   SimValue operator | (const SimValue& sv) const
   { return SimValue(_v | sv._v); }
   SimValue operator ! () const { return SimValue(_v ^ INV); }

   bool isInv (const SimValue& sv) const { return (!(*this)) == sv; }
private:
   size_t _v;
};
```

Before simulation, we have to create a simulation value (simvalue). Because 64-bit machines are far more popular than 32-bit machines, I decided to make `SimValue` a `size_t`. To make it convenient and clean, I decided to create `class SimValue` and overload several operators. This not only make the code neater and more readable, but also let me code more intuitively. (e.g. `!a & !b` == `!(a | b)`)

`CirMgr::simulate()`

```C++
CirMgr::simulate()
{
    genInput(_inputs);    // fileSim or randSim
    simulateCircuit();    // run over the whole circuit and simulate
   findFECs();           // divide into fecgrps
}

CirMgr::simulateCircuit()
{
   for (auto& i : _PIList)               // plug in simvalues
      PIGate->setSimVal(_inputs[i]);
   for (auto& i : _DFSList)              // iterate and simulate
      AIGGate->simulate();
   for (auto& i : _POList)
      POGate->simulate();
}
```

The first step is to get the inputs and simulate the circuit.
To simulate the circuit, simply plug the simvalues into the PIs. After iterating through `_DFSList` and `_POList`, the circuit will easily be simulated (because it is of post order).

Find FEC groups

Dividing all the gates into different FEC groups requires more thinking.
Generally, we can differ the gates by identifying their simvalues.
If they don't share the same simvalue, they belong to different FEC groups and will be separated forever.
If they share the same simvalue, we should simulate more to identify if they really behave the same.
I store all the FEC groups in one big `_fecGrps`, which is a vector.
The approach I use is as follow:

1. Add all the gates into one FEC group (multiply their IDs by 2).
2. For the first time, iterate through all the gates in the FEC group and separate them into groups.
3. Collecting all new groups and add it back to `_fecGrps`.
4. Simulate the circuit and divide it into more groups. However, this time we take another approach.
5. Go back to 3. until reaching threshold.
6. After reaching threshold, set all the gate to the corresponding FEC group.

For the first step, because it is needed to separate inverted from non-inverted, multiply the IDs by 2 and add it all into a FEC group to obtain a group filled with literal IDs.

C++

```cpp
if (!_isSimulated){
   SimHash newFecGrps;
   for (auto& i : firstgrp){
      unsigned x = firstgrp[i];
      SimValue sv = getGate(x)->getSimVal();
      if (newFecGrps.find(sv))
          newFecGrps[sv]->add(x);
      else if (newFecGrps.find(!sv))
          newFecGrps[!sv]->add(x + 1);
      else newFecGrps.insert(sv, new FecGrp(x));
   }
   collectGrps(newFecGrps);
   addNewGrps();
   _isSimulated = true;
}
```

The code above is what we should do for the first time we seperate FEC groups.
Search for the non-inverted and the inverted simvalue of a gate in the hash. If it is found, add the gate to the FEC group. Otherwise, create a new group with the simvalue being the key and add it to the hash. After this operation, we get a bucket of FEC groups having different simvalues among different groups, but share a common one within the group.

Collect all the groups and add it all back to `_fecGroups`. However, don't add the ones with only one gate, for it will never be in the same group with others afterwards.

```cpp
for (auto& i : _fecGrps){
    FecGrp grp = *_fecGrps[i];
        SimHash newFecGrps;
        for (auto& j : grp){
            unsigned x = grp[j];
            SimValue sv = getGate(x)->getSimVal();
            // x is odd, find inverse only
            // if found, add x with x being odd (change x at collect phase)
            // if not found, create new entry with simvalue inverted
            if (x % 2){
                if (newFecGrps.find(!sv))
                    newFecGrps[!sv]->add(x);
                else newFecGrps[!sv] = new FecGrp(x);
            }
            // x is even, find directly
            // if found, add x with x being even (no need to change)
            // if not found, create new entry with simvalue un-inverted
            else{
                if (newFecGrps.find(sv))
                    newFecGrps[sv]->add(x);
                else newFecGrps[sv] = new FecGrp(x);
            }
        }
        collectGrps(i, newFecGrps);
    }
    addNewGrps();
}
```

When dividing FEC groups for the second time and after, we have to apply another method.
If the literal ID in the group is odd, find the inverse of its simvalue only. Add it into the group if found or add a new entry if not. If the literal ID is even, directly search for its simvalue in the hash. Add it in if found or create a new entry otherwise.
This way, although we have to invert the "odd" cases and add it in, the result will be correct. Because the inversion only indicates the difference between gates and does not affect the results.

Same as the above, collect all the groups and add them all back to `_fecGrps`.

After reaching the threshold, insert all the `(gateID, FecGrp*)` pair into a hash to make the searching time complexity constant time. This ends the simulation process.

3. Fraig

As mentioned above, the fraig process utilizes the property that the gate we are merging will have the least possible fanin cone. We can observe this by going through a circuit once.

1. If a gate is a base of a group, there is no need to merge because merging now might end in a large fanin cone and result in huge proving time. So skip it.
2. If a gate is in a group and is not the base of a group, merge it. If the gates are proven to be identical, the gate must be merged and `_DFSList` has to be updated. This process can also reduce gates if updating `_DFSList` make some gates unused. If some gates become unused, erase it from the FEC group it belong if it has one.

By following these two rules, it can be seen that the fanin cone of the gate we are merging will be the smallest possible. There must be no more gate to be merged in the fanin cone. Now, let's go through the details.

`CirMgr::fraig()`

C++

```cpp
void CirMgr::fraig()
{
    SatSolver s;
    for (auto& i : _DFSList){
        // continue if doesn't belong FEC group
        if (_fecFriends.find(_DFSList[i])) continue;
        FecGrp grp = _fecFriends[_DFSList[i]];
        // continue if itself is the base
        if (grp.getBase() == _DFSList[i]) continue;
        CirGate *base = getGate(grp.getBase()),
                *g = getGate(_DFSList[i]);
        // build model, assume property ... etc.
        initSat(s, base, g);
        if (!s.assumpSolve()){
            g->fraigMerge(base);
            i = base->getPos() - 1;
        }
        else reSim(s);
    }
    resetLists();
    endFraig();
}
```

Following the rules above, we can obtain this piece of code.
After simulating, we can start to prove if the gates in the same FEC groups are identical.
In the `initSat` part, I choose to initialize and add all the gates to the solver every time. After testing, I found that initializing once after proving 1000 times will take longer than initializing and adding the gates every time. The process of adding the gates is simple: iterate through the fanin cone of the two gates by post order and set flags, just like setting `_DFSList`.

If two gates are proven to be identical, merge the gates, reset `_DFSList`, and the remove gates that become unused from the FEC groups they belong. We can do both by iterating the circuit once.

After merging, I decide to go on proving the gates from the base that has just been merged. `_DFSList` has been changed and there might be something new that could be merged after the base.

The reason why I don't rerun the circuit from the first one is that removing the gate at the upper part of the circuit won't affect the bottom part. Maybe some of the gates at the bottom part will become unused, but those gates could be swept by `CirMgr::sweep()` and has no need to go through again.

On the over hand, if the gates are proven to be different, we can get the pattern that differ the two gates and resimulate the circuit by plugging those into the corresponding PIs. This is the same process as `CirMgr::simulate()` mentioned above and thus will not be mentioned again.

At the end of the fraig operation, the `endFraig()` method will reset all the variables back to original (e.g. `_isSimulated` to false, SimVals of gates = 0, delete `_fecGrps[i]`, clear hash from gateID to FecGrp* ... etc). However, I will set `_isFraiged` to true, since after fraiging without skipping any cases the circuit should be simplest (except for the `CirMgr::optimize()` operation because the PIs won't be fraiged in `CirMgr::fraig()`).

That's all for my algorithm and implementation for fraig. :)

Performance

`cirSIMulate -r`

My Code

testcase	Memory	Time
sim07	2.391 M	0.02s
sim09	1.121 M	0.03s
sim10	0.457 M	0.01s
sim12	2.895 M	0.23s
sim13	19.07 M	4.01s

Reference Code

testcase	Memory	Time
sim07	2.391 M	0.02s
sim09	1.145 M	0.07s
sim10	0.539 M	0.01s
sim12	3.43 M	0.52s
sim13	17.91 M	8.71s

`cirFraig`

My Code

testcase	Memory	Time
sim07	2.656 M	7.54s
sim09	1.397 M	0.97s
sim10	0.7422 M	0.15s
sim12	4.379 M	5.57s
sim13	20.07 M	56.64s

Reference Code

testcase	Memory	Time
sim07	7.988 M	10.68s
sim09	3.047 M	0.26s
sim10	0.9492 M	0.05s
sim12	6.98 M	3.23s
sim13	145.05 M	98.35s

Improvements

1. The sweeping method indicated above is as follow:
    - no need to go through the circuit and mark the gates in `_DFSList` first
    - start sweeping from the unused gates

        - if the fanin of the gate has zero fanouts after deleting current gate, add it to unused gate
    - keep sweeping until `_ususedList` is empty

2. The simulation process takes some time to reassign and inserting the `(gateId, FecGrp*)` pair into the hash. This can be improved by modifying the `addNewGrps()` method.