Data Structure and Programming

Final Project

Functionally Reduced And Inverter Graph (FRAIG)

Student ID: B00901086
電機系三年級 林宏驊
Email: b00901086@ntu.edu.tw
Mobile phone number: 0912881828

0. Overview

This is a program that deals with combinational “And-Inverter-Graphs”. Given an and-inverter-graph circuit specified in a file in aag format, this program can read in the circuit and construct the circuit model. Simulation and simplification/optimization can then be performed on the circuit. Simulation output and the simplified/optimized circuit can be written into files specified by the user.

1. Commands and functions

There are several commands performing different task on an AIG:

CIRRead

2. Functionality Explanation

And-Inverter-Graph

An and-inverter-graph is a logic circuit that is composed solely of two input AND gates with possibly inverted fanins and/or fanouts (called AIG gates) along with the circuit's input and output ports (PIs and Pos), and possibly a constant gate (CONST0). A more general and-inverter-graph may contain latches, but this program only handles combinational circuits.

The program can perform several checks to simplify/optimize the circuit, which means merging functionally equivalent gates. Gates being “functionally equivalent” means that their output are always the same regardless of the input pattern. Simplification and optimization are performed on the AIG gates only. That is, only AIG gates would be removed. The input ports PIs, POs, and CONST0 gate are left intact.

Sweep

Some of the AIG gates cannot be reached by any POs. In other words, there does not exist any signal path so that its output signal can reach any PO, and hence its output value does not affect the output of the circuit. Such gates can therefore be eliminated from the circuit.

Trivial optimization

The outputs of some AIG gate do not depend on the input pattern or directly follows its fanins. There are several possibilities:

An AIG gate with identical fanins

An AIG gate having CONST0 in inverted phase as one of its fanins

An AIG gate with fanins from identical gate in opposite phase

An AIG gate having CONST0 as one of its fanins

In the first and second cases, the output of the AIG gate directly follows that of the fanin (or the other can in in the second case). In the third and fourth cases, the output is 0 regardless of the PI signals. Such gates can be found out solely by checking the fanins of each AIG gate in the circuit without comparing with others and can be trivially replaced with its fanin or CONST0.

Structural hashing

A group of gates with identical fanins are equivalent and can thus replace each other. Structural hash identifies such groups in the circuit and choose one of them to replace the others.

Fraig and simulation

There may remain other functionally equivalent pairs/groups that cannot be identified. Given a

proposed pair of gates, an SAT (Boolean Satisfiability) solver can determine whether they are equivalent or not. Instead of proving every possible pair, note that functionally equivalent gates have the same output value regardless of the input pattern (feeding the PIs). Therefore, we can feed various input pattern to the circuit and perform simulation. Those that come out with different simulated value are guaranteed not to be functionally equivalent to each other, and by comparing the simulation value, the circuit can be split into many functionally equivalent candidate groups with much smaller sizes than the entire circuit. SAT solver is then called to determine the equivalence within each group. This can save the saving effort of the SAT solver, which is often quite large.

3. Implementation Explanation

Gates and circuit model

There is a pure virtual base class CirGate providing generic functions and storing generic data such as gate ID, fanins, fanouts, simulation value, symbol etc. The gates are linked via data member fanins and fanouts, whose type is pointer wrapper class containing the information of invertion. AigGate, PIGate, POGate, CONST0Gate and UndefGate are classes derived from class CirGate implementing their own simulation function, which must be different, among other things.

_AllList: random access through gate ID

A circuit manager is in charge of maintaining the entire circuit model (containing many instances of gates) and all the simplification/optimization task discussed in the previous section. Within an object of the manager class CirMgr, there is an array (vectors actually) holding each kind of gates: AIG, PI, PO, UNDEF and CONST0, so that they are not newed one by one and scatter in the memory. The gates are stored inside these arrays. However, they are not accessed directly through these arrays. There is an array of pointers, AllList, with the ith pointer pointing to the gate with the gate ID i. If such a gate does not (or no longer) exists, the pointer is set to a NULL pointer. The gates can thus be randomly accessed through the pointer array by their IDs in constant time. Subsequent deletion of gates is performed by setting the corresponding pointer in AllList to NULL, and would be considered to be extinct from then on. The gate object itself is left intact.

_DFSList: keeping the "fanin first" order

A post-order depth-first-search list starting from the POs, _DFSList, is stored in the manager and maitained whenever the circuit is modified. The output value of a gate cannot be determined before that of its fanins are determined, so many operation on the circuit (simulation, for example) have to be done on each gates in an order such that no gates are manipulated before its fanins. The order of the _DFSList matches such requirement, and it useful in many operations. Of course, a depth-first traversal has to be performed each time the circuit is modified to maintain the _DFSList.

Sweeping

Those that can be reached by the POs must be inside the _DFSList. Therefore, simply remove those not contained in the _DFSList and update the fanouts of the affected gates.

Trivial Optimization

Order matters: post order DFS

Note that the information need to determine whether a gate can be trivial optimized is their fanins. It depends on nothing else. Therefore, we can just iterate through the whole circuit and check for all the gates. However, order matters. Optimization of the fanins of a gate may make a previously not optimizable gate optimizable. On the other hand, the optimization of the fanouts of a gate does not affect anything. Therefore, a gate should not be checked for optimization before its fanins, and we can do this by iterating through the _DFSList.

Structural hashing

Order matters: post order DFS

In this feature, we need to identify gates with identical fanin, taking permutation into consideration (the order of fanins does not matter). As in trivial optimization, the fanins of a gate need to be checked before itself.

Hash: ideally constant time identification of equivalence

Instead of checking every possible pair, we use hash datastructure. The hash key is the two fanins of the AIG gates, and by the hash function the gates are hashed into buckets of the hash. We iterate through the _DFSList and check/insert the gates to the hash to preserve the "fanin first" order. Gates with identical fanins would be hashed into the same bucket, and can then be identified and merged. For a balanced hash, identify such an equivalent gate would be constant time, and the over complexity would be linear, in contrast n square of the brute force method.

For the hash to be balanced, the number of buckets is usually made to be a prime number, and in this implementation it is performed by a table lookup provided in the reference code util.cpp.

Simulation and Functionality Equivalent Candidate (FEC) identification

Parallel simulation and all gate simulation

For the simulation part, it is parallel simulation, that is, pack the input bit pattern in groups of 32 to unsigned intergers and perform bitwise operation on them. All gate simulation is used, since the parallel input patterns should be very versatile, so that the output value is very likely to change, and the simulation operation (bitwise operation) is not very time consuming. Hence the overhead of the event driven simulation may not pay off. Here, iterate through the _DFSList and perform corresponding logic function of the gate (e.g. AND for an AIG).

After the simulation, we have to split the gates into groups by their simulation value. This is similar to the structural hashing part. Here, we also use the hash data structure, with the simulation value as the key.

FEC identification after first simulation

Note that in addition to having identical output regardless of the input pattern, a gate may also be "inversely" equivalent to another gate, that is, they are all ways the inverse of each others no matter what. Therefore, the equivalence of hash key in this key is determined by checking if the simulation values are identical "or" totally inverted. In both cases, the two gates are potentially equivalent.

Subsequent simulation

If we perform subsequent simulation, since gates within one group may not all be equivalent, distinct simulation value might come up, and the group can be further split. Also note that gates that are split cannot be recombined, since there already exist some input pattern to distinguish them and they cannot be equivalent. Therefore, we need only consider the regrouping, or hashing, within each group itself.

FEC identification for subsequent simulation: different

The situation is a bit different in the group splitting in subsequent simulation. Before any simulation is performed, a gate may be equivalent or inversely equivalent to any other. However, after the first simulation, the simulation value within each group determines the one possibility of the phase of potential equivalence. However, two gates cannot be "equivalent" and "inversely equivalent" to each other at the same time. The possibility of one of them eliminates that of the other, Therefore, the phase of equivalence if then determined and determined for sure.

Hence, for FEC group identification of subsequent simulation, the equivalence of hash key (the simulation value) needs to be: "exactly identical, considering the phase of potential equivalence", which is different from the first simulation and must be noticed.

FEC storage: manager keeps all, gate keeps the one it belongs to

The FEC groups are stored in the manager object. Also, the corresponding id of the FEC group is stored in each gate object, so that we can get the FEC peers of each gate, without searching among all the FEC groups.

Fraig

This part is to invoke SAT solver to determine whether each FEC pairs are equivalent or not.

Order: post order DFS

The circuit manager iterates through the _DFSList. For each gate encountered, here called the "leader" gate, get the FEC group it belongs to, and determine the equivalence of this gate with every other members within that group.

Equivalence (UNSAT) handling

Equivalent gates are removed from the FEC group and stored together with the leader gate for future merging. When merging, the leader persists, replacing others, for the leader should be the first one in the _DFSList and this would avoid a gate merging another within its fanin cone and result in a loop.

Non equivalence (SAT) handling

Non equivalent gates are left within the group, waiting for one of them visited by the DFSList traversal to prove the equivalence with the others. Also, the input counter examples are collected for further simulation.

Midway simulation

The circuit manager performs further simulation and FEC identification if 32 input patterns are collected from the SAT solver. The FEC groups and the fecGrpId stored inside each gate indicating which group it belongs to would also be updated. Note that such simulation may occur midway when the proving operation of one FEC group is progressing. At this moment, there are three kinds of gates, those equivalent with the leader, those that are not, and those not yet proved and hence unknown. The equivalent ones are to be merged by the leader and can be safely removed. However, the unknown gates may be equivalent to the leader, to somebody among them, to somebody among the non-equivalent ones, or no one. Therefore, the leader, the unknowns and the traitors (non equivalent to the leader) must still be packed together for correct identification of FEC groups. Fortunately, the input patterns are the counter examples proving the non-equivalence between the leader and the traitors, so those two are still going to be split. Therefore, the leader, unlike its followers (equivalent ones), is left within the group until everyone else is proved equivalent or non-equivalent with it.

Gate merging

The merging operation would be done if 32 groups up for merging are collected, and the gates would be merged by their leader. DFS traversal would be operated after that since the circuit is modified. Those isolated from the DFS list is then swept out of the circuit. Also, those removed from the circuit need also be removed from their corresponding FEC groups. After this, the manager starts it all over again from the beginning of the DFS list, until no FEC group is left.

4. Performance consideration and prediction

Circuit model

Random access

The _AllList, which enables random access via gate ID, greatly enhances speed, especially at the circuit construction stage, where gates are interconnected via the ID of fanins and/or fanouts.

An interface

Also, _AllList provides an interface for the manager and masks the actual state of the gates. For example, if the pointer to gates is used to access gates instead of IDs, a removal of gate may not be easily broadcast to all the places still holding that ID, since every information within that gate is lost. However, if the ID is used instead, a removal would be just to set the pointer in _AllList to NULL, and future reference to such ID would get the message that the gate is no longer present. This improves consistency.

Sweeping and Optimization

Those are trivial operations, linear in circuit size (number of gates), and is simply iterating the circuit. They are not time consuming.

Structural hashing

The complexity is ideally linear, and the actual performance is determined by the hashing function. The hash key itself, fanins, is usually not seriously biased, and ordinary hashing function may yield satisfiable result. Further enhancement of hash function is not implemented in this program.

Simulation and FEC identification

Simulation by itself is linear in time, iterating through the _DFSList, and hence fast. It is the speed of FEC identification that determines the performance.

The FEC identification is also hashing. A well-balanced hashing function should enhance the performance. Also, since this operation deals with splitting of groups, there are a lot of copying operation, whne one FEC group is split into many. Minimizing copying operation should also be an issue. This is not implemented in the program.

In random simulation, the input patterns are randomly generated, so the generated pattern should be as distributed as possible, so that non equivalent pairs are more likely to be identified.

Fraig

Friag is followed by simulation, so there should be a balance between the effort of the two. In random simulation, we set a max fail time, the time that a simulation cannot identify any new FEC groups, and it is determined mostly by the number of PIs.

Order of proof: DFS is for avoiding loop creation, not for performance

Observation of the execution indicates that SAT solving is the most time-consuming process in fraig opration. Others such as merging, simulation etc are usually a blink of an eye compared to the SAT solving. The SAT solving speed cannot be further enhanced too much, given that it is an NP problem, so the proof effort should be minimized by carefully choosing the order of prooving, so that the easier ones are proved first, and the difficult ones are left for later when considerable simplification is performed, reducing the effort that might be required if it is countered at the beginning. By proving in the order of the DFS list, many of the gates are proved later then their fanins, which may reduce the effort. However, the order in DFSList does not correspond directly to depth and some rather easy proof may be postponed to latter operation, a kind of waste. The choice to prove by DFS order is because of ease of implementation, avoiding a gate merging another in its fanin cone, producing loop. It is not for performance, and price may be paid. \

Proof effort limiting

It is also be worthwhile to give up some proof if it has already consumed too much time and move on to the next, since subsequent simplification may reduce the effort required, and come back to it some other time later, since such NP-complete problem can be very time consuming. However, this would require further knowledge of the SAT solver and is hence not implemented in this program.

5. Result

Fraig, including simulation and FEC group identification

The performance of the fraig operation along with the preceding simulation is summarized here. The other operations are very fast compared to fraig and is not listed here.

Simulation by pattern file

Sim??.aag	Simulated pattern	Simulation time (s)	Fraig time (s)
6	34	0.02	0.55
9	1920	0.06	0.38
10	896	0.02	0.05
12	11936	0.64	106
13	22912	6.78	89.41

Random simulation

Sim??.aag	Simulated pattern	Simulation time (s)	Fraig time (s)
6	192	0.03	0.68
9	2048	0.05	0.4
10	640	0.02	0.05
12	2208	0.16	106
13	22912	6.78	89.41

Discussion

The major problem surfaced at sim12.aag. This is a circuit composed of two functionally equivalent part, and would be reduced to CONST0.

Proof order of the fraig

In the execution, the program is stuck in proving the equivalence/non equivalence of one pair of gates. Evidently, the proving order of fraig need to be tuned. This does not occur in the largest circuit available, sim13.aag, which is not a CONST0 circuit. It seems that proving equivalence, or UNSAT, can be more time consuming. This can be expected, since proving SAT requires only one counter example, and this counter example may be easy to find if lucky. However, proving UNSAT need to ensure that there is “no” such pattern, and is reasonable to take more time on average. It is expected that this program would hang if two large equivalent circuits “mited” together is fed in.