ABC basics (compilation from different articles)

1. AIG construction
2. AIG optimization
3. Technology mapping

1. BACKGROUND

An *And-Inverter Graph* (AIG) is a directed acyclic graph (DAG), in which a node has either 0 or 2 incoming edges. A node with no incoming edges is a primary input (PI). A node with 2 incoming edges is a two-input AND gate. An edge is either complemented or not. A complemented edge indicates the inversion of the signal. Certain nodes are marked as primary outputs (POs). Registers if present are considered as PI/PO pairs.
The combinational logic of an arbitrary Boolean network can be factored [4] and transformed into an AIG using DeMorgan's rule. Structural hashing is applied during AIG construction to ensure that no two AND gates have identical pairs of incoming edges. A *cut C* of node *n* is a set of nodes of the network, called *leaves*, such that each path from PIs to *n* passes through at least one leaf. A cut is *K-feasible* if the number of leaves does not exceed *K*. The *cut function* is the function of node *n* in terms of the cut leaves. Two Boolean functions, *F* and *G*, belong to the same NPN-class (are *NPN-equivalent*) if *F* can be derived from *G* by negating (N) and permuting (P) inputs and negating (N) the output. *Example*. Functions $F = ab + c$ and $G = ac + b$ are NPN equivalent because swapping *b* and *c* make them identical. Functions $F = ab + c$ and $G = ab$ are not NPN-equivalent because no amount of permuting and complementing variables can make a 3-variable function equivalent to a 2-variable function.

2. AIG Construction

AIGs for Boolean functions can be constructed starting from different functional representation:
SOP: Given an SOP representation of a function, it can be factored [3] and the factored form can be converted into the AIGs. Each two-input OR-gate is converted into a two-input AND-gate using the DeMorgan rule.
Circuit: Given a circuit representation of a (multi-output) Boolean function, the (multi-output) AIG is constructed in a bottom-up fashion, by calling a recursive construction procedure for each PO of the circuit. When called for a PI node, the procedure returns the elementary AIG variable. Otherwise, it first calls itself for the fanins of a node and then builds the AIG for the node using the factored form of the node. When an AIG is constructed from a circuit, the number of AIG nodes does not exceed the number of literals in the factored forms. When the AIG is constructed from a BDD, the number of AIG nodes does not exceed three times the BDD number since each MUX can be represented using three ANDs. It follows that the size of the AIG is proportional to the size of the circuit or BDD. Quantifications performed on AIGs have an exponential complexity in the number of quantified variables because quantifying each variable is done by ORing the cofactors and can potentially duplicate the graph size. Except for quantification, Boolean computation is more robust with AIGs than with BDDs. This is because Boolean operations on AIGs lead to the resulting graphs whose size is bounded by the *sum* of the sizes of their arguments, while in the case of BDDs the worst case complexity of the result is equal to the *product* of the sizes of the arguments.

2.1 Structural Hashing

Structural hashing (*strashing*) of AIGs introduces partial canonicity into the AIG structure. When a new AND-gate is added to the graph, several logic levels of the fanin AND-gates are mapped into a canonical form. Although the resulting AIG is not canonical, it contains sub-graphs, which are canonical as long as they have less than the given number of logic levels.
No strashing: When an AIG is constructed without strashing, AND-gates are added one at a time without checking whether an AND-gate with the same fanins already exists in the graph.
One-level strashing: When a new AND-gate is added, checks is performed for a node with the same fanins (up to permutation).
Two-level strashing: In the pre-computation phase, all two-level AND-INV combinations are enumerated and, for each Boolean function realizable by a two-level AIG, one representation is selected as the representative one. In the AIG construction phase, when adding a new AND-gate, the canonical form of the two-level AIG rooted in this gate is constructed, which may require building new AND-gates for the fanins. A detailed discussion of two-level structural hashing can be found in [9][13]. An efficient implementation runs in time linear in the number of AIG nodes. The resulting graphs may have 5-10% fewer nodes, compared to one-level strashing. A drawback of two-level strashing is that when multiple AIGs are constructed repeatedly, it leads to an increase in the number of unused nodes in the AIG manager, which in turn leads to the need to perform repeated garbage collection. This may slow down some AIG-based applications, such as image computation.

2.2 Redundancy Removal

AIGs have been applied as a circuit representation in combinational equivalence checking (CEC) [13] and an object graph representation in technology mapping [15]. In both cases, AIGs are built initially using strashing, and later optionally postprocessed to enforce functional reduction. If [18] AIGs are used for unbounded model checking, in which both the circuits and interpolants computed from the unsatisfiability proofs are represented by AIGs. This work recognizes the need for functional reduction ([18], Section 3.2, paragraph 1) noting that AIGs tend to have many redundancies not captured by strashing. Two procedures have been proposed to perform functional reduction. BDD sweeping [13] constructs BDDs of the AIG nodes in terms of the PIs and intermediate "cut-point" variables. BDD construction is controlled by resource limits, such as a restriction on the BDD size. Any pair of AIG nodes with the same BDD is merged, and the fanout cones are rehashed. As long as all BDDs can be built within the resource limits, the result is a FRAIG. The second procedure, SAT sweeping [14][16], achieves the same merging and propagation by solving a sequence of incremental topologically-ordered SAT problems designed to prove or disprove the equivalence of cut-point pairs. The candidate pairs are detected using simulation. In both approaches, the initial graph is constructed in a redundant form followed by functional reduction applied as a post-processing step.

Another approach to CEC was developed using NAND graphs [7] but the authors do not discuss what methods are used to perform functional reduction or how they prove the equivalence of the output functions represented using NAND graphs.

2.3 Implementation Details

This section discusses the details of the FRAIG implementation.

a) Simulation

The performance of the proposed algorithm critically depends on the efficiency of simulation. The larger are simulation vectors, the better is their distinguishing power and the fewer SAT-based equivalence tests are needed. In the current implementation,approximately 4000 random bit-patterns are used for random simulation. The simulation runtime is typically about 10% of the SAT solver runtime. The memory overhead for storing simulation information is about 0.5K per node. This memory is allocated independently from the memory used for the AIG nodes. When the FRAIG is constructed, the simulation memory can be de-allocated and re-used by the application. Another way of increasing the efficiency of simulation is using the counter-examples returned by the SAT solver during unsuccessful equivalence tests. As pointed out in [13], these counter-examples distinguish functions, which are hard to distinguish by random simulation. In the current implementation, random simulation is performed when a node is first constructed. To use the SAT solver feedback, we re-simulate the AIG periodically, each time 32 new counter-examples are accumulated.

b) SAT Solving

For efficiency, the algorithm requires tight integration of the circuit-based AIG data structure and a SAT solver. The solver used in the implementation is MiniSat [8], with modifications to restrict incremental SAT solving to a subset of variables and clauses. The CNF for the AIG is loaded in the SAT solver incrementally, by adding three CNF clauses each time a new AIG node is created. Checking functional equivalence for AIG nodes n1 and n2 is performed as follows: (1) collect the AIG nodes in the union of the transitive fanin cones of n1 and n2; (2) set the "branchable" variables to be those corresponding to the above AIG nodes; (3) run the solver to prove or disprove equivalence. Incremental runs of the SAT solver create learned clauses, which are stored in the global clause database. Because the logic cones of different equivalence checking problems often overlap, the learned clauses are shared and reused, which improves the performance of the SAT solver.

c) Handing Functionally Equivalent Nodes

In Figure 4, when a new node is found to be functionally equivalent to the old node, the new node can be garbage collected. However, in the current implementation, the new node is left in the graph as a node without fanouts. The node is stored in the list of equivalent nodes, and from the node, we have the pointer to the representative of the equivalence class. Keeping the equivalent nodes around works as a "structural record" of equivalences proved, similar to the computed table in the BDD package. If we hit the same structure again, we look at the pointer to the representative node, and return this representative immediately, without going through the potentially expensive equivalence test. Saving the functionally equivalent nodes is also beneficial for some applications discussed in the following section.

3. AIG REWRITING

Rewriting is a fast greedy algorithm for minimizing the AIG size by iteratively selecting AIG subgraphs rooted at a node and replacing them with smaller pre-computed subgraphs, while preserving the functionality of the root node. Our rewriting algorithm is developed by extending the prior work [3] as follows:

• Using 4-feasible cuts instead of two-level subgraphs.
• Restricting rewriting to preserve the number of logic levels.
• Developing several variations of AIG rewriting to
    o selectively collapse and *refactor* [4] larger subgraphs,
    o *balance* AIGs using algebraic tree-height reduction [8].
• Experimental tune-up for logic synthesis applications.

For the purposes of 4-input AIG rewriting, all 4-feasible cuts of the nodes are enumerated using the procedure in [17]. For each cut, the Boolean function is computed and its NPN-class is determined by hash-table lookup. Fast manipulation of 4-variable functions is achieved by representing them using truth tables stored as 16-bit bit-strings. Altogether there are 222 NPN equivalence classes of 4-variable functions [15], of which only about one hundred appear more than once as functions of 4-feasible cuts in the numerous benchmarks tested, and only about 40 of these have been found experimentally to lead to improvements in rewriting. The unifying characteristic of the useful NPN-classes of functions is that they are decomposable using simple disjoint-support decomposition [2].

All non-redundant AIG subgraphs of the representative functions of the useful equivalence classes are pre-computed in advance as a shared DAG containing approximately one thousand nodes and hashed by the truth table. This DAG is compiled into the program as an integer array, which noticeably reduces the setup time of the rewriting package.

Figure 1 shows the AIG rewriting procedure. The nodes are visited in a topological order. For each 4-input cut of a node, all pre-computed subgraphs of its NPN class are considered. Logic sharing between the new subgraphs and nodes already in the network is determined. First, the old subgraph is dereferenced and the number of nodes, whose reference counts became 0, is returned. These nodes will be removed if the old subgraph is replaced. Next, a new subgraph is added while counting the number of new nodes and the nodes whose reference count went from 0 to a positive value. These nodes will be added. The difference of the counters is the gain in the number of nodes if the replacement is done. The new node is de-referenced and the old node is referenced to return the AIG to its original state. After trying all available subgraphs for the given node, the one that leads to the largest improvement at a node is used. If there is no improvement and "zero-cost replacement" is enabled, a new subgraph that does not increase the number of nodes is used.

Example. Figure 2 shows three AIGs for $F = abc$ that are precomputed and stored. Figure 3 shows two instances of AIG rewriting. The upper part of the figure shows the situation when Subgraph 1 is detected and replaced by Subgraph 2. The lower part of the figure shows two nodes AND(a, b) and AND(a, c) that
are already present in the network. In this case, Subgraph 2 can be replaced by Subgraph 1. In both cases, one node is reduced.

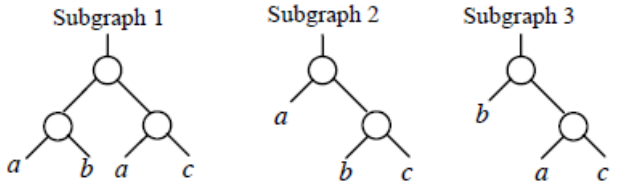

Figure 2. Different AIG structures for function $F = abc$.

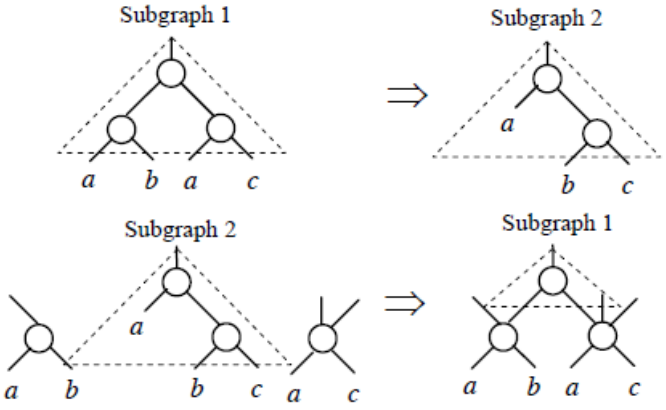

Figure 3. Two cases of AIG rewriting of a node.

A variation of AIG rewriting called refactoring uses a heuristic algorithm [12] to compute one large cut for each AIG node. Refactoring tries to replace the current AIG structure of the cut by a factored form of the cut function. The change is accepted if there is an improvement or no increase in the number of nodes.

4. Delay Optimization using SOP Balancing

4. 1 AND Balancing

AND-balancing of an AIG is a well-known fast transform that reduces the number of AIG levels. AND-balancing is performed in two steps: covering and tree-balancing. The covering step identifies large multi-input ANDs in the AIG by grouping together two-input ANDs that have no complemented attributes in between and no external fanout, except possibly at the root node of each multi-input AND. The covering step is illustrated in Figure 3.1.1. The circles stand for two-input ANDs and the small bubbles on the edges stand for the complemented attributes. The tree-balancing step decomposes each multi-input AND into two-input ANDs while trying to reduce the total number of AIG levels. As the result of this step, a new structure of two-input ANDs is created. This structure is constructed to minimize the delay while taking into account logic levels of the inputs. The tree-balancing step is illustrated in Figure 3.1.2. It should be noted that the covering step is unique, while the tree-balancing step is not unique and depends on the grouping of the inputs with equal delay, while transforming multi-input ANDs into trees of two-input ANDs. Because the covering step stops at the multiple-fanout nodes, AND-balancing cannot increase the total number of two-input AND nodes. However, some nodes can be reduced when AND-balancing is applied to a large AIG and logic sharing is created in the process.

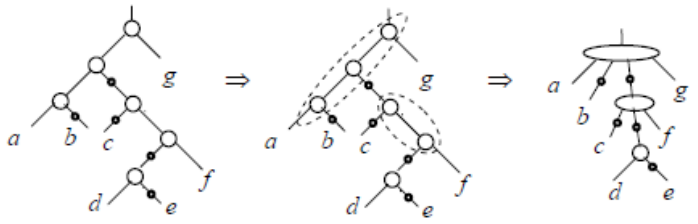

Figure 3.1.1: Illustration of the covering step.

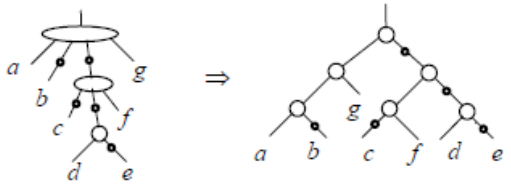

Figure 3.1.2: Illustration of the tree-balancing step.

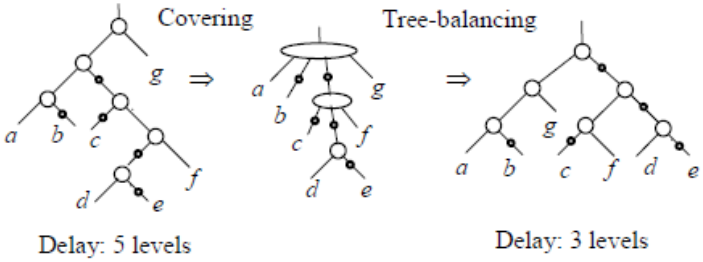

Figure 3.1.3: Illustration of AND-balancing.

Figure 3.1.3 illustrates AND-balancing, which combines covering and tree-balancing. In the above figures, the delays of the PIs are assumed to be 0. The total delay of the AIG in this example is reduced from 5 to 3 levels. AND-balancing described in this section is implemented in ABC [1] as command balance.

4.2 SOP-balancing

In this paper, an AIG is considered small if it depends on roughly 10 or less inputs. A small AIG can be converted into an SOP, and then AND-balancing can be applied to each product and the sum. In doing so, the products and the sum are treated as multi-input ANDs and decomposed to minimize the delay of the output node.

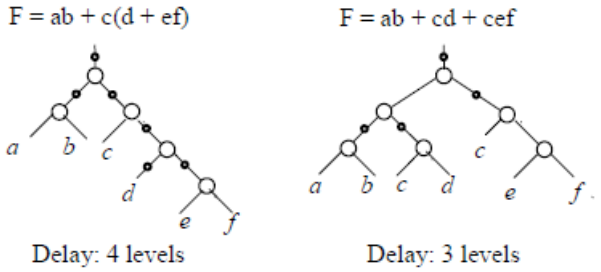

Figure 3.2.1: Illustration of SOP-balancing.

Figure 3.2.1 illustrates SOP-balancing for a small AIG, where the delays of the PIs are equal to 0. The total delay of the AIG in this example is reduced from 4 to 3. Note that AND-balancing cannot reduce the delay in this example. Figure 3.2.1: Illustration of SOP-balancing. In general, AND-balancing is limited to multi-input ANDs, while SOP-balancing looks at larger functions. As a result, in many cases, SOP-balancing can reduce delay when AND-balancing cannot.

A large AIG, representing combinational logic of an industrial design, can contain millions of AIG nodes. It is impossible to apply SOP balancing to such an AIG as a whole, but it is possible to break it down into parts, try SOP-balancing for each part, and if the delay is improved, locally update the large AIG with the structure derived by SOP-balancing. The latter is, in essence, the SOP-balancing algorithm described in this paper.