Scalable Logic Synthesis using a Simple Circuit Structure

Alan Mishchenko Robert Brayton

Department of EECS
University of California, Berkeley
Berkeley, CA 94720

{alanmi, brayton}@eecs.berkeley.edu

Abstract

This paper proposes a resurgence of rewriting and peephole optimization. However, instead of structural matching and rule-based synthesis used in the classical approach, the proposed local transformations rely on efficient modern techniques, such as precomputation, reconvergence analysis, cut enumeration, Boolean matching, exhaustive simulation of small logic cones, and local resource-aware decision procedures based on Boolean satisfiability. The new transformations are tuned to work on a simple homogeneous logic representation composed of two-input ANDs and inverters (*AIGs*). *The result is a logic synthesis flow that is orders of magnitude faster than traditional ones and applicable to large industrial netlists with millions of gates.*

1 Introduction

The use of local transformations in logic synthesis is not new. Some of the first logic synthesis methods were based on local transformations [11]. In these methods, optimization started by representing the netlist in terms of the gate types to be used in the final implementation. Then, *peephole optimization* was applied. A set of gates in a peephole was recognized as a known graph structure and looked up in a hash table. Associated with the structure was a set of transformations that could improve area or delay. The decision to "fire" a local transformation and the order of firing them was controlled by a set of rules. The combinations of such methods became known as *rule-based logic synthesis* and were the first type used in commercial applications.

One disadvantage of this approach was that when new gates were added to the database, new rules were needed to control which gates were to be used. As the rule-base grew, new rules interacted and interfered with the old rules. Another difficulty was that the rule base was specific to the set of gates being used. Still another was that the system became slow as the rule-base grew.

In the late 80's, rule-based rewriting gave way to technology independent optimization [6][7] followed by technology dependent mapping [14]. The notion was that there were a set of operations relatively good for a netlist, regardless of the final set of gates that would be used for the implementation. This decomposed the problem into separate steps, both of which were amenable to deeper mathematical analysis, leading to an algorithmic treatment of sub-problems and improved understanding.

This separation into technology independent and dependent parts required an abstract measure of goodness to guide the technology independent transformations. The *number of literals in the factored forms* became a standard metric for area and many algorithms were developed to improve this metric. For example, the operation "eliminate" collapsed the node into its fanouts if the worth of a node computed using this metric did not exceed a specified threshold. "Kerneling" selected divisors similarly. Node optimization minimized the logic function of a node represented in SOP form, which then was factored to obtain a literal count. Even though a minimum SOP and a factor were not unique, the heuristic used was that a factorization of a minimum SOP was close to the best factorization for the node function. If the result improved the metric, it was accepted and the current node function was replaced. Similar metrics were developed for delay.

At the same time, technology mapping methods were created based on algorithms which steadily improved over the years. In both cases, the algorithms became more sophisticated and generally more complex, driven by the need to process larger networks. Scalability of the algorithms became a dominant issue. Over the years, most of the classical logic synthesis algorithms developed since the late 80's were extended, made more scalable, or in some cases replaced by more scalable ones. These developments increased the barrier to understanding and implementation, required more complex code structures, making code maintenance more difficult. Much of these developments were done within commercial CAD companies, becoming valuable assets that were rarely made public.

In this paper, essentially we start over, keeping the separation between the technology independent and technology dependent aspects, but emphasizing simplicity, speed, and scalability. Local AIG-based transformations, such as *rewriting* [2][20], *resubstitution*, and *redundancy removal*, come back in a simplified form, making them fast and scalable for the following reasons:

- The network is an And-Inverter Graph (AIG) and all the transformations work on this uniform representation. The metric used for area is the total number of AND nodes, and for delay it is the maximum number of AND nodes on a path from the inputs to the outputs. These metrics are easy to compute and have a direct impact on technology mapping based on the AIG as the subject graph.
- Instead of graph isomorphic and SOP-based optimization, Boolean functions of a group of AIG nodes is computed in terms of various fanin cuts. Several exhaustive and selective cut computations can be used, including a recent development that can efficiently enumerate all cuts up to 12 inputs [9]. A quality/runtime tradeoff is controlled by selecting cuts of appropriate size.
- The Boolean functions of cuts are represented using truth tables, manipulated directly or used as hash keys into a

database where new rewriting structures are found. Due to the high speed of bit-parallel manipulation, sub-problems arising in local optimization are often solved, without using SAT or BDDs, by exhaustive simulation, which scales up to 16 inputs.

Since this kind of local optimization is fast, it can be repeated many times, making the transformations more global and hence more effective. For example, performing 10 rewriting passes over a typical network is still at least an order of magnitude faster than running the resource-aware implementation of the traditional synthesis found in MVSIS, and several orders of magnitude faster than *script.alegraic* in SIS. By applying greedy local optimizations many times, the scope of changes is no longer local. As a result, the cumulative effect of several optimization passes is often superior in quality to traditional synthesis .

The paper is organized as follows. Section 2 surveys Boolean networks, AIGs, windowing, and reconvergence-driven cut computation. Section 3 presents the algorithms for local rewriting, resubstitution, and redundancy removal adapted for AIGs and made efficient by rigorous enforcement of resource limits. Section 4 reports experimental results of the new scalable logic synthesis flow. Section 5 concludes and lists directions for future work.

2 Background

2.1 Boolean networks

Definition. A *Boolean function* is a mapping from n-dimensional ($n \geq 0$) Boolean space into a 1-dimensional one: $\{0,1\}^n \rightarrow \{0,1\}$.

Definition. A *Boolean network* is a directed acyclic graph (DAG) with nodes represented by Boolean functions. The sources of the graph are the *primary inputs* (PIs) of the network; the sinks are the *primary outputs* (POs).

Definition. The output of a node may be an input to other nodes called its *fanouts*. The inputs of a node are called its *fanins*. If there is a path from node A to B, then A is in the *transitive fanin* of B and B in the *transitive fanout* of A. The transitive fanin of B, $TFI(B)$, includes node B and the nodes in its transitive fanin, including the PIs. The *transitive fanout* of B, $TFO(B)$, includes node B and the nodes in its transitive fanout, including the POs. An *edge* in a Boolean network is a connection between two nodes, which are in the fanin/fanout relationship. The fanin/fanout of an edge is the fanin/fanout node of the connected pair of nodes.

Definition. A *maximum fanout free cone* (MFFC) of node n is a subset of the fanin cone, such that every path from a node in the subset to the POs passes through n. Informally, the MFFC of a node contains the node and all the logic used exclusively by the node. When a node is removed or substituted, the logic in its MFFC can be removed also.

Definition. A *cut* C of node n is a set of nodes of the network, called *leaves*, such that each path from a PI to n passes through at least one leaf. Node n is called *root* of cut C. The cut *size* is the number of its leaves. A *trivial cut* of the node is the cut composed of the node itself. A cut is *K-feasible* if the number of nodes in the cut does not exceed K. A cut is said to be *dominated* if there is another cut of the same node, which is contained, set-theoretically, in the given cut. The *volume* of a cut is the total number of nodes encountered on all paths between node n and the cut leaves.

Definition. The *local* function of an AIG node n, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in n and expressed in terms of the leaves, x, which form a cut of n. The *global function* of an AIG node is its function in terms of the PIs of the network.

Definition. *Exhaustive simulation* is a practical way of checking equivalence of Boolean function of a node whose size does not exceed 16 inputs. Exhaustive simulation is performed using bitwise simulation of the cone with 2^k different input patterns, where k is the number of leaves. Another way of looking at exhaustive simulation is that it computes the truth-table of the root in terms of the elementary truth-tables set at the leaves.

2.2 And-Interver Graphs

And-Interver Graph (AIG) is a Boolean network composed of two-input ANDs and inverters. AIGs were used in a variety of applications since the early 60's [12] as a convenient representation for combinational logic of an arbitrary Boolean network. To derive an AIG, the SOPs of the nodes in the network are factored [6], the AND and OR gates of the factored forms are converted into two-input ANDs and inverters using DeMorgan's rule, and these nodes are added to the AIG manager.

Structural hashing of netlists composed of arbitrary gates was proposed and used in early CAD tools [11] to detect and merge isomorphic circuit structures. For AIGs, structural hashing is performed when AND nodes are added to the AIG manager. It ensures that, for each pair of nodes, there is only one AND node having them as fanins (up to permutation).

Additionally, the AIG derived from a logic network is often *balanced,* by applying the associative transform, $a(bc) = (ab)c$, to reduce the number of AIG levels. Both structural hashing and balancing are done in one topological sweep from the PIs and have linear complexity in the number of AIG nodes.

The *size* (*area*) of the AIG is the number of its nodes. The *depth* (*delay*) of the AIG is the number of nodes on the longest path from the PIs to the POs. The goal of optimization by local transformations is to reduce both area and delay of an AIG, although in this paper we focus on the area optimization.

Software implementation of an AIG package is similar to that of an efficient BDD package [4]. Inverters are represented as flipped pointers to the AIG nodes. The AIG nodes have reference counters, which show the number of fanouts of each node. Reference counting leads to fast counting of nodes in an MFFC and efficient add/remove operations for individual nodes and their MFFCs.

Most BDD and AIG packages support the unique table to ensure that there is only one node with the given fanins. Due to the uniqueness of the Shannon expansion with respect to a variable, this leads to the canonical BDD structure for a given variable order. For AIGs, the AND-decomposition is not unique, so the use of a unique table guarantees only structural canonicity within one logic level (described above as structural hashing).

Maintaining an analogy with Reduced Ordered BDDs led to the development of the FRAIG package [18], which uses a balanced combination of simulation and Boolean satisfiability [16] to enforce functional canonicity of AIG nodes on-the-fly, as they are added to the package. The FRAIG package has found extensive use in logic synthesis, technology mapping [8], and equivalence checking [22] implemented in ABC [1].

It should be noted that, although AIGs are used as the main representation in this paper and in ABC, other types of simple circuit structures would also work: NAND graphs, OR-INV graphs, AND-XOR-INV graphs, Reduced Boolean Circuits [3], etc. All these representations are often close in size for practical circuits and have the same expressive power as AIGs, provided that AIGs allow for efficient detection of embedded XOR-subgraphs, which require specialized handling in some procedures.

Therefore, our choice of AIGs is somewhat arbitrary and is motivated by its straight-forward interpretation and convenient implementation.

The material on windowing and reconvergence-driven cut computation in the following two subsections assumes a general Boolean network, but the synthesis algorithms in Section 3 are optimized assuming an AIG.

2.3 Windowing

Windowing is a method of limiting the scope of logic synthesis to work only on a small portion of a Boolean network. This method is indispensable for scalability when working with large Boolean networks arising in industrial applications.

The material in this section is borrowed from [17], where windowing is used to compute complete don't-cares. A full presentation of the original algorithm is included below because the windowing algorithms, presented in this paper, build on and extend it in a number of ways.

Definition. Two non-overlapping subsets of nodes, the *leaf set* and the *root set*, are in a leaf/root relation if every path from the sources of the DAG to any node in the root set passes through some node in the leaf set.

Definition. Given two subsets in the leaf/root relationship, its *window* is the subset of nodes of the DAG that contains the root set plus all nodes on paths between the leaf set and the root set. The nodes in the leaf set are not included in the window.

Definition. A path connecting a pair of nodes is *distance-k* if it spans exactly *k* edges between the pair. Two nodes are *distance-k* from each other if the shortest path between them is distance-*k*.

The pseudo-code in Figure 2.3.1 and the example in Figure 2.3.2 describe the flow of a window construction algorithm. Procedure *Window* takes a node and two integers defining the number of logic levels on the fanin/fanout sides of the node to be included in the window. It returns the leaf set and the root set of the window.

```
(nodeset, nodeset) Window( node N, int nFanins, int nFanouts )
{
    nodeset  I1  = CollectNodesTFI( {N}, nFanins );
    nodeset  O1 = CollectNodesTFO( {N}, nFanouts );
    nodeset  I2  = CollectNodesTFI( O1, nFanins + nFanouts );
    nodeset  O2 = CollectNodesTFO( I1, nFanins + nFanouts);
    nodeset S = I2 ∩ O2;
    nodeset L = CollectLeaves( S );
    nodeset R = CollectRoots( S );
    return (L, R);
}
```

Figure 2.3.1. Computation of a window for a node.

The procedure *CollectNodesTFI* takes a set *S* of nodes and an integer $m \geq 0$, and returns a set of nodes on the fanin side that are distance-*m* or less from the nodes in *S*. An efficient implementation of this procedure for small *m* (for most applications, $m \leq 10$) iterates through the nodes that are distance-*k* ($0 \leq k \leq m$) from the given set. The distance-0 nodes are the original nodes. The distance-(*k*+1) nodes are found by collecting the fanins of the distance-*k* nodes not visited before. The procedure *CollectNodesTFO* is similar.

Procedures *CollectLeaves* and *CollectRoots* take the set of the window's internal nodes and determine the leaves and roots of this window. The leaves are the nodes that do not belong to the given set but are fanins of at least one of the nodes in the set. The roots are the nodes that belong to the given set and are also fanins of at least one node not in the set. Note that some of the roots thus computed are not in the TFO cone of the original node, for which the window is being computed, and therefore can be dropped without violating the definition of the window and undermining the usefulness of the window for logic synthesis computations.

We refer to the window constructed for a node by including *n* TFI logic levels and *m* TFO logic levels as an $n \times m$ window.

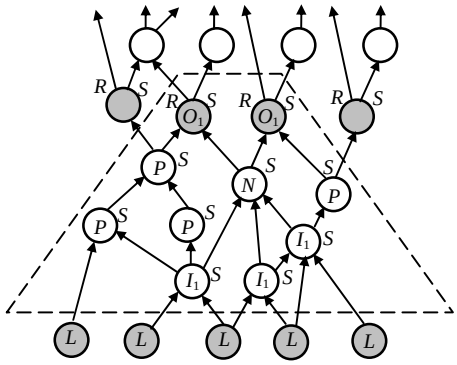

Figure 2.3.2 Example of the 1×1 window of node *N*.

Example. Figure 2.3.2 shows a 1×1 window for node *N* in a DAG. The nodes labeled I_1, O_1, *S, L,* and *R* are in correspondence with the pseudo-code in Figure 2.3.1. The window's roots (top) and leaves (bottom) are shaded. Note that the nodes labeled by *P* do not belong to TFI and TFO of *N*, but represent reconvergent paths in the vicinity of *N*. The left-most root and right-most root are not in the TFO of *N* and can be dropped, as explained above.

2.4 Reconvergence-driven cut computation

Definition. *Reconvergence* occurs when the paths starting at the output of a node meet again before reaching the POs. Reconvergence is inevitable due to logic sharing in multi-level logic networks, but excessive reconvergence is often redundant.

Some applications, such as resubstitution or computation of a subset of satisfiability don't-cares of a node, require only one *K*-feasible cut of the node, of size between 5 and 20 inputs, depending on the computational effort allowed. Such a cut can be computed using procedure *CollectNodesTFI* presented in the previous section on windowing. For example, this procedure can be called for a node, resulting in a cut composed of the leaves of the TFI cone extending several logic levels down from the node.

However, a problem with this is that it is hard to predict how many logic levels to traverse to get a cut of the desired size. A second problem is that the cut computed this way may not lead to a good optimization because it includes few nodes or a non-reconvergent (tree-like) logic structure. A small volume may lead to only a few nodes being available as resubstitution candidates. A tree-like structure does not lead to any don't-cares in the local space of the node. In both cases, the time spent computing the cut and attempting optimization using this would be wasted.

In this section, we present a simple and efficient cut computation algorithm, which computes a cut close to a given size (if such cut exists) while heuristically maximizing the cut volume and the number of reconvergent paths subsumed in the cut. This algorithm has been used recently in several applications, including technology-dependent resynthesis [19] and refactoring [20]. It will be used for resubstitution in Section 3.2.

Procedure *ReconvergenceDrivenCut* in Figure 2.4.1 computes one good-quality cut of a given size for node *N*. The procedure

uses two sets of nodes to store the leaves and the visited nodes, and initializes both with {*N*}. Next, the recursive procedure *ConstructCut_rec* is called. On termination, the computed cut is in the leaf set.

Procedure *ConstructCut_rec* tries to expand the cut by incrementally adding one node. If the cut is composed of only PIs, the procedure quits. Else, it selects a non-PI node *M* that minimizes the cost, equal to the number of nodes that will be added to the leaves if the given node is used to expand the cut. This cost is computed using procedure *LeafCost*. Note that the cost can be -1 if the node and both of its fanins are currently in the cut. Otherwise, the cost is 0 or more. If the least expansion of the cut makes it exceed the cut-size limit, the procedure quits. Otherwise, it updates the leaves and the visited nodes and calls itself recursively.

```
nodeset ReconvergenceDrivenCut( node N, int CutSizeLimit )
{
    nodeset Leaves = { N };
    nodeset Visited = { N };
    ConstructCut_rec( Leaves, Visited, CutSizeLimit );
    return Leaves;
}
ConstructCut_rec( nodeset Leaves, nodeset Visited, int CutSizeLimit )
{
    if ( Leaves contain only PI nodes )
        return;
    M = non-PI node in Leaves with the minimum LeafCost;
    if ( |Leaves| + LeafCost( M, Visited ) > CutSizeLimit )
        return;
    Leaves = Leaves ∪ fanins(M) \ M;
    Visited = Visited ∪ fanins(M);
    ConstructCut_rec( Leaves, Visited, CutSizeLimit );
}
int LeafCost( node M, nodeset Visited )
{
    int Cost = -1;
    for each fanin F of node M
        if F does not below to Visited
            Cost = Cost + 1;
    return Cost;
}
```

Figure 2.4.1. Reconvergence-driven cut computation.

The above procedure works well because, at each step, it greedily minimizes the number of the cut leaves. In doing so, it tends to subsume into the cut those nodes that contribute to the volume but not to the cut size. It also prefers nodes with the costs -1 and 0, which are the cut leaves with one or more reconvergent paths inside the cut.

3 Logic synthesis algorithms for AIGs

In this section, we describe several algorithms for efficient AIG-based logic synthesis. The first algorithm (*rewriting*) was developed specifically for AIGs. Other algorithms (*resubstitution* and *redundancy removal*) were adapted to the AIG representation from traditional synthesis. Using the AIG representation simplifies the algorithms, improves their speed and scalability, and makes them easier to implement, compared to the general ones.

3.1 Rewriting

Rewriting is a fast greedy algorithm for minimizing the number of AIG nodes by iteratively selecting AIG subgraphs rooted at a node and replacing them with smaller pre-computed subgraphs, while preserving the functionality of the node. The AIG rewriting algorithm [20] used in this paper was inspired by [2], which has been extended in the following ways:

- Using 4-input cuts instead of two-level subgraphs.
- Restricting rewriting to preserve the number of logic levels.
- Developing several variations of AIG rewriting that look at larger subgraphs and/or attempt to reduce the delay.
- Experimental tune-up for logic synthesis applications.

For the purposes of AIG rewriting, all 4-feasible cuts of the nodes are found by the fast cut enumeration procedure [24][21]. For each cut, the Boolean function is computed and its NPN-class is determined by table lookup. Manipulation of 4-variable functions is fast because they are represented using truth tables stored as 16-bit bit-strings. Altogether there are 222 NPN equivalence classes of 4-variable functions [23], of which only about 100 appear as functions of 4-feasible cuts in any of the available benchmarks, and only about 40 of these have been found experimentally to lead to improvements in rewriting. The unifying characteristic of these 40 remaining NPN-classes is that they are decomposable using simple disjoint-support decomposition [2].

All non-redundant AIG implementations of the representative functions of the 40 equivalence classes are pre-computed in advance and stored in the hash table, hashed by the truth table of their NPN-class. The AIG subgraphs are stored in a shared DAG with approximately 2,000 nodes. This DAG is compiled into our program as an integer array, which noticeably reduced the setup time of the rewriting package.

```
Rewriting( network AIG, hash table PrecomputedStructures, bool UseZeroCost )
{
    for each node N in the AIG in the topological order {
        for each 4-input cut C of node N computed using cut enumeration {
            F = Boolean function of N in terms of the leaves of C
            PossibleStructures = HashTableLookup( PrecomputedStructures, F );
            // find the best logic structure for rewriting
            BestS = NULL; BestGain = -1;
            for each structure S in PossibleStructures {
                NodesSaved = DereferenceNode( AIG, N );
                NodesAdded = ReferenceNode( AIG, S );
                Gain = NodesSaved – NodesAdded;
                Dereference( AIG, S );
                Reference( AIG, N );
                if ( Gain > 0 || (Gain = 0 && UseZeroCost) )
                    if ( BestS = NULL || BestGain < Gain )
                        BestS = S; BestGain = Gain;
            }
            // use the best logic structure to update the netlist
            if ( BestS != NULL ) {
                NodesSaved = DereferenceNode( AIG, N );
                NodesAdded = ReferenceNode( AIG, S );
                assert( BestGain = NodesSaved – NodesAdded );
            }
        }
    }
}
```

Figure 3.1.1. 4-input rewriting algorithm.

Figure 3.1.1 shows the pseudo-code of the AIG rewriting procedure. The nodes are visited in the topological order. For each 4-input cut of a node, all pre-computed subgraphs are considered. Logic sharing between the new subgraphs and nodes already in the network is detected using an AIG with reference counters implemented similarly to those in a BDD package. In this case, the old subgraph is de-referenced and the number of nodes, whose reference counts became 0, is returned. This is the number of nodes saved by not having the old subgraph in the network. Then, a new

subgraph is added to the network while counting the number of new nodes and the nodes whose reference count changes from 0 to a positive value. This is the increase in the number of nodes due to having the new subgraph in the network. The difference between the former and the latter numbers is the gain in the number of nodes if the replacement is done. The new node is de-referenced and the old node referenced to return the AIG to its original state.

After trying all available subgraphs, the one that leads to the largest improvement at a node is used at the node. If there is no improvement but "zero-cost replacement" is enabled, a new subgraph that does not increase the number of nodes is used.

Example. Figure 3.1.2 shows three AIG subgraphs for the function $F = abc$. They are pre-computed and stored. Figure 3.1.3 shows two instances of AIG rewriting. The upper part of the figure shows the situation when Subgraph 1 is detected in the circuit and replaced by Subgraph 2. The lower part of the figure shows two nodes AND(*a, b*) and AND(*a, c*) already present in the network. In this case, Subgraph 2 can be replaced by Subgraph 1. In both cases, the network has one node less.

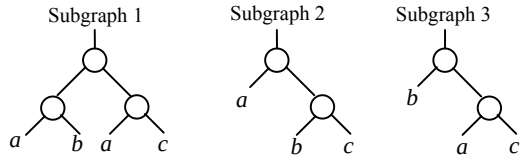

Figure 3.1.2. Different AIG structures for function $F = abc$.

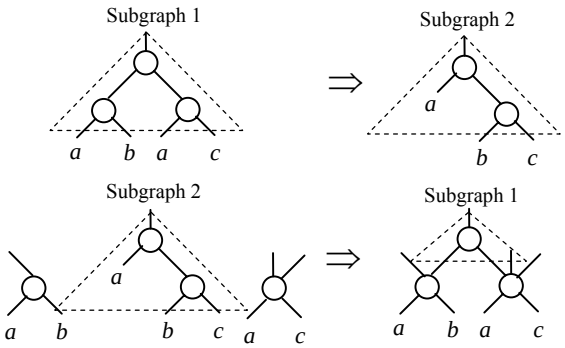

Figure 3.1.3. Two cases of AIG rewriting of a node.

For further details on AIG rewriting and experimental comparison with the traditional logic synthesis, refer to [20].

3.2 Resubstitution

Resubstitution expresses the function of a node using other nodes (called *divisors*) already present in the network. The transformation is accepted if the new implementation of the node is in some sense better than its current implementation using the immediate fanins.

In the case of an AIG, the best outcome of resubstitution is when the whole MFFC of the node can be freed and the node's function can be expressed using other nodes, currently outside of the node's MFFC. This outcome corresponds to a 0-resubstitution when no new nodes are added to the AIG while the MFFC is removed. Similarly, if the MFFC is composed of more than one node, and there is no 0-resubstitution, the best outcome is given by a 1-resubstitution, which exists if the function of the node, whose MFFC is being considered, can be expressed using the available node and exactly one additional node.

This approach generalizes to the case when a node's MFFC has k+1 nodes and a *k-resubstitution* of the node is attempted as a way to reduce the total number of nodes in the AIG. It is conceptually similar to technology-dependent resynthesis based on resubstitution [15][18]. The difference is, in the case of AIGs, a technology-independent metric (the AIG size) is used, and for scalability and speed, we restrict to cuts with no more than 12-16 leaves. For such relatively small cuts, resubstitution can be performed without BDDs and SAT, using explicitly computed truth tables and exhaustive simulation.

```
Resubstitution( network AIG, int CutSizeLimit, int DivisorLimit, bool UseZeroCost )
{
    for each node n in the AIG in the topological order {
        int MffcSize = |MFFC( n )|;  assert( MffcSize ≥ 1 );
        nodeset C = ReconvergenceDrivenCut( n, CutSizeLimit );
        nodeset D = CollectNodesTFOChanged( C, Level(n), DivisorLimit );
        ComputeFunctions( D, n );
        nodeset R = Try0Resubstitution( n, D );
        if ( R == NULL && (MffcSize > 1 || (MffcSize == 1 && UseZeroCost)) )
            R = Try1Resubstitution( n, D );
        if ( R == NULL && (MffcSize > 2 || (MffcSize == 2 && UseZeroCost)) )
            R = Try2Resubstitution( n, D );
        if ( R == NULL && (MffcSize > 3 || (MffcSize == 3 && UseZeroCost)) )
            R = Try3Resubstitution( n, D );
        If ( R != NULL )
            UpdateNetwork( AIG, n, R );
    }
}
```

Figure 3.2.1. AIG resubstitution algorithm.

Figure 3.2.1 shows the pseudo-code of the resubstitution algorithm. For each node processed, the MFFC (which includes at least the node itself) and a reconvergence-driven cut is computed (Section 2.4). Next, a limited TFO computation is performed using the cut, the level of node *n,* and a limit on the number of divisors. It differs from *CollectNodesTFO* in Section 2.3: (a) instead of collecting the nodes that are no more than *k* levels from the nodes in the starting set, it collects the nodes whose level does not exceed the level of node *n*, (b) it collects the leaves but does not collect nodes that are in the MFFC, and (c) it collects no more than a given number of nodes (*DivisorLimit*). Next, the Boolean functions of the collected nodes and the node *n* are computed.

Several resubstitutions of the node are attempted, ordered by the number of new nodes added to the AIG. First, procedure *Try0Resubstitution* checks whether the Boolean function of node *n* is a constant or equal (up to complementation) to that of a divisor. If there is no 0-resubstitution, *Try1Resubstitution* is attempted by trying to find two divisors that, when ANDed in some polarities, are equal (up to complementation) to the function of node *n*. Note that 1-resubsitution can only lead to improvement in the AIG size if the size of MFFC is more than one. It can be used to restructure the AIG without changing its size if the size of the MFFC is exactly two. In other cases, 1-resubsitution will increase the AIG size and, therefore, it is not considered. Similar observations hold for the higher-order resubstitutions.

Our resource-aware implementation limits the considered set of divisors, *D*, because the complexity of *k*-resubstitution is $O(|D|^{k+1})$. Our current implementation uses $k = \{0, 1, 2, 3\}$ while the number of divisors computed in *CollectNodesTFOChanged* in Figure 3.2.1 is limited to 150. It is interesting that, with so many divisors and the high polynomial complexity of the resubstitution test, the runtime of the current implementation is often dominated by collecting the nodes in the TFO of the cut. For large cuts, the exhaustive simulation of the nodes takes about 1/4 of the runtime.

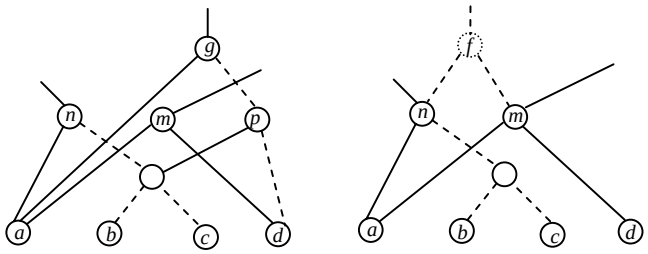

Figure 3.2.2. Example of AIG resubstitution.

Example. In the AIG shown in Figure 3.2.2 (left), node *g* has MFFC of size 2 composed of nodes *g* and *p*. One node can be saved if node *g* is replaced by the complement of node *f,* which does not belong to the original AIG but can be added on top of the already present nodes *n* and *m,* as shown in Figure 3.2.2 (right). Indeed, $g = a(b + c + d)$, $\overline{f} = n + m = a(b + c) + ad = a(b + c + d)$. This resubstitution is an example of applying the algebraic distributive transform

3.3 Redundancy removal

Redundancy removal (RR) is a transformation identifying and removing connections between nodes, which do not impact the functionality of the network POs.

RR is closely related to a don't-care-based optimization of general Boolean networks, which simplifies the local functions of the nodes using subsets of internal don't-cares caused by the surrounding network. More than a decade of research resulted in several scalable computations, which for each node, a subset of compatible observability don't-cares (CODCs) [23] or a subset of complete don't-cares computed using windowing [15].

These don't-care computations can be adapted to work on AIGs. The presence of a don't-care at the inputs of a two-input AND node can cause a node to be replaced by a constant, or an inverter or buffer, or cause no change. Any improvement can be seen as due to a redundancy in the network. In one case, the edges from a node to all its fanouts are redundant; in the other cases, one of the fanin edges of the node is redundant. Therefore, don't-care-based minimization of logic networks becomes RR for AIGs. Note that SOP minimization with a two-level minimizer (for example, ESPRESSO) is not needed, which contributes to the scalability of RR for AIGs. This motivated the following efficient procedure based on structural analysis, simulation, and Boolean satisfiability.

The computation begins by global simulation of the network with the goal of detecting as many non-redundant edges as possible. The remaining edges are potentially redundant and will be checked by the procedure shown in Figure 3.3.1.

The procedure *RedundancyRemoval* iterates over candidate redundant edges. For each edge, it creates a window for RR. The window computation is different from the general windowing presented in Section 2. The first difference is that when *TFO2* is computed, the given edge is not traversed. As a result, *TFO2* contains only the nodes reachable by the reconvergent paths around the node. The second difference is the intersection of the transitive fanout cone of the node (*TFO*) and the transitive fanout cone of the leaves (*TFO2*). If the given node (the fanout of the edge) is among the roots of the intersection, then the propagation of the node's value to the POs cannot be blocked by reconvergent paths. In this case, structural analysis is enough to show that the edge cannot be redundant. Otherwise, the intersection of *TFI2* and *TFO2* gives the desired window.

```
RedundancyRemoval( network AIG, int WindowSize, int Timeout )
{
    edgeset E = candidate AIG edges not disproved by random simulation;
    for each candidate edge in E {
        W = CreateWindowForRR( E, WindowSize, WindowSize );
        if ( containing windows W of this size does not exist )
            continue;
        network Miter = ConstructMiter( W, E );
        if ( RandomSimulation( Miter ) )  // edge is not disproved by simulation
            if ( CheckRRusingSat( Miter, Timeout ) )  // proved redundant by SAT
                UpdateNetwork( AIG, E );
    }
}

(nodeset, nodeset) CreateWindowForRR( edge E, int nFanins, int nFanouts )
{
    nodeset TFI = CollectNodesTFI( {Fanin(Edge)}, nFanins );
    nodeset TFO2 = CollectNodesTFO( CollectLeaves(TFI), nFanins + nFanouts );
    nodeset TFO = CollectNodesTFO( {Fanout(Edge)}, nFanouts );
    nodeset Roots = CollectRoots( TFO ∩ TFO2 );
    if ( Fanout(Edge) ∈ Roots )
        return window of this size does not exist;
    nodeset TFI2 = CollectNodesTFI( Roots, nFanins + nFanouts );
    nodeset S = TFI2 ∩ TFO2;
    nodeset L = CollectLeaves( S );
    nodeset R = CollectRoots( S );
    return (L, R);
}
```

Figure 3.3.1. Redundancy removal for AIGs.

If the window exists, the miter [4] is constructed, as shown in Figure 3.3.2. The output of the miter is constant 0 iff the edge is non-redundant. The miter construction is similar to the computation of complete don't-cares [15] when two copies of the window are compared: the unmodified window and the window, in which the given edge is assumed redundant.

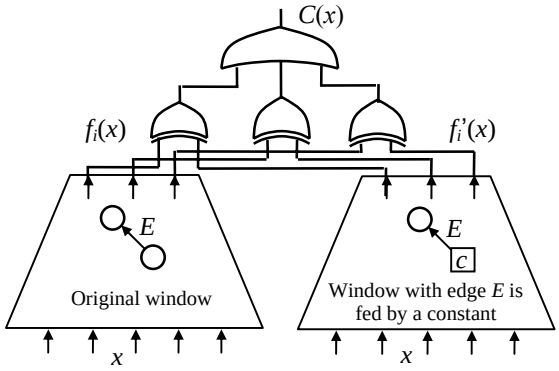

Figure 3.3.2. A miter used for RR.

Random simulation is applied to the miter for early detection of non-redundancy. Although some random simulation was performed at the beginning to filter out obvious non-redundant edges, the additional simulation may be helpful because, after removing some of the redundancies, previous candidate redundant edges may have become non-redundant. If the new random simulation test passes, the edge can still be redundant. In this case, the miter is solved by SAT [22]. If the miter is proved unsatisfiable, the edge is indeed redundant because, for all assignments to the leaves of the window, the output of the original

window and the modified window are the same. In this case, the redundant edge is removed from the AIG.

Another interesting aspect of the RR in AIGs is that, for nodes with multiple fanouts, we can define a fanin edge to be redundant with respect to one fanout. This can be illustrated as follows. Consider as many copies of the node as there are fanouts. A fanin edge may be redundant in one of the copies. As a result of removing a redundancy in this copy, the fanout no longer depends on the node but instead on one of the node's fanins. In the original network, this allows us to replace the associated fanout of the node by one of the nodes fanins. No nodes are saved, but the network is restructured and false logic sharing has been removed. Possibly, the level (delay) of the AIG is reduced. Such redundant logic sharing may be the result of greedy AIG rewriting. Removing it using RR may be desirable to improve stuck-at testability of the network, even if it does not lead to savings in terms of AIG nodes.

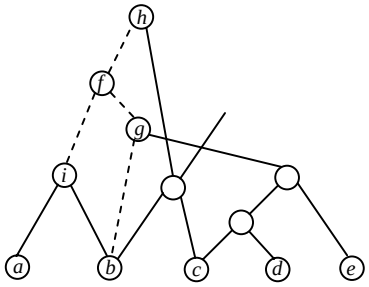

Figure 3.3.3. Example of redundancy removal in AIGs.

Example. In the AIG shown in Figure 3.3.3, the AIG edge $g \to f$ is redundant because the function of h, the only fanout of f, does not change when the edge $g \to f$ is dropped (when g is replaced by constant 0). Indeed, $\overline{f} = ab + \overline{b}cde$, $h = \overline{f}\, bc = (ab + \overline{b}cde\,)bc = abc$. After the change, $h = \overline{f}\, bc = (ab+0)bc = abc$. Edge $g \to f$ can be proved redundant by considering a window with roots $\{h\}$ and leaves $\{a, b, c, d, e\}$.

4 Experimental Results

The presented algorithms are implemented in a public-domain logic synthesis and verification system called ABC [1].

An experimental comparison of AIG rewriting and traditional logic synthesis can be found in [20]. This section compares AIG rewriting with and without resubstitution for area-only AIG optimization for several large IWLS 2005 benchmarks [13]. The benchmarks for this experiment are selected based on their size, and not based on the improvements by the proposed algorithms.

Tables 1 and 2 compare the reduction in AIG size and runtime. The runtimes are measured on a 1.6GHz IBM ThinkPad with 1Gb RAM. All synthesis results have been verified using the combinational equivalence checking command *cec* in ABC [22].

Table 1 lists the benchmark names and the number of AIG nodes after structural hashing (column *strash*), and after using a number of scripts described in Figure 4.1; three iterations of AIG rewriting (column 3×*rw*), three iterations of resubstitution (column 3×*rs*), three iterations of rewriting interleaved with resubstitution (column 3×*rws*), an area-minimization script *compress*2 (column *com*2), and a script derived from *compress*2 by interleaving rewriting and resubstitution (column *com*2*rs*). Table 2 compares the runtimes of the commands listed in Table 1.

Table 1 shows that for area minimization resubstitution alone cannot compete with rewriting (columns 3×rw and 3×rs). However, interleaving resubstitution with rewriting leads to a 2-3% average improvement in the number of AIG nodes, compared to iterating rewriting (columns 3×rw and 3×rws) and using rewriting as part of an area-minization script (columns *com*2 and *com*2*rs*). Resubstitution improves rewriting because it explores a larger search space than rewriting, which looks only at 4-input cuts.

It should be noted that although a 3% average reduction in area may not seem to be a substantial improvement, in practice reducing the AIG size further, after the 10 passes of AIG rewriting, is hard to achieve. Even when a relatively expansive don't-care-based optimization is applied [17] and the resulting netlist is converted back into an AIG, the reduction is rarely more than 3%.

Table 1. The number of AIG nodes with and w/o resubstitution.

design	**strash**	**3×rw**	**3×rs**	**3×rws**	**com2**	**com2rs**
aes_core	21213	19814	20432	19413	19735	19326
des_perf	78299	69870	70577	66931	69196	65671
ethernet	19729	14415	18175	14153	13020	12784
pci_bridge32	22784	18062	22202	17899	17817	17701
usb_funct	15873	13779	14579	13299	13059	12540
vga_lcd	126711	91152	118926	90977	90844	90713
wb_conmax	47853	43847	45248	41666	40407	39645
average ratio	**1.21**	**1.00**	**1.13**	**0.97**	**0.96**	**0.94**

Table 2. The runtime (in seconds) with and w/o resubstitution.

design	**strash**	**3×rw**	**3×rs**	**3×rws**	**com2**	**com2rs**
aes_core	0.34	1.82	13.64	15.23	4.96	25.70
des_perf	1.00	16.31	34.19	47.16	31.89	73.42
ethernet	0.18	1.13	4.25	4.23	1.97	5.87
pci_bridge32	0.22	1.36	4.89	5.26	2.59	7.90
usb_funct	0.14	0.96	2.39	2.66	1.87	4.91
vga_lcd	1.71	20.50	120.75	92.98	31.00	131.53
wb_conmax	0.37	5.30	12.91	15.78	8.47	25.42
average ratio	**0.12**	**1.00**	**4.00**	**4.16**	**1.92**	**6.56**

3×rw: st; rw -l; rwz -l; rwz -l
3×rs: st; rs -K 6 -N 2 -l; rs -K 9 -N 2 -l; rs -K 12 -N 2 -l
3×rws: st; rw -l; rs -K 6 -N 2 -l; rwz -l; rs -K 9 -N 2 -l; rwz -l; rs -K 12 -N 2 -l
compress2: b -l; rw -l; rf -l; b -l; rw -l; rwz -l; b -l; rfz -l; rwz -l; b –l
compress2rs: b -l; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l; rf -l; rs -K 8 -l; b -l; rs -K 8 -N 2 -l; rw -l; rs -K 10 -l; rwz -l; rs -K 10 -N 2 -l; b -l; rs -K 12 -l; rfz -l; rs -K 12 -N 2 -l; rwz -l; b -l
Semicolons separate individual commands in the scripts:
st is structural hashing
b is algebraic tree-balancing
rw is rewriting
rwz is rewriting with zero-cost replacement
rf is refactoring
rfz is refactoring with zero-cost replacement
rs is resubstitution
-K <num> is the limit on the number of cut inputs (default is 6)
-N <num> is the limit on the number of new nodes that can be added while performing resubstitution at each node (default is 1)
-l turns on the area-minimization mode (minimizing the number of AIG nodes while disregarding the number of AIG levels).

Figure 4.1. Description of various rewriting scripts.

Table 2 shows that the current implementation of resubstitution is as scalable as AIG rewriting although several times slower. In most cases, the runtime of the current implementation is dominated by the computation of candidate Boolean divisors at each node by

backward reachability from the cut leaves (procedure *CollectNodesTFOChanged* in Figure 3.2.1). The truth table computation using exhaustive simulation is relatively fast even for cuts with more than 10 inputs. Similarly, divisor checking, which is quadratic and cubic in the number of divisors for resubstitution with 1 and 2 additional nodes, respectively, is less time-consuming because of the resource limits on the number of divisors.

Additional experimental results showing the impact of redundancy removal on the above flow will be available in the final version of the paper.

5 Conclusions and Future Work

This paper presents several local transformations for combinational logic synthesis. The old idea of peephole optimization acquires new life when used together with several recent techniques, such as precomputation of AIG subgraphs, reconvergence analysis, efficient bottom-up cut enumeration, Boolean matching, exhaustive simulation of logic cones of up to 16 inputs, and local resource-aware runs of Boolean satisfiability.

Another distinctive feature of this work is that it uses AIGs exclusively during all synthesis steps as a simple multi-level logic representation. This allows for the local transformations to be applied at high speed and in a scalable manner. The cumulative gain of several rounds of local transformations is comparable in quality with the logic synthesis scripts in MVSIS and SIS while being one or two orders of magnitude faster as well as applicable to larger examples.

The new technique has a potential for completely replacing traditional logic synthesis in the CAD tools. The extreme speed and good quality of the proposed algorithms make new scalable synthesis useful in many applications, from hardware emulation and early estimation of the design complexity to software synthesis [27] and preprocessing miters before equivalence checking [22].

Future work will include extending the local transformations to use even larger cut sizes. The challenge is to search a larger space of alternative circuit structures while keeping runtime low, which will allow multiple optimization passes.

Acknowledgement

This research was supported in part by NSF contract, CCR-0312676, by the MARCO Focus Center for Circuit System Solution under contract 2003-CT-888, and by the California Micro program with our industrial sponsors, Intel, Magma, and Synplicity.

References

[1] Berkeley Logic Synthesis and Verification Group, *ABC: A system for sequential synthesis and verification*, *Release 60306*. http://www.eecs.berkeley.edu/~alanmi/abc/

[2] V. Bertacco and M. Damiani, "Disjunctive decomposition of logic functions," *Proc. ICCAD '97*, pp. 78-82.

[3] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification", *Proc. ICCAD '04*, pp. 42-49.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package", *Proc. DAC '90*, pp. 40-45.

[5] D. Brand. "Verification of large synthesized designs". *Proc. ICCAD '93*, pp. 534 -537.

[6] R. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

[7] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, "Multilevel logic synthesis", *Proc. IEEE*, Vol. 78, Feb.1990.

[8] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD'05*. http://www.eecs.berkeley.edu/~alanmi/publications/2005/iccad05_map.pdf

[9] S. Chatterjee, A. Mishchenko, and R. Brayton, "Factor-cut computation for FPGA mapping with large cut sizes", *Submitted to IWLS '06*. ttp://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_cut.pdf

[10] J. Cong, C. Wu and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," *Proc. FPGA `99*, pp. 29-35.

[11] A. Darringer, W. H. Joyner, Jr., C. L. Berman, L. Trevillyan, "Logic synthesis through local transformations," *IBM J. of Research and Development*, Vol. 25(4), 1981, pp 272-280.

[12] L. Hellerman, "A catalog of three-variable Or-Inverter and And-Inverter logical circuits", *IEEE Trans. Electron. Comput.,* Vol. EC-12, June 1963, pp. 198-223.

[13] *IWLS '05 Benchmarks*. http://iwls.org/iwls2005/benchmarks.html

[14] K. Keutzer, "DAGON: Technology binding and local optimizations by DAG matching", *Proc. DAC '87*, pp. 617-623.

[15] V. N. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization", *Proc. DAC '04*, pp. 438-441.

[16] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking". *Proc. ICCAD'04*, pp 50-57.

[17] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization", *Proc. DATE '05*, pp. 418-423. http://www.eecs.berkeley.edu/~alanmi/publications/2005/date05_satdc.pdf

[18] A. Mishchenko, S. Chatterjee, R. Jiang, R. Brayton. "FRAIGs: A unifying representation for logic synthesis and verification". *Technical Report, UC Berkeley, 2004*. http://www.eecs.berkeley.edu/~alanmi/publications/2005/tech05_fraigs.pdf

[19] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks", *IEEE TCAD*, May 2006. http://www.eecs.berkeley.edu/~alanmi/publications/2005/tcad05_s&s.pdf

[20] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Submitted to DAC '06*. http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_rwr.pdf

[21] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs", *Proc. FPGA'06*, pp. 41-49. http://www.eecs.berkeley.edu/~alanmi/publications/2006/fpga06_map.pdf

[22] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Eén, "Improvements to combinational equivalence checking", *Submitted to IWLS '06*. ttp://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_cec.pdf

[23] S. Muroga, *Logic design and switching theory*, John Wiley & Sons, Inc., New York, NY, 1979

[24] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *Proc. FPGA '98*, pp. 35-42.

[25] H. Savoj. *Don't cares in multi-level network optimization*. Ph.D. Dissertation, UC Berkeley, May 1992.

[26] E. Sentovich et al. "SIS: A system for sequential circuit synthesis." *Technical Report, UCB/ERI, M92/41, ERL*, Dept. of EECS, UC Berkeley, 1992.

[27] A. Solar-Lezama, R. Rabbah, R. Bodik, K. Ebcioglu, "Programming by sketching for bitstreaming programs", *ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '05*), Chicago, IL, June 2005