DAG-aware AIG rewriting: a fresh look at combinational logic synthesis

Conference Paper · January 2006

DOI: 10.1145/1146909.1147048 · Source: DBLP

3 authors, including:

Satrajit Chatterjee

45 PUBLICATIONS **1,519** CITATIONS

SEE PROFILE

DAG-Aware AIG Rewriting
A Fresh Look at Combinational Logic Synthesis

Alan Mishchenko
Department of EECS
University of California, Berkeley
Berkeley, CA 94720
alanmi@eecs.berkeley.edu

Satrajit Chatterjee
Department of EECS
University of California, Berkeley
Berkeley, CA 94720
satrajit@eecs.berkeley.edu

Robert Brayton
Department of EECS
University of California, Berkeley
Berkeley, CA 94720
brayton@eecs.berkeley.edu

ABSTRACT
This paper presents a technique for preprocessing combinational logic before technology mapping. The technique is based on the representation of combinational logic using And-Inverter Graphs (AIGs), a networks of two-input ANDs and inverters. The optimization works by alternating DAG-aware AIG rewriting, which reduces area by sharing common logic without increasing delay, and algebraic AIG balancing, which minimizes delay without increasing area. The new technology-independent flow is implemented in a public-domain tool ABC. Experiments on large industrial benchmarks show that the proposed methodology scales to very large designs and is several orders of magnitude faster than SIS and MVSIS while offering comparable or better quality when measured by the quality of the network after mapping.

Categories and Subject Descriptors
B.6.3 [**Logic Design**]: Design Aids – Automatic synthesis.

General Terms
Algorithms, Performance, Experimentation, Theory.

Keywords
Technology-independent logic synthesis, And-Inverter Graphs, NPN equivalence, technology mapping.

1 INTRODUCTION
Optimization of multi-level logic networks using logic synthesis [4][5] plays an important role in automated design flow. Logic synthesis is often applied to the network derived by compiling HDLs, such as VHDL or Verilog, before performing technology mapping for standard cells or programmable devices. Other uses of logic synthesis include hardware emulation, design complexity estimation, software synthesis, and fast preprocessing of circuits before equivalence checking [3]. Traditional combinational logic synthesis, exemplified by SIS [18] and MVSIS [16], applies a sequence of optimization steps, having the goal of removing redundant nodes (*sweep*), finding better logic boundaries (*eliminate, resubstitute*), discovering shared logic (*fast_extract*), and simplifying the node representations (*simplify, full_simplify*).

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

Traditional synthesis has several drawbacks:
- It often relies on trial-and-error and hand-tuning of the optimization scripts.
- Improvements are measured using the reduction in the number of literals in the factored forms of the node SOPs, while technology mappers [7][10] often use cost functions not correlated with the literal counts.
- It is complicated and hard to implement. An implementation of a robust technology-independent synthesis flow in SIS and MVSIS takes several person-months, in addition to in-depth knowledge of logic synthesis.
- Even in its robust implementations, with resource limits controlling runtime and memory, traditional synthesis is often slow because it involves time-consuming steps, such as computation of internal don't-cares [11].

We propose a new technology-independent combinational logic synthesis flow using fast local transformations of And-Inverter Graphs (AIGs), composed of two-input ANDs and inverters. The flow improves on the traditional logic synthesis by addressing the above difficulties. Advantages are summarized as follows:
- While still being heuristic and suboptimal, the new algorithm does not require as much hand-tuning and trial-and-error.
- Improvements in the complexity of the logic are measured by AIG nodes and levels, in better correspondence with both standard-cell [6] and FPGA mappers [14], which use AIGs or similar data structures as subject graphs.
- It is much simpler. A robust implementation reported in this paper took a few person-weeks to conceive and implement.
- It is orders of magnitude faster than the traditional flow, even when compared with its most rugged and robust versions, while the quality is comparable or better when measured by the delay and area of the network after technology mapping.

AIG rewriting is local; however, rewriting is very fast and can be applied to the network many times. For example, performing ten rewriting passes over a typical network is still at least an order of magnitude faster than running the resource-aware implementation of the traditional flow in MVSIS. By applying rewriting many times, the scope of changes is no longer local. The result is that the cumulative effect of several rewriting passes is often superior to traditional synthesis in terms of quality.

2 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.

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
  - selectively collapse and *refactor* [4] larger subgraphs,
  - *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.

```
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;
            }
            if ( BestS == NULL ) continue;
            // use the best logic structure to update the netlist
            NodesSaved = DereferenceNode( AIG, N );
            NodesAdded = ReferenceNode( AIG, S );
            assert( BestGain = NodesSaved – NodesAdded );
        }
    }
}
```

Figure 1. 4-input rewriting algorithm.

Example. Figure 2 shows three AIGs for $F = abc$ that are pre-computed 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.

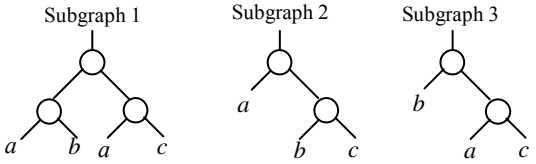

Figure 2. Different AIG structures for function $F = abc$.

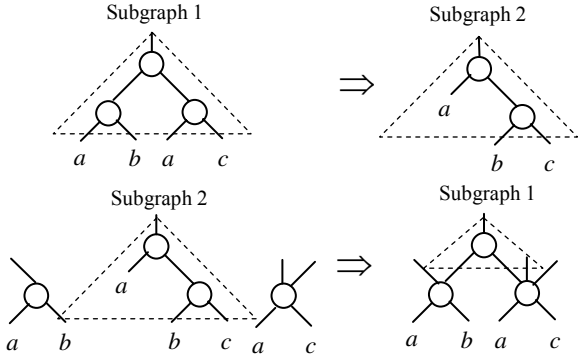

Figure 3. Two cases of AIG rewriting of a node.

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 EXPERIMENTAL RESULTS

AIG rewriting is implemented in the sequential logic synthesis and verification system, ABC [1], as commands *rewrite, refactor,* and *balance*. A rewriting script, *resyn2*, was defined as an alias in the resource file *abc.rc* [1]. This script performs 10 passes over the network as follows: *b; rw; rf; b; rw; rwz; b; rfz; rwz; b.* In the abbreviated notation, *b* (*balance*) stands for AIG balancing, *rw/rf* (*rewrite/refactor*) stands for AIG rewriting/refactoring, and *rwz/rfz* is the same but with zero-cost replacements allowed.

The *resyn2* script optimizes area under delay constraints. It starts by balancing to reduce delay upfront as much as possible. Next, rewriting/refactoring and balancing are interleaved. During this, rewriting/refactoring tries to reduce area while not increasing delay. Balancing tries to reduce delay while not increasing area. Zero-cost replacements are enabled later in the script to facilitate creating new rewriting opportunities. This process in *resyn2* is stopped after three iterations. Generally, this heuristic approach works well for a variety of benchmarks.

One difficulty in comparing the quality of AIG rewriting with traditional logic synthesis is their use of different cost functions. Previously, improvements were measured by counting the sum total of literals in the factored forms while AIG rewriting looks at the total number of AIG nodes and the maximum number of AIG levels. Therefore, in Tables 2 and 3, we compare the impact of AIG rewriting to that of logic synthesis in SIS and MVSIS, *after technology mapping*. We used the technology mappers in ABC, for FPGAs [14] and standard cells [6] using the library *mcnc.genlib* from the SIS distribution. A load-independent timing model was assumed. Our experiments with a load-independent combinational mapper in an industrial setting confirm that gate sizing and buffering can be done in later stages of the flow.

Experiments were performed on many public-domain benchmarks, including industrial circuits from IWLS 2005 [9]. Section 4.1 analyzes the performance of the rewriting script. Section 4.2 compares AIG rewriting with logic synthesis scripts in SIS and MVSIS. Section 4.3 gives detailed statistics for IWLS 2005 benchmarks, showing the impact of AIG rewriting on tech-mapping for FPGAs and standard cells.

In all cases, the netlists produced by SIS, MVSIS and ABC were structurally hashed and algebraically balanced for minimum delay in ABC before mapping. The resulting netlists were verified using a SAT-based equivalence checker in ABC [13].

Due to page limitation only the largest 10 IWLS benchmarks are shown in Tables 2 and 3, although the average ratios listed in the last row of the tables refer to a set of 21 benchmarks used.

4.1 Performance and runtime analysis

The performance of rewriting is analyzed in Table 2. The first column lists the benchmarks. The next five columns show the number of primary inputs (PI), primary outputs (PO), latches (Latch), AIG nodes (AND2), and logic levels of two-input AND gates (Lev). The number of gates and logic levels is given for an AIG after structural hashing and algebraic balancing.

The next eight columns show the AIG rewriting statistics after two successive applications of *rwz* to the original benchmarks. The columns show the number of 4-input cuts computed for all internal nodes ("Cuts"), the number of subgraphs tried during rewriting ("Subgrs"), the number of times a rewriting was accepted ("Upds"), and the improvement in the number of AIG nodes after each rewriting pass.

The data shows that the second pass of rewriting leads to smaller but still non-negligible gains in the number of AIG nodes (18% of the first pass). This confirms that the zero-cost replacements are useful for restructuring logic, allowing new rewriting possibilities. Without zero-cost replacements, the second pass improves by only 11% (data is not shown in the table). With replacements the first pass reduces the number of nodes by 14%, while without zero-cost replacements, by 12%.

The last three columns of the table show the runtime of logic synthesis in MVSIS (script *mvsis.rugged*), ABC (*resyn2*), and, as a sanity check, the runtime of standard-cell technology mapping in ABC (command *map –s*). All runtimes are on a 1.6GHz laptop.

In summary, AIG rewriting as implemented in ABC (*resyn2*) performs 10 passes over the network to improve area and delay of the AIG. It is much faster than the resource-aware traditional logic synthesis script in MVSIS.

4.2 Comparison using MCNC benchmarks

In Table 1, we compare the average ratios of improvements achieved by technology mapping for standard cells and FPGAs after running several optimization scripts. The complete set of MCNC benchmarks [19] is used in this experiment. The results of mapping unoptimized circuits are used as the base for comparison (Line 1 of Table 1). The optimization in SIS (*script.rugged*) did not complete on several benchmarks, which were excluded.

The last column shows the average ratios of runtime using AIG rewriting (*resyn2*) as the base. On these relatively small benchmarks, MVSIS is 7 times slower while SIS is slower by several orders of magnitude, depending on the script used. In terms of quality, rewriting tends to produce better area and worse delay than the combination of *script.rugged* followed by *speed_up* in SIS. It is likely that a more powerful rewriting that uses larger cuts will outperform SIS in delay while taking only a small fraction of the SIS runtime.

Table 1. Summary of comparison on MCNC benchmarks.

Logic synthesis flow used for optimization	Stand. cells		FPGAs		Runtime
	Area	Delay	Area	Delay	
No optimization	1.00	1.00	1.00	1.00	0.00
ABC (AIG rewriting)	0.87	0.96	0.93	0.98	1.00
MVSIS (mvsis.rugged)	0.91	1.10	0.93	1.03	7.12
SIS (script.delay)	0.94	0.99	0.98	0.97	~100.00
SIS (script.rugged+speed_up)	0.94	0.90	0.98	0.94	~1000.00

4.3 Comparison using IWLS 2005 benchmarks

This section compares AIG rewriting in ABC with logic synthesis in MVSIS on the large benchmarks from IWLS 2005. A similar comparison proved impossible for ABC vs. SIS because several key commands in SIS timed out on circuits from this set.

The following notation is used in Table 3. Columns "Original", "MVSIS", and "ABC" show the results of mapping of the original circuit, the circuit optimized by *mvsis.rugged* in MVSIS, and the same circuit optimized by *resyn2* in ABC, respectively. Two sets of mapping results are reported, one for LUT-based FPGAs and another for standard cells using *mcnc.genlib*.

In summary, the ratios of improvements demonstrate that on average, AIG rewriting performs better than traditional synthesis. In particular, the results of technology mapping for FPGAs confirm that literal-based optimization in MVSIS does not reduce area and delay while AIG rewriting reduces both.

It should be noted that the original IWLS benchmarks were optimized by an industrial tool prior to distribution. They were structurally hashed and balanced in ABC before running SIS and MVSIS. When starting with unoptimized networks, the difference between rewriting and traditional synthesis should be greater.

5 CONCLUSIONS AND FUTURE WORK

This paper presents AIG rewriting, an innovative technique for combinational logic synthesis. The technique was inspired by research in the field of formal verification where a similar algorithm was used for fast compression of redundant logic circuits [3]. Our experiments show that AIG rewriting often leads to quality comparable or better than those afforded by 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 proposed technique plays the crucial role in a new logic synthesis flow [12] which may replace the traditional logic synthesis in the CAD tools. The extreme speed and good quality of the proposed algorithm might make the new flow useful in a variety of applications such as hardware emulation, estimation of design complexity, and equivalence checking [13].

Future work will include extending the baseline AIG rewriting to use larger cut sizes. The challenge is to search a much larger space of possible replacements while keeping runtime low in order to allow multiple optimization passes.

6 REFERENCES

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. December 2005 Release. http://www-cad.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] R. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

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

[6] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD '05,* pp. 519-526.

[7] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs", *IEEE Trans. CAD*, vol. 13(1), January 1994, pp. 1-12.

[8] J. Cortadella, "Timing-driven logic bi-decomposition", *IEEE TCAD*, vol. 22(6), June 2003, pp. 675-685.

[9] IWLS 2005 Benchmarks. http://iwls.org/iwls2005/benchmarks.html

[10] E. Y. Kukimoto, R. Brayton, P. Sawkar, "Delay-optimal technology mapping by DAG covering", *Proc. DAC '98*, pp. 348-351.

[11] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization", *DATE '05*, pp. 418-423.

[12] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure", *Proc. IWLS '06*. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_sls.pdf.

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

[14] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs", *FPGA '06*, pp. 41-49.

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

[16] MVSIS Group. *MVSIS: Multi-Valued Logic Synthesis System.* UC Berkeley. http://www-cad.eecs.berkeley.edu/mvsis/

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

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

[19] S. Yang. *Logic synthesis and optimization benchmarks*. Version 3.0. Tech. Report. Microelectronics Center of North Carolina, 1991.

Table 2. IWLS benchmark statistics, rewriting performance, and runtime comparison.

IWLS	Network statistics					First iteration (*rwz*)				Second iteration (*rwz*)				Runtime, s		
benchmarks	PI	PO	Latch	AND2	Lev	Cuts	Subgrs	Upds	Gain	Cuts	Subgrs	Upds	Gain	MVSIS	ABC	Map
ac97_ctrl	84	48	2199	14261	11	30583	114770	4105	3242	21081	102688	2108	135	20.46	1.13	0.91
aes_core	259	129	530	21125	21	64314	350417	10849	697	60386	331836	10205	141	175.80	5.54	1.57
des_perf	234	64	8808	76716	17	394629	1701867	37530	4935	381979	1687585	30212	969	1010.70	33.23	8.87
ethernet	98	115	2235	19654	27	55413	326972	7401	4619	38365	238450	4080	381	50.39	3.81	0.99
mem_ctrl	115	152	1083	15191	28	45670	297941	8257	5416	28759	188528	3762	686	17.20	1.95	0.72
pci_bridge32	162	207	3359	22742	22	67838	331636	7865	3624	53148	279038	5191	155	51.57	3.14	1.46
systemcaes	260	129	670	12279	44	48620	164882	4539	1186	39962	145844	3993	273	28.17	1.87	0.87
usb_funct	128	121	1746	15670	23	40237	195654	4679	1383	35017	171805	3730	325	66.59	2.29	0.96
vga_lcd	89	109	17079	126687	19	463129	2887484	51088	34208	292358	2077801	32477	145	1998.27	32.55	8.51
wb_conmax	1130	1416	770	47535	18	159658	978491	15460	1891	149295	993207	16907	862	2323.01	12.38	3.31
Ratio						**1.00**	**1.00**	**1.00**	**1.00**	**0.82**	**0.88**	**0.79**	**0.18**	**24.53**	**1.00**	**0.40**

Table 3. Effect of AIG rewriting on technology mapping for LUT-based FPGAs ($k = 5$) and standard cells (*mcnc.genlib*).

IWLS	Results of mapping into LUTs ($k = 5$)						Results of mapping into mcnc.genlib					
benchmarks	Original		MVSIS		ABC		Original		MVSIS		ABC	
	Area	Delay	Area	Delay	Area	Delay	Area	Delay	Area	Delay	Area	Delay
ac97_ctrl	3391	4	3864	5	3532	3	25961	9.20	23494	13.80	19491	8.30
aes_core	6772	7	7214	8	7180	6	39635	17.70	38855	20.30	38555	17.30
des_perf	19177	5	23406	5	19163	5	162228	14.10	155708	17.50	145133	14.80
ethernet	4665	9	5170	9	4297	8	33949	22.40	29180	24.40	23142	21.30
mem_ctrl	4854	9	4551	10	3191	9	25521	23.30	23537	26.50	15865	21.10
pci_bridge32	6150	8	5888	9	5908	7	40322	18.60	35254	20.60	34860	17.70
systemcaes	2547	9	2770	13	2329	10	21715	28.70	16483	34.60	16533	28.10
usb_funct	4530	7	4475	8	4030	7	27617	17.80	24386	28.30	23637	19.70
vga_lcd	28458	8	28866	8	29562	7	240071	15.70	169276	16.50	201141	15.50
wb_conmax	16073	7	17165	8	13370	7	82353	15.90	87082	17.60	66124	15.90
Ratio	**1.00**	**1.00**	**1.01**	**1.17**	**0.94**	**0.97**	**1.00**	**1.00**	**0.88**	**1.22**	**0.83**	**0.97**