Logic Optimization of Majority-Inverter Graphs

Heinz Riener[1], Eleonora Testa[1], Winston Haaswijk[1], Alan Mishchenko[2],
Luca Amarú[3], Giovanni De Micheli[1], Mathias Soeken[1]
[1]EPFL, Lausanne, Switzerland
[2]UC Berkeley, CA, United States
[3]Synopsys Inc, Sunnyvale, CA, United States

Kurzfassung

Majority-Inverter Graphen (MIGs) sind eine multi-level Logikrepräsentation von Booleschen Funktionen mit bemerkenswerten algebraischen und Booleschen Eigenschaften, die effiziente Logikoptimierungen über die Möglichkeiten traditioneller Logikrepräsentationen erlauben. In dieser Arbeit, überblicken wir zwei moderne Logikoptimierungsmethoden für MIGs: *cut rewriting* und *cut resubstitution*. Beide Algorithmen sind generisch und können auf beliebige graphbasierte Logikrepräsentationen angewandt werden. Wir beschreiben sie in einem vereinheitlichen Framework und präsentieren experimentelle Ergebnisse für Größenoptimierung von MIGs unter Verwendung der EPFL Benchmarks.

Abstract

Majority-inverter graphs (MIGs) are a multi-level logic representation of Boolean functions with remarkable algebraic and Boolean properties that enable efficient logic optimizations beyond the capabilities of traditional logic representations. In this paper, we survey two state-of-the-art logic optimization methods for MIGs: *cut rewriting* and *cut resubstitution*. Both algorithms are generic and can be applied to arbitrary graph-based logic representations. We describe them in a unified framework and show experimental results for MIG size optimization using the EPFL combinational benchmark suite.

1 Introduction

Logic optimization of multi-level Boolean networks plays an important role in automated design flows for digital systems and is responsible for substantial area and delay reductions [1, 2]. These logic optimizations are typically carried out on a simple and technology-independent representation of the digital logic. Particularly, homogeneous datastructures, such as *and-inverter graphs* (AIGs) [3, 4]—being composed of two-input ANDs and inverters—or *majority-inverter graphs* (MIGs) [5]—being composed of majority-of-three gates and inverters—have been proven to be successful. Structural hashing on the intermediate representation ensures that no two nodes have identical incoming edges. Arbitrary Boolean networks can be transformed into AIGs or MIGs, for which a repertoire of scalable optimization techniques is available [6].

In particular MIGs, recently, received much attention due to their remarkable algebraic and Boolean properties. On the one hand MIGs share many characteristics of AIGs such that simple and efficient optimization are possible, on the other hand MIGs generalize AIGs and enables a more compact representation of logic functions. The logic AND $x \wedge y$ of two functions x and y can be represented with a majority expression $\langle 0xy \rangle$ by assigning the third input to constant 0. Consequently, all AIGs are convertible to MIGs without increasing the number of nodes. Figure 1 illustrates the compactness of MIGs by showing the function $\text{prime}_5(x_1,\dots,x_5) = [(x_5 \dots x_1)_2 \text{ is prime}]$ represented

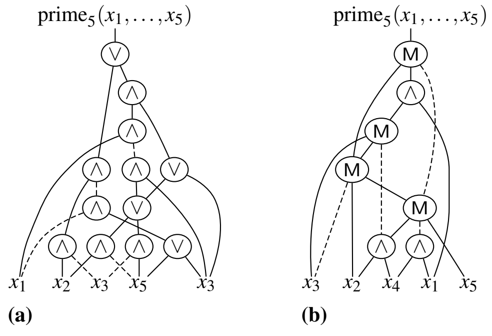

Figure 1 Example of a (a) logic representation using AND, OR, and complemented edges and a (b) MIG representation for function $\text{prime}_5(x_1,\dots,x_5) = [(x_5 \dots x_1)_2 \text{ is prime}]$. Majority-3, AND, and OR nodes are distinguished by label M, $\wedge$, and $\vee$, respectively. Complemented edges are drawn using dashed lines.

using ANDs, ORs, and complemented edges (on the left) and as MIGs (on the right).

The focus of this paper lies on two Boolean optimization techniques:

1) *Boolean rewriting* is a coarse-grained optimization technique that iteratively selects small parts of a Boolean network and replaces them with more compact implementations in order to reduce the overall number of nodes, while maintaining the global output functions

of the Boolean network.

2) *Boolean resubstitution* is a more fine-grained technique that reexpresses the Boolean functions of particular nodes using nodes already present in the Boolean network. Nodes which are no longer used (including nodes in the transitive fan-ins) can then be removed from the Boolean network.

Effective implementation of both ideas are available for AIGs [6, 7], which exploit peephole optimization techniques using cuts, truth tables, and pre-computation in order to scale to large Boolean networks.

We survey state-of-the-art generalization of Boolean rewriting and Boolean resubstitution applicable to arbitrary graph-based logic representations. In particular, we discuss *cut rewriting* [8], an on-the-fly rewriting technique using exact synthesis, and *cut resubstitution* [9], a scalable rule-based resubstitution technique. Both techniques are DAG-aware and exploit structural hashing to obtain a gain even when a smaller part of logic is replaced with a larger one, by reusing already existing logic in the Boolean network. The two techniques are implemented in the EPFL logic synthesis library *mockturtle*[1] [10]. In experiments using the EPFL combinational benchmarks suite, we show that the proposed techniques are capable of reducing the benchmark's size by 23.54% in 392.72s when applied interleaved until convergence.

2 Preliminaries

A *Boolean network* N is a *directed acyclic graph* (DAG). Each node corresponds to a logic gate. Each directed edge (n,m) is a wire connecting node n with node m. The *fanin*, respectively *fanout*, of a node $n \in N$ are the incoming, respectively outgoing, edges of the node. The *primary inputs* (PIs) are the nodes of the Boolean network without fanin. The *primary outputs* (POs) are the nodes of the Boolean network without fanout. All other nodes in the Boolean network are *gates*.

A *cut* is a pair (r,L) where r is a node, called *root*, and L is a set of nodes, called *leaves*, such that 1) each path from any primary input to r passes through at least one leaf and 2) for each leaf $l \in L$, there is at least one path from a primary input to r passing through l and not through any other leaf. The *cover* $\mathrm{N.cover}(r,L)$ of a cut (r,L) of network N is the set of all nodes $n \in N$ that appear on a path from any $l \in L$ to r including r, but excluding the leaves.

A *fanout-free cone* (FFC) of a node r is a cut (r,L) such that no node $r' \in \mathrm{N.cover}(r,L)$ with $r' \neq r$ has a parent node that is outside of $\mathrm{N.cover}(r,L)$. The *maximum fanout-free cone* (MFFC) of a node r is its largest FFC. In other words, the MFFC of a node contains all the logic used exclusively by the node. When a node is removed or substituted, the logic in its MFFC can be removed [11].

[1]EPFL Logic Synthesis Lib., https://github.com/lsils/lstools-showcase

3 Cut Rewriting

Algorithm 1 shows the pseudo code of cut rewriting. The algorithm starts by enumerating all cuts of network N with cut size l and cut limit p using cut enumeration techniques [12, 13, 11].

Since cuts found by cut enumeration may not be an FFC, DAG-aware rewriting techniques [7] are used to compute the gain of possible replacement candidates. After all replacement candidates and their gain have been computed, the algorithm finds a set of replacement candidates that maximizes the overall gain.

Next, an empty graph $G(V,E)$ is initialized that will be constructed when enumerating replacement candidates for the cuts. The graph has vertices V for cuts, and an edge in E if two cuts have overlapping logic and can therefore not be replaced simultaneously. Each vertex is also assigned to a root node r' of a best replacement candidate and the potential gain when being replace by r'. The replacements for the cuts are constructed in the network with dangling root nodes while computing the potential gains. On termination, all remaining dangling nodes are recursively removed from the network.

For each cut (r,L) the algorithm enumerates possible replacements (r',L) either looking the replacements up from a pre-computed database of best implementations or on-the-fly using SAT-based exact synthesis. The replacements must not necessarily be optimum in size. The runtime of exact synthesis can be controlled by setting thresholds on the conflict limit of the SAT solver. For each replacement candidate the gain is stored in a variable together with the best replacement candidate [14]. If a replacement with root r' that leads to a gain can be found, a vertex (r,L,r') for the cut is added to G, i.e., the cut (r,L) can be replaced by the cut (r',L). Afterwards edges are added to G for each two cuts that have overlapping covers. To obtain a good subset of non-conflicting replacement candidates we heuristically solve the maximum weighted independent vertex set problem on G with respect to the gain weights in the graph using the greedy algorithm GWMIN [15], which provides an approximation guarantee of finding a solution with a weight of at least $\frac{1}{\Delta}\alpha(G)$, where Δ is the degree of G and $\alpha(G)$ is the weight of the exact solution.

4 Cut Resubstitution

Algorithm 2 shows the pseudo code of cut resubstitution. The algorithm iterates over all nodes r in a given network N, identifies possible node replacements r' of r using existing logic in N, and resubstitutes r with r' if the overall number of nodes in the logic network is reduced.

For each node r, first a reconvergence-driven cut [6] is computed restricted with cut size limit k. Next, from the same node r, an MFFC M is constructed to estimate how many nodes can be freed if r is replaced. Each node of the cut, which is not part of M, is considered a potential candidate for replacing r and added to a list D of divisors.

The local functions of the nodes n within the reconvergence-driven cut are computed using truth

```
Input : Boolean network $N$, cut size $k$, cut limit $p$
Set $C \leftarrow$ N.enumerateCuts$(k,p)$;
Set $T \leftarrow$ N.simulateCuts(C);
Set $G \leftarrow (V = \emptyset, E = \emptyset)$;
foreach node $r \in N$ do
    Set $M \leftarrow$ N.computeMFFC$(r)$;
    if $|M| = 1$ then continue;
    foreach leaves $L \in C(r)$ do
        $r' \leftarrow$ N.computeBestReplacement$(r,L,T)$;
        if $r' \neq \bot$ then $G$.addVertex$(r,L,r')$;
    end
end
foreach $L_1 \in C(r_1)$ and $L_2 \in C(r_2)$ do
    if N.cover$(r_1,L_1) \cap$ N.cover$(r_2,L_2) \neq \emptyset$ then
        G.addEdge$(r_1,L_1) — (r_2,L_2)$;
    end
end
Set $V' \leftarrow$ G.maximalIndependentVertexSet();
foreach $(r,L,r') \in V'$ do
    N.replaceNode$(r,r')$;
end
return $N$;
```

Algorithmus 1 : Cut rewriting

```
Data : Logic network $N$, cut size $k$
Result : Optimized logic network
foreach node $r \in N$ do
    Set $L \leftarrow$ N.computeReconvDrivenCut$(r,k)$;
    Set $M \leftarrow$ N.computeMFFC$(r,L)$;
    Set $D \leftarrow$ N.collectDivisors$(r,L,M)$;
    Set $T \leftarrow$ N.simulate$(L,D)$;
    Set $r' \leftarrow$ N.resubKernel$(r,D,M,T)$;
    if $r' \neq \bot$ then N.replaceNode$(r,r')$;
end
return $N$;
```

Algorithmus 2 : Cut resubstitution

tables. The core of the algorithm is a rule-based *resubstitution kernel* that identifies possible replacements of r using divisors in D. If a possible replacement r' is found by the resubstitution kernel, then r is replaced with r' and the Boolean network is updated. If no replacement is found (i.e., the kernel returns $\bot$), then the algorithm continues with the next node.

The actual resubstitutions are computed by the resubstitution kernel that compares divisors and suggests possible replacements. The resubstitution kernel contains those parts of the resubstitution algorithm, which have to be customized for the logic representation in use. In particular, a resubstitution kernel defines resubstitution rules and filtering rules:

1. A *resubstitution rule* is a simple, repetitive test to determine if a given node can be reexpressed with divisors using a fixed resubstitution pattern. For instance, a 1-AND resubstitution rule tests for each pair of candidate divisors $d_1, d_2 \in D$ with $d_1 \neq d_2$ if $r = d_1 \wedge d_2$.

2. A *filtering rule* implements a necessary or sufficient condition to pre-filter the divisors in D with the objective to reduce the number of tests in resubstitution rules. For instance, in order to speed-up 1-AND resubstitution, one may pre-compute those divisors $U \subset D$ that imply r, i.e.,

$$d \in U \text{ iff } d \in D \wedge d \implies r.$$

   Filtering rules lead to performance improvements if the filters can be leveraged by multiple resubstitution rules.

For MIGs, we consider five resubstitution rules:

1. *Constant resubstitution* replaces r if equivalent to a Boolean constant 0 or 1.

2. *Divisor resubstitution* replaces r if equivalent to a divisor in the current cut or its complement.

3. *Relevance resubstitution* replaces r if one of its children can be replaced by a divisor.

4. 1-*MAJ resubstitution* replaces r with one newly added majority gate using three divisors from the current cut.

5. 2-*MAJ resubstitution* replaces r with two newly added majority gates using five divisors from the current cut.

5 Experiments

We have implemented the proposed algorithms in C++-17 using the EPFL logic synthesis library[10] *mockturtle* in a generic way such that they can in principle be applied to arbitrary logic representations.

We present MIG size optimization results for the EPFL combination benchmark suite. We apply cut rewriting (RW) using a database of best MIGs [8] and cut resubstitution (RS) using a resubstitution kernel specifically designed for MIGs [9], which adds at most two MIG nodes to the Boolean network. Both techniques, RW and RS, are applied to the Boolean network interleaved until convergence. Table 1 is organized as follows: the first three columns name the benchmarks (**Name**) and show the initial size (**Size**) and depth (**Depth**) of the circuits. The next four columns present the results after size optimization, i.e., the reduced size (**Size**) and depth (**Depth**) of the benchmarks, the number of iterations until convergence (**It**), and the runtime (**Time**). One iteration refers to one execution of cut rewriting and cut resubstitution. The last column (**Improv.**) shows the size reduction when compared to the initial benchmarks. Overall, the proposed size optimization flow achieves a size reduction of 23.54% (108954 MIG nodes) in 392.72s.

6 Conclusion

We have presented two state-of-the-art methods for logic optimizing of Boolean networks: *cut rewriting* and *cut resubstitution*. Both techniques are generic and can be applied to arbitrary logic representations.

Both algorithms leverage DAG-awareness, cut-based computations, and truth tables to scale to large Boolean networks.

Table 1 Size Optimization of EPFL Benchmarks

Benchmark			$(\mathbf{RW} \cdot \mathbf{RS})^+$				Improv.
Name	**Size**	**Depth**	**Size**	**Depth**	**It**	**Time** [s]	η_a**(Size)** [%]
adder	1020	255	512	130	3	0.14	49.80
arbiter	11839	87	11839	87	1	2.10	0.00
bar	3336	12	3073	13	2	0.65	7.88
cavlc	693	16	602	16	4	3.49	13.13
ctrl	174	10	81	10	3	0.04	53.45
dec	304	3	304	3	1	0.02	0.00
div	57247	4372	36154	4337	6	46.52	36.85
hyp	214335	24801	162416	16795	4	251.36	24.22
i2c	1342	20	1180	18	5	0.51	12.07
int2float	260	16	209	16	3	0.10	19.62
log2	32060	444	30387	422	6	25.07	5.22
max	2865	287	2301	208	4	1.02	19.69
mem_ctrl	46836	114	41757	113	5	25.58	10.84
multiplier	27062	274	24496	273	4	12.78	9.48
priority	978	250	683	181	5	0.39	30.16
router	257	54	244	53	2	0.07	5.06
sin	5416	225	4910	196	13	9.13	9.34
sqrt	24618	5058	11433	4131	5	14.05	53.56
square	18484	250	17137	131	4	8.03	7.29
voter	13758	70	4822	53	10	6.04	64.95
Total	462884		353930			392.72	23.54

We have described both algorithm in a unified framework and have shown experimental results for MIG size optimization using the EPFL combinational benchmark suite.

7 Acknowledgments

This research was supported by the Swiss National Science Foundation (200021-169084 MAJesty); by the European Research Council in the project H2020-ERC-2014-ADG 669354 CyberCare and by SRC contracts "SAT-based methods for scalable synthesis and verification" and "Deep integration of computation engines for scalability in synthesis and verification".

8 Literatur

[1] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 1990.

[2] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[3] L. Hellerman, "A catalog of three-variable Or-invert and And-invert logical circuits," *TEC*, vol. 12, no. 3, pp. 198–223, 1963. [Online]. Available: http://dx.doi.org/10.1109/PGEC.1963.263531

[4] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *TCAD*, vol. 21, no. 12, pp. 1377–1394, 2002. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2002.804386

[5] L. Amarú, P. Gaillardon, and G. De Micheli, "Majority-Inverter Graph: A New Paradigm for Logic Optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2016.

[6] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Int'l Workshop on Logic and Synthesis*, 2006, pp. 15–22.

[7] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *DAC*, 2006, pp. 532–535. [Online]. Available: http://doi.acm.org/10.1145/1146909.1147048

[8] H. Riener, W. Hasswijk, A. Mishchenko, G. De Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis," in *DATE*, 2019, p. To appear.

[9] H. Riener, E. Testa, L. Amaru, M. Soeken, and G. De Micheli, "Size optimization of MIGs with an application to QCA and STMG technologies," in *NANOARCH*, 2018.

[10] M. Soeken, H. Riener, W. Haaswijk, and G. De Micheli, "The EPFL logic synthesis libraries," *Computer Science - Logic in Computer Science*, vol. abs/1805.05121, 2018. [Online]. Available: http://arxiv.org/abs/1805.05121

[11] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton, "Combinational and sequential mapping with priority cuts," in *ICCAD*, 2007, pp. 354–361. [Online]. Available: http://dx.doi.org/10.1109/ICCAD.2007.4397290

[12] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," *TVLSI*, vol. 2, no. 2, pp. 137–148, 1994. [Online]. Available: http://dx.doi.org/10.1109/92.285741

[13] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *FPGA*, 1999, pp. 29–35. [Online]. Available: http://doi.acm.org/10.1145/296399.296425

[14] W. Haaswijk, A. Mishchenko, M. Soeken, and G. De Micheli, "SAT based exact synthesis using DAG topology families," in *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, 2018, pp. 53:1–53:6. [Online]. Available: https://doi.org/10.1145/3195970.3196111

[15] S. Sakai, M. Togasaki, and K. Yamazaki, "A note on greedy algorithms for the maximum weighted independent set problem," *Discrete Applied Mathematics*, vol. 126, no. 2-3, pp. 313–322, 2003. [Online]. Available: https://doi.org/10.1016/S0166-218X(02)00205-6