Logic Optimization of Majority-Inverter Graphs

Heinz Riener¹, Eleonora Testa¹, Winston Haaswijk¹, Alan Mishchenko², Luca Amarú³, Giovanni De Micheli¹, Mathias Soeken¹
¹EPFL, Lausanne, Switzerland
²UC Berkeley, CA, United States
³Synopsys Inc, Sunnyvale, CA, United States

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 conventional 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 commonly carried out on a simple and technology-independent representation of the digital logic. Particularly, homogeneous data-structures, 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].

Recently, MIGs have 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 enable a more compact representation of logic functions. The logic AND \( x \land y \) 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 \( \text{prime}_5(x_1, \ldots , x_5) = \lceil (x_5 \ldots x_1) \text{ is prime} \rceil \) represented using AND, OR, 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 [9, 10], and cut resubstitution [11], 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\textsuperscript{1} [12]. In our 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 $N.\text{cover}(r,L)$ 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 $r' \in N.\text{cover}(r,L)$ with $r' \neq r$ has a parent node that is outside of $N.\text{cover}(r,L)$. 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 [13].

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 [14, 15, 13].

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 are not required to be size optima. 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 [9]. If a replacement with root $r'$ that leads to a gain can be found, a vertex $(r,L)$ 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 [16], which provides an approximation guarantee of finding a solution with a weight of at least $\frac{1}{2}\alpha(G)$, where $\Delta$ is the degree of $G$ and $\alpha(G)$ is the weight of the globally optimum 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 $C = N$.enumerateCuts($k, p$);
Set $T = N$.simulateCuts($C$);
Set $G = \{(V = \emptyset, \emptyset)\}$

**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 $r' \neq \perp$ 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($r_1, L_1$)$\rightarrow$(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 $L \leftarrow N$.computeReconDrivenCut($r, k$);
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 $r' \neq \perp$ then $N$.replaceNode($r, r'$);

end

return $N$:

**Algorithmus 2** : Cut resubstitution

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 $\perp$), 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 $r = d_1 \land d_2$.

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 $U \subseteq D$ that imply $r$, i.e.,

$$d \in U \text{ if and only if } d \in D \land d \implies r.$$ 

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[12] 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 [11], 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 an average 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 | Size (| RW | RS | Total | Improv. | \( \eta_{|L|} \):Size (%) |
|-----------|------|----|----|-------|--------|----------------------|
| addr      | 1020 | 255 | 512 | 130   | 0.14   | 49.85                |
| arbiter   | 11839| 87  | 11839| 87    | 2.10   | 0.00                 |
| bar       | 3336 | 12  | 3073 | 13    | 0.65   | 7.88                 |
| cavlc     | 693  | 16  | 602  | 16    | 3.49   | 13.13                |
| ctrl      | 174  | 10  | 81   | 10    | 0.04   | 53.45                |
| dec       | 304  | 3   | 304  | 3     | 0.02   | 0.00                 |
| div       | 57247| 4732| 36154| 4337  | 6      | 46.52                |
| hyp       | 214335| 2481| 162416| 16795| 4      | 25.36                |
| i2c       | 1342 | 20  | 1180 | 18    | 0.51   | 12.07                |
| int2float | 260  | 16  | 209  | 16    | 0.10   | 19.62                |
| log2      | 32060| 444 | 30387| 422   | 6      | 5.22                 |
| max       | 2865 | 287 | 2301 | 208   | 4      | 19.69                |
| mem_ctrl  | 46836| 114 | 41757| 113   | 5      | 25.58                |
| multiplier| 27062| 274 | 24496| 273   | 4      | 12.78                |
| priority  | 978  | 260 | 683  | 181   | 0.39   | 30.16                |
| router    | 5416 | 225 | 4910 | 196   | 5      | 9.34                 |
| sin       | 5416 | 225 | 4910 | 196   | 5      | 9.34                 |
| sqrt      | 24618| 5058| 11433| 4131  | 5      | 14.05                |
| sqrt2     | 260  | 16  | 209  | 16    | 0.10   | 19.62                |
| square    | 18484| 250 | 17137| 131   | 4      | 8.03                 |
| voter     | 13758| 70  | 482  | 53    | 10     | 64.95                |
| Total     | 462884| 353930| 392.72| 23.54 |        |                      |

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 “SA T-Based Exact Synthesis: Encodings, Topology Families, and Parallelism,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019, to appear.

8 Literatur