Optimizing Majority-Inverter Graphs With Functional Hashing

Mathias Soeken[1,2] Luca Gaetano Amarù[1] Pierre-Emmanuel Gaillardon[1] Giovanni De Micheli[1]
[1]Integrated Systems Laboratory, EPFL, Switzerland
[2]Group of Computer Architecture, University of Bremen, Germany
{mathias.soeken,pierre-emmanuel.gaillardon,luca.amaru,giovanni.demicheli}@epfl.ch

***Abstract*—A *Majority-Inverter Graph* (MIG) is a recently introduced logic representation form whose algebraic and Boolean properties allow for efficient logic optimization. In particular, when considering logic depth reduction, MIG algorithms obtained significantly superior synthesis results as compared to the state-of-the-art approaches based on AND-inverter graphs and commercial tools. In this paper, we present a new MIG optimization algorithm targeting size minimization based on functional hashing. The proposed algorithm makes use of minimum MIG representations which are precomputed for functions up to 4 variables using an approach based on *Satisfiability Modulo Theories* (SMT). Experimental results show that heavily-optimized MIGs can be further minimized also in size, thanks to our proposed methodology. When using the optimized MIGs as starting point for technology mapping, we were able to improve both depth and area for the arithmetic instances of the EPFL benchmarks beyond the current results achievable by state-of-the-art logic synthesis algorithms.**

I. Introduction

Due to their simplicity, homogeneous logic representations have attracted scientists in the area of logic synthesis. In the combinational case, logic networks are modeled as a *Directed Acyclic Graph* (DAG) in which terminal nodes represent primary inputs and constants and all other nodes represent the same Boolean logic operation. Edges may be complemented in order to guarantee a functional universal representation. Primary outputs are (potentially complemented) pointers to any node in the DAG.

The major advantage of homogeneous logic representations is that they simplify manipulation algorithms significantly and particularly enable an efficient implementation of them. Popular instances of homogeneous logic representations are NAND and NOR circuits [1] or *AND-Inverter Graphs* (AIGs) [2]. All of them implement a binary Boolean operation, i.e., each non-terminal node has two incoming logic operands (fanin). Recently, *Majority-Inverter Graphs* (MIGs) [3] have been proposed, which implement the ternary majority function as logic operation. MIG logic manipulation is supported by a consistent algebraic framework. By using MIG algebra axioms, it is possible to reach all points in the representation space [3]. Such remarkable algebraic property enables strong synthesis results as compared to AIG techniques [3]. MIG global properties also permit the insertion of certain type of logic errors without affecting the target functionality [4]. For instance, orthogonal logic errors can be safely inserted in MIGs because they are successively masked by the voting nature of majority nodes. On the other hand, orthogonal logic errors enable strong logic simplifications, which are very useful to MIG Boolean optimization [4].

MIG algebraic and Boolean methods together attain high optimization quality. For example, when targeting depth reduction, MIG optimization was shown capable of rewriting a ripple carry structure into a carry look-ahead-like one [4]. In general, the proposed algorithm allows for significant depth reductions with much simpler algorithms as compared to AIGs [4]. However, the same logic rewriting idea has yet to be shown for reduction of MIG size, another important metric to measure the quality of the logic representation.

In this paper, we present an effective algorithm to reduce size in MIGs based on functional hashing. *Functional hashing* describes rewriting based on functionally equivalent substructures. The main idea of our algorithm is to enumerate all subgraphs of an MIG, called *cuts*, with up to 4 inputs and replace them with precomputed equivalent minimal representations, if applicable. In this paper, we discuss several variants of the algorithm.

In order to find minimum MIGs for 4-input functions we have combined *Satisfiability Modulo Theories* (SMT) [5] techniques with NPN classification. The latter is used to reduce the search space, as inverting and permuting inputs or outputs preserves the size of an MIG.

Experimental results show that heavily-optimized MIGs can be further optimized in size, thanks to our proposed methodology. A special depth-preserving variant allows for a size reduction by not increasing the depth or keeping the increase low—typically size and depth are considered contrary objectives. Supplementary size reductions of 8%, on average, are reported on the arithmetic instances of the EPFL benchmark suite. When using the optimized MIGs as starting point for technology mapping, we were able to reduce the area for 7 out of existing 8 best results of the arithmetic EPFL benchmarks, and on top of that, we were able to reduce depth for the instance *Log2*. The current best results are produced by the strongest AIG and MIG optimization scripts from Berkeley and EPFL groups, hence, any improvement advances the state-of-the-art in logic optimization.

In summary, the main contributions of this paper are:

1) An SMT-based algorithm to find minimum-size MIGs (also applicable to functions with more than 4 inputs).
2) Several variants of a size optimization approach for MIGs based on functional hashing.
3) An upper bound for the size of MIGs which makes use of all 4-input minimum-size MIG representations.
4) New best results for a public benchmark suite.

This research was supported by ERC-2009-AdG-246810.

II. Background

A. Related Work

Most similar to our proposed approach is DAG-aware AIG rewriting [6]. This rewriting approach also enumerates all 4-input subgraphs and replaces them by smaller precomputed minimal representations. However, not all functions are considered but only those that are determined *useful* by experimental observation. Further, only a bottom-up greedy approach is implemented, while we present several variants of functional hashing. In [6], delay is explicitly considered by only allowing depth-preserving rewrites. Further, depth is optimized by integrating AIG balancing using algebraic tree-height reduction [7] as part of the algorithm. The approach in [6] extends [8] which presents a similar rewriting algorithm that is applied in a top-down manner. The approach in [8] does not consider depth-preserving rewrites. In [9], the approach in [6] is extended to 5-input cuts.

None of the two approaches makes explicit use of a partitioning into fanout-free regions. We propose variants of the functional hashing algorithm that work on fanout-free regions and the experimental evaluations show that this is often advantageous as compared to rewriting the whole MIG at once in terms of quality of the resulting optimized MIG.

B. Majority-Inverter Graphs

The majority function of three Boolean variables a, b, and c, denoted $\langle abc \rangle$, evaluates to true if and only if at least two of the three variables are true. The majority function is self-dual and can be expressed in disjunctive and conjunctive normal form as:

$$ab \vee ac \vee bc = (a \vee b)(a \vee c)(b \vee c), \tag{1}$$

Setting any variable to 0 gives the conjunction of the other two variables, and analogously one obtains the disjunction by setting any variable to 1, i.e., $\langle 0ab \rangle = a \wedge b$ and $\langle 1ab \rangle = a \vee b$.

We introduce a formal definition for MIGs. A *Majority-Inverter Graph* (MIG) over the primary input variables $X = \{x_1, \ldots, x_n\}$ is a DAG $M = (V, E, Y)$ with

- a finite set of *nodes* $V = X \cup G \cup \{0\}$, where $G = \{g_1, \ldots, g_k\}$ are non-terminal nodes representing the logic operations in the graph and 0 is the constant 0 input,
- a finite multiset of *edges* $E \subseteq G \times (V \times \mathbb{B})$, where the first element in the tuple is a source node and the second element is a pair of a target node and a polarity bit,
- and a finite multiset of *outputs* $Y \subseteq V \times \mathbb{B}$.

Each operation $g \in G$ must have three successors, called $\mathrm{adj}(g)$. For convenience, we denote an output $(v, p) \in (V \times \mathbb{B})$ as v^p or v if $p = 1$ and as $\bar{v}$ if $p = 0$. The functional semantics of an MIG is described in terms of an interpretation function Φ that maps MIG nods to a Boolean function:

$$\begin{aligned}
&\Phi(0) = \bot \\
&\Phi(x) = x && \text{for } x \in X \\
&\Phi(g) = \langle \Phi^{p_1}(g_1)\Phi^{p_2}(g_2)\Phi^{p_3}(g_3) \rangle && \text{for } g \in G \\
&\qquad \text{with } \mathrm{adj}(g) = \{(g_1, p_1), (g_2, p_2), (g_3, p_3)\} \\
&\Phi(y) = \Phi^p(g) && \text{for } y = (g, p) \in Y
\end{aligned}$$

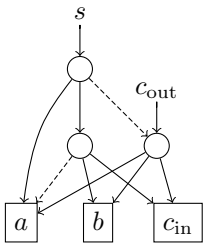

Fig. 1. MIG for a full adder

Example 1: Fig. 1 shows an example of an MIG for a full adder that computes $a+b+c_{\mathrm{in}} = 2c_{\mathrm{out}}+s$, i.e., $s = a \oplus b \oplus c_{\mathrm{in}}$ and $c_{\mathrm{out}} = \langle abc_{\mathrm{in}} \rangle$. Circles represent majority operations and boxes represent primary inputs and the constant node, if used. Complemented edges are drawn dashed and primary outputs are (possibly complemented) pointers to a node. This MIG has a size of 3 and a depth of 2.

C. Cut Enumeration

Given an MIG $M = (V, E, Y)$, a pair (v, L) consisting of a *root* $v \in V$ and *leafs* $L \subseteq V \setminus \{0\}$ is called a *cut* (see, e.g., [10]), if

1) all paths from v to a non-terminal visit at least one leaf $l \in L$,
2) and each leaf is contained in at least one path.

Paths to the constant node are exempt from these constraints. A cut is called k-feasible, if $|L| \leq k$ and we denote all k-feasible cuts of a node $v \in V$ as

$$\mathrm{cuts}_k(v) = \{L \mid (v, L) \text{ is cut and } |L| \leq k\}. \tag{2}$$

Each cut (v, L) describes a subgraph which may have nodes apart from v and L. These nodes are called internal nodes. All k-feasible cuts can be generated using the recursive algorithm

$$\begin{aligned}
&\mathrm{cuts}_k(0) = \{\{\}\} \\
&\mathrm{cuts}_k(x) = \{\{x\}\} && \text{for } x \in X \\
&\mathrm{cuts}_k(g) = \mathrm{cuts}_k(g_1) \otimes_k \mathrm{cuts}_k(g_2) \otimes_k \mathrm{cuts}_k(g_3) && \text{for } g \in G
\end{aligned}$$

in a depth-first manner starting from the outputs, where g_1, g_2, and g_3 are the child nodes of g. The operation

$$M_1 \otimes_k M_2 = \{m_1 \cup m_2 \mid m_1 \in M_1, m_2 \in M_2, |m_1 \cup m_2| \leq k\}$$

is a saturating union over all combinations of subsets. An exhaustive enumeration of all cuts in an MIG is feasible as long as $k \leq 6$. Since we are interested in inspecting all subgraphs with four inputs, we enumerate all 4-feasible cuts. For k larger than 6, priority cuts provide an alternative approach [11].

D. NPN Classification

Two functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ are NPN-equivalent, if there exists a permutation $\sigma \in S_n$ and polarities $p, p_1, \ldots, p_n \in \mathbb{B}$ such that

$$f(x_1, \ldots, x_n) = g^p(x_{\sigma(1)}^{p_1}, \ldots, x_{\sigma(n)}^{p_n}), \tag{3}$$

i.e., g can be made equivalent to f by negating inputs, permuting inputs, or negating the output. NPN-equivalence is an equivalence relation that partitions the set of all Boolean functions over n variables into a smaller set of NPN classes. As an example all 2^{2^n} Boolean functions over n variables

can be partitioned into 2, 4, 14, 222, 616126 NPN classes for $n = 1, 2, 3, 4, 5$. As the representative of each NPN class, we take the function with the smallest truth table, when truth tables are viewed as a binary number of 2^n bits. For a detailed introduction into NPN classification the reader is referred to [12], [13].

III. Exact Synthesis

We are interested in finding the smallest MIG w.r.t. the size for a given Boolean function, called exact synthesis. One way to find such a smallest MIG is to formulate a decision problem that asks whether there exists an MIG with k nodes that can represent f. To find a minimum solution, one starts by solving the decision problem for $k = 0$ and increases k until a satisfying solution is found.

This section describes the formulation of exact synthesis a as decision problem that can be automatically solved with an SMT solver. The instance is a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ and a non-negative constant k. In other words, the exact synthesis problem asks whether there exists an MIG

$$M = (\{x_1, \ldots, x_n\} \cup \{g_1, \ldots, g_k\} \cup \{0\}, E, \{y\})$$

with k majority operations that represents f, i.e., $\Phi(y) = f$. For the encoding of the SMT formulation, we assume that $k > 0$ and, in our algorithm, we check for the case $k = 0$, i.e., $f = 0^p$ or $f = x_i^p$, explicitly.

Each majority node with index $l \in \{1, \ldots, k\}$ is duplicated for each function value $0 \leq j < 2^n$ and is represented by 10 variables:

- three inputs $a_{1,l}^{(j)}, a_{2,l}^{(j)}, a_{3,l}^{(j)} \in \mathbb{B}$ of gate l,
- one output $b_l^{(j)} \in \mathbb{B}$ of gate l,
- three select variables $s_{1,l}, s_{2,l}, s_{3,l} \in \mathbb{B}^{\lceil \log_2(n+l) \rceil}$ that encode which are the child nodes of gate l, and
- polarity variables $p_{1,l}, p_{2,l}, p_{3,l} \in \mathbb{B}$ that describe whether the edges to the child nodes are complemented.

There is one single variable $p \in \mathbb{B}$ that encodes the polarity of the output edge of the root node, which represents y.

The node indexes form a topological ordering of the nodes. The following constraints are contained in the SMT formula. Index j ranges from 0 to $2^n - 1$ and index l ranges from 1 to k.

Majority functionality: The formula

$$b_l^{(j)} \leftrightarrow \langle a_{1,l}^{(j)} a_{2,l}^{(j)} a_{3,l}^{(j)} \rangle \tag{4}$$

ensures the correct functionality of the node, i.e., the output value of the l^{th} node $b_l^{(j)}$ is the majority of the node's three input values $a_{1,l}^{(j)}$, $a_{2,l}^{(j)}$, and $a_{3,l}^{(j)}$ for all assignments j.

Input connections: Constraints on the input connections are given in terms of implications of the select variables $s_{c,l}$, where c ranges from 1 to 3. The formula

$$s_{c,l} < n + l \tag{5}$$

ensures that node inputs can only be the constant, primary inputs, or topologically smaller nodes. In other words, the constraint prohibits cycles in the MIG. A value 0 for $s_{c,l}$ implies a connection to the constant node, i.e.,

$$(s_{c,l} = 0) \to (a_{c,l}^{(j)} = \bar{p}_{c,l}). \tag{6}$$

Values from 1 to n imply a connection to a variable node, i.e.,

$$(s_{c,l} = i) \to (a_{c,l}^{(j)} = \text{bv}(j)_{i-1} \oplus \bar{p}_{c,l}) \quad \text{for } 1 \leq i \leq n, \tag{7}$$

where $\text{bv}(j)_{i-1}$ refers to the $(i-1)^{\text{th}}$ bit in the binary representation of j. All other values to $s_{c,l}$ imply a connection to the output of the corresponding majority node, i.e.,

$$(s_{c,l} = n + i) \to (a_{c,l}^{(j)} = b_i^{(j)} \oplus \bar{p}_{c,l}) \tag{8}$$

for $1 \leq i < l$.

Function semantics: Finally, the function semantics is ensured by the formula

$$b_k^{(j)} = \bar{p} \oplus f(j). \tag{9}$$

Note that k is the largest node index and therefore refers to the root node. Since the majority operation is self-dual, i.e., $\overline{\langle x_1 x_2 x_3 \rangle} = \langle \bar{x}_1 \bar{x}_2 \bar{x}_3 \rangle$, the variable p can be omitted and a minimum solution is still found if one exists.

Symmetry breaking: All the above constraints ensure a correct result in case of a satisfying assignment. In order to reduce the search space, we exploit associativity of the majority operation and add the following symmetry breaking formula to enforce a unique order of the operands:

$$(s_{1,l} < s_{2,l}) \wedge (s_{2,l} < s_{3,l}) \tag{10}$$

Note that there cannot be two edges of a majority node pointing to the same child node in an irreducible MIG, as $\langle aab \rangle = a$.

The connections between the constraints and the resulting MIG for the exact synthesis problem are summarized in the following theorem which also illustrates how to derive the MIG from a satisfying assignment to the variables.

Theorem 1: Let $f : \mathbb{B}^n \to \mathbb{B}$ and k be an instance to the exact synthesis problem. If there exists a satisfying solution to the corresponding SMT instance as described in this section, then let

$$M = (\{x_1, \ldots, x_n\} \cup \{g_1, \ldots, g_k\} \cup \{0\}, E, \{y\})$$

be the extracted MIG with:

$$E = \bigcup_{l=1}^{k} \bigcup_{c=1}^{3} (g_l, \text{target}(s_{c,l}), p_{c,l}) \quad \text{and} \quad y = (g_k, p)$$

and

$$\text{target}(s) = \begin{cases} 0 & \text{if } s = 0, \\ x_s & \text{if } 1 \leq s \leq n, \text{ and} \\ n_{s-n} & \text{if } n < s \leq n + k. \end{cases}$$

Then $\Phi(y) = f$.

IV. Functional Hashing

This section describes an MIG-size reduction algorithm based on functional hashing that makes use of precomputed minimum MIGs for functions with up to 4 inputs. Since the size of an MIG is invariant to inversion and permutation of inputs and outputs, it is sufficient to find minimum representations for each representative in all 222 NPN classes (instead of all 65,536 functions). Already for 5 inputs, the enumeration of all NPN classes becomes impractical, which can be circumvented by considering a much smaller subset (see, e.g., [9]).

We implemented a top-down and a bottom-up approach of the algorithm. For the top-down approach, we also implemented a depth-preserving variant. All algorithms can be applied globally to the MIG or locally to each of its fanout-free region.

Input : MIG $M = (V, E, Y)$
Output: Optimized MIG $\hat{M} = (\hat{V}, \hat{E}, \hat{Y})$
1 set $\hat{Y} \leftarrow \{\texttt{opt}^p(v) \mid (v, p) \in Y\}$;
2 **Function** `opt` $(v \in V)$
  Output: An MIG function optimizing v
3 find $L \in \text{cuts}_4(v)$ that leads to best size reduction;
4 **if** *such an L exists* **then**
5 let $M_{\min} = (X_{\min} \cup G_{\min} \cup \{0\}, E_{\min}, \{y\})$ be the minimum representation for (v, L);
6 replace each $x \in X_{\min}$ by $\text{opt}(l)$ for corresponding $l \in L$;
7 **return** y;
8 **else**
9 let $\text{adj}(v) = \{(v_1, p_1), (v_2, p_2), (v_3, p_3)\}$;
10 **return** $\langle \text{opt}^{p_1}(v_1)\, \text{opt}^{p_2}(v_2)\, \text{opt}^{p_3}(v_3) \rangle$;
11 **end**

Algorithm 1: Top-down approach

A. Top-down Approach

The top-down approach is illustrated by means of Algorithm 1. It creates an optimized MIG $\hat{M}$ for a given MIG M. Starting from each output node v, it tries to find a cut L for which a replacement by a minimum MIG $M_{\min}$ results in the largest reduction (line 3). If such a cut can be found, all internal nodes of the cut can be ignored, and the function `opt` recurs on the leaf nodes in L. Note that the steps in line 6–7 must be aware of the permutations and negations of the NPN representation, which is omitted for clarity in the algorithmic description. If no cut can be found that leads to a size reduction, the algorithm recurs using the child nodes of v (lines 9–10).

In order to locally consider in the top-down approach, a simple heuristic has been implemented that discards cuts in line 3 for which the minimum MIG is locally increasing the depth. Notice that this approach may increase the depth even if the depth of the minimum MIG is smaller than the one of the cut. This is the case if an individual path is enlarged by the replacement.

B. Bottom-up Approach

The bottom-up approach is implemented using dynamic programming and is illustrated by means of Algorithm 2. It visits all nodes in topological order from the inputs to the outputs (line 1). For each node, it stores all replacements of the node's cuts with minimum representations that reduce the size and preserve the depth or only allow slight increases. Such replacements are called candidates and are stored in cand. Each entry in cand contains the candidate function, its size, and its depth. Each terminal node contains one candidate (line 3, where $\hat{v}$ refers to v's corresponding terminal in $\hat{M}$). When replacing a cut by its minimum MIG for each leaf of the minimum MIG one needs to evaluate all computed candidates for the respective nodes (line 7). This may lead to a tremendous number of candidates which may have a dramatic effect on the run-time. To reduce the run-time requirements, we only store a predetermined number of best candidates, similar to priority cuts in technology mapping (see [11]). The function `insert` takes care of keeping only the best candidates w.r.t. to the preferred optimization criteria. After having computed candidates for all nodes, for each output the best candidate is taken for the optimized MIG $\hat{M}$ (line 14).

Input : MIG $M = (V = X \cup G \cup \{0\}, E, Y)$
Output: Optimized MIG $\hat{M} = (\hat{V}, \hat{E}, \hat{Y})$
1 **foreach** $v \in \texttt{topsort}(V)$ **do**
2 **if** $v \in X \cup \{0\}$ **then**
3 $\mathsf{cand}[v] \leftarrow \{((\hat{v}, 1), 0, 0)\}$;
4 **else**
5 **foreach** $L = \{l_1, \ldots, l_k\} \in \text{cuts}_4(v)$ **do**
6 let $M_{\min} = (X_{\min} \cup G_{\min} \cup \{0\}, E_{\min}, \{y\})$ be the minimum representation for (v, L);
7 **foreach** *combination* $c_1 = \mathsf{cand}[l_1], \ldots, c_k = \mathsf{cand}[l_k]$ **do**
8 replace each $x \in X_{\min}$ by corresponding function in candidate c_i;
9 `insert` (cand $[v]$, (y, size, depth));
10 **end**
11 **end**
12 **end**
13 **end**
14 set $\hat{Y} \leftarrow \{\mathsf{cand}[v].\texttt{best}^p(v) \mid (v, p) \in Y\}$;

Algorithm 2: Bottom-up approach

C. Fanout-free Regions

When replacing a subgraph with its minimum size MIG one needs to pay attention not to remove internal nodes with fanout to nodes that are outside of the cut. There are two ways to prevent this. The first one is not to include them when enumerating cuts. Another possibility is to partition the MIG into its fanout-free regions first and then apply the functional hashing to each fanout-free region. This not only reduces the run-time of the algorithm but also shows a positive effect on the resulting area of the optimized MIG as shown in the following section.

Fanout in the logic representation typically results from structural hashing, i.e., reusing structurally equivalent subgraphs when creating the MIG. This already leads to initial reduction in size, but may be undone when applying rewriting outside of fanout boundaries. Another possibility to circumvent the problem of internal fanout is to apply DAG-aware rewriting as in [6].

V. Experiments

We have implemented the exact synthesis and the functional hashing approaches in C++.[1] The following sections provide details of the experimental evaluation.

A. Computing Optimal MIGs

Using the exact synthesis method, we computed all optimal MIGs for each representative of the 222 NPN classes with the SMT solver Z3 [14]. Table I lists the results partitioned by the number of majority nodes. Two classes, the constant functions and the one-variable functions, require no majority node. Another two classes, the two-variable AND and OR-like functions as well as the MAJ-like functions require one majority node. All other functions require at least two majority nodes. The representative of the single most difficult NPN class is the symmetric function $S_{0,2}(x_1, x_2, x_3, x_4)$:

$$S_{0,2}(x_1, x_2, x_3, x_4) = \overline{x_1 \oplus x_2 \oplus x_3 \oplus x_4} \wedge \overline{x_1 x_2 x_3 x_4}$$

[1]The code can be downloaded from github.org/msoeken/cirkit, see also lsi.epfl.ch/MIG.

TABLE I. OPTIMAL MIGS FOR ALL 4-VARIABLE NPN CLASSES

Majority nodes	Classes	Functions	Time	Avg. time
0	2	10	0.00	0.00
1	2	80	0.04	0.02
2	5	640	0.14	0.03
3	18	3300	1.21	0.07
4	42	10352	6.32	0.15
5	117	40064	115.19	0.98
6	35	11058	1458.95	41.68
7	1	32	16796.30	16796.30
Σ	222	65536	18378.15	16839.23

TABLE II. COMPLEXITY OF 4-VARIABLE MIGS

$C(f)$	Class.	Func.	$L(f)$	Class.	Func.	$D(f)$	Class.	Func.
0	2	10	0	2	10	0	2	10
1	2	80	1	2	80	1	2	80
2	5	640	2	5	640	2	48	10260
3	18	3300	3	18	3300	3	169	55184
4	42	10352	4	37	9312	4	1	2
5	117	40064	5	84	28680	5	0	0
6	35	11058	6	63	22568	6	0	0
7	1	32	7	7	832	7	0	0
8	0	0	8	2	80	8	0	0
9	0	0	9	2	34	9	0	0

which is NPN-equivalent to:

$$(x_1 \oplus x_2 \oplus x_3 \oplus x_4) \vee x_1x_2x_3x_4.$$

Its MIG representation is illustrated in Fig. 2.

The number of majority nodes obtained by the exact synthesis algorithm for a function f corresponds to the combinational complexity $C(f)$, in this case, restricted to the majority operation and inversion. Other interesting metrics are the length $L(f)$ that counts the number of operators in the smallest expression and $D(f)$ which corresponds to the longest path in an MIG from a root to a non-terminal. The length of the path is measured by means of visited nodes. Similarly to Table I we have counted the number of functions and NPN classes and partitioned them by length and depth. The results are given in Table II. The representative of the single most deep NPN class is the parity function $S_{1,3}(x_1, x_2, x_3, x_4)$

$$S_{1,3}(x_1, x_2, x_3, x_4) = x_1 \oplus x_2 \oplus x_3 \oplus x_4$$

which is NPN-equivalent to $S_{0,2,4}(x_1, x_2, x_3, x_4)$. There is no further function in that class.

B. Theoretical Results

Based on the exact synthesis algorithm and the exhaustive application to all 222 NPN classes for 4-variable functions, we have derived an upper bound on the size of MIGs for Boolean functions with n variables. In the following, let $C_{\langle\rangle}(n)$ denote the number of majority operations of the most expensive Boolean functions of n variables if only the majority operation together with inversion is allowed.

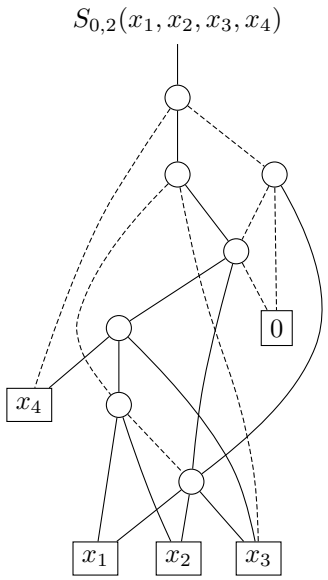

Fig. 2. Optimal MIG for $S_{0,2}(x_1, x_2, x_3, x_4)$

Theorem 2: For $n \geq 4$, we have $C_{\langle\rangle}(n) \leq 10 \cdot (2^{n-4} - 1) + 7$.

Proof: We use induction on n. For the base case $n = 4$, we know from the exhaustive enumeration that the most expensive function requires 7 majority operations. Also, $C_{\langle\rangle}(4) = 7$.

We assume that $C_{\langle\rangle}(n) \leq 10 \cdot (2^{n-4} - 1) + 7$. In the induction step, we make use of Shannon's expansion

$$\begin{aligned} f(x_1, \ldots, x_{n+1}) &= \bar{x}_{n+1} f_{\bar{x}_{n+1}} \vee x_{n+1} f_{x_{n+1}} \\ &= \langle 1 \langle 0 \bar{x}_{n+1} f_{\bar{x}_{n+1}} \rangle \langle 0 x_{n+1} f_{x_{n+1}} \rangle \rangle. \end{aligned}$$

Therefore,

$$\begin{aligned} C_{\langle\rangle}(n+1) &\leq 2C_{\langle\rangle}(n) + 3 \\ &= 2(10 \cdot (2^{n-4} - 1) + 7) + 3 \\ &= 10 \cdot (2^{(n+1)-4} - 1) + 7, \end{aligned}$$

which concludes the proof. ■

C. Functional Hashing

Tables III and IV show the results of the functional hashing algorithm. Each variant is described by an acronym whose letters indicate whether it is *Top-down* (T), *Bottom-up* (B), partitions the MIG first into *Fanout-free regions* (F), and whether it uses the *Depth-preserving heuristic* (D), as described in Section IV. As benchmark sets and comparison baselines we used best results for the arithmetic benchmarks of the EPFL benchmark suite.[2] Most of the best results were obtained using the depth reduction proposed in [3] and [4]. Table III shows the *Size* (S) and *Depth* (D) of the MIG whereas Table IV shows the *Area* (A) and *Depth* (D) of a synthesized circuit after mapping the optimized MIG using ABC [15]. Table III gives *Time* (RT) in seconds. The last row in both tables shows the average improvements.

The functional hashing approach is capable of reducing the size of the MIGs (see Table III) by 8% in average using the BF variant when accepting a modest increase in depth. The depth heuristic has a noticeable effect, particularly when comparing variant T with variant TD. Applying the optimization only to the fanout-free regions leads to significantly better results.

Results obtained from technology mapping are of higher practical relevance. When mapping the obtained MIG representations using ABC, we were able to obtain better implementations in all cases except for the *Adder*. For *Log2*, the top-down approach T was able to find an MIG that leads to both better area and depth in the resulting mapping. It can be

[2] lsi.epfl.ch/benchmarks

TABLE III. FUNCTIONAL HASHING (MIG SIZE AND DEPTH)

Benchmark	I/O	S	D	TF S	TF D	TF RT	T S	T D	T RT	TFD S	TFD D	TFD RT	TD S	TD D	TD RT	BF S	BF D	BF RT
Adder	256/129	2978	**12**	2926	14	0.39	3099	13	0.34	2978	**12**	0.37	2850	13	0.32	**2761**	16	0.91
Divisor	128/128	75666	**636**	70397	694	193.46	75332	703	21.47	75666	**636**	191.34	75558	**636**	18.24	**66133**	720	251.49
Log2	32/32	37582	**181**	36359	186	27.30	38177	207	7.60	37560	**181**	28.67	37653	185	6.91	**34834**	208	41.75
Max	512/130	7202	**27**	6818	29	1.31	6943	29	0.59	7202	**27**	1.32	7195	**27**	0.55	**6539**	28	2.88
Multiplier	128/128	41885	**111**	40781	127	34.92	43116	123	11.32	41480	**111**	32.37	41825	117	10.54	**35869**	121	40.21
Sine	24/25	7890	**91**	**7525**	97	1.65	7993	104	0.96	7887	**91**	1.64	7877	92	0.91	7568	103	5.32
Square-root	128/64	52344	**690**	49142	748	71.05	52590	774	11.88	52320	**690**	66.83	52332	694	10.15	**48494**	793	101.74
Square	64/128	19200	**36**	**18505**	39	6.36	20372	43	2.58	19200	**36**	6.38	19191	**36**	2.34	19046	39	9.41
Average improvement (new/old)				0.96	1.09		1.02	1.12		1.00	1.00		0.99	1.02		0.92	1.14	

TABLE IV. FUNCTIONAL HASHING (AREA AND DEPTH AFTER TECHNOLOGY MAPPING)

Benchmark	I/O	A	D	TF A	TF D	T A	T D	TFD A	TFD D	TD A	TD D	BF A	BF D
Adder	256/129	**414**	**6**	428	**6**	433	**6**	418	**6**	418	**6**	423	**6**
Divisor	128/128	14576	**238**	**10619**	250	14172	**238**	10762	239	14494	246	13486	247
Log2	32/32	9275	55	9290	55	**9126**	**54**	9170	55	9166	55	9272	55
Max	512/130	906	**10**	910	**10**	904	**10**	**888**	**10**	913	**10**	892	**10**
Multiplier	128/128	7180	**29**	7055	**29**	**7051**	**29**	7178	**29**	7141	**29**	7137	**29**
Sine	24/25	1835	**30**	1822	**30**	2112	**30**	1848	**30**	**1801**	**30**	1916	**30**
Square-root	128/64	11745	**254**	11758	255	11881	256	**11449**	256	11579	255	11712	256
Square	64/128	4203	**11**	4169	**11**	4175	**11**	4214	**11**	4174	**11**	**4154**	**11**
Average improvement (new/old)				0.97	1.01	1.02	1.00	0.96	1.00	0.99	1.00	0.99	1.01

seen that—in contrast to Table III—the best mapping results are distributed among the different variants of the optimization algorithm. Hence, it is beneficial to have several variants of the algorithm in order to obtain the best mapping. In summary, for all but one instances of the arithmetic benchmarks, we were able to advance the state-of-the-art, which already are heavily optimized circuits obtained from state-of-the-art logic synthesis algorithms.

In all experiments, we have performed the functional hashing algorithm only once. Running it several times or combining it with other optimization or reshaping algorithms will likely lead to further improvements [3].

VI. CONCLUSIONS

We have presented an algorithm for area reduction in MIGs based on functional hashing. For this purpose, we computed all minimum MIGs for all NPN classes for 4-variable functions using an SMT-based approach. This approach may also be applicable to *individual* larger functions, however, the enumeration of *all* NPN classes for more than 4 variables is impractical. Compared to previously presented approaches, we have implemented several variants of the functional hashing algorithm (top-down and bottom-up, optional partitioning into fanout-free regions, and a heuristic for preserving depth). This enables more flexibility to the design, as we for example observed in the experimental evaluation that for different purposes different variants lead to better results. We were able to improve the best known results of the arithmetic benchmarks in the EPFL suite in 7 out of 8 cases, and an improvement of both area and depth in one case. The current best results are produced by the strongest AIG and MIG optimization scripts from Berkeley and EPFL groups. Consequently, any improvement advances the state-of-the-art in logic optimization.

REFERENCES

[1] R. A. Smith, "Minimal three-variable NOR and NAND logic circuits," *IEEE Transactions on Electronic Computers*, vol. EC-14, no. 1, pp. 79–81, Feb 1965.

[2] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[3] L. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Design Automation Conference*, 2014, pp. 194:1–194:6.

[4] ——, "Boolean logic optimization in majority-inverter graphs," in *Design Automation Conference*, 2015.

[5] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): fast decision procedures," in *Computer Aided Verification*, 2004, pp. 175–188.

[6] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–535.

[7] J. Cortadella, "Timing-driven logic bi-decomposition," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 675–685, 2003.

[8] P. Bjesse and A. Borälv, "DAG-aware circuit compression for formal verification," in *International Conference on Computer-Aided Design*, 2004, pp. 42–49.

[9] N. Li and E. Dubrova, "AIG rewriting using 5-input cuts," in *International Conference on Computer Design*, 2011, pp. 429–430.

[10] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," in *FPGA*, 1998, pp. 35–42.

[11] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton, "Combinational and sequential mapping with priority cuts," in *International Conference on Computer-Aided Design*, 2007, pp. 354–361.

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

[13] L. Benini and G. De Micheli, "A survey of boolean matching techniques for library binding," *ACM Trans. Design Autom. Electr. Syst.*, vol. 2, no. 3, pp. 193–226, 1997.

[14] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[15] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.