Majority-Inverter Graph: A New Paradigm for Logic Optimization

Luca Amarú, *Student Member, IEEE*, Pierre-Emmanuel Gaillardon, *Member, IEEE*, Giovanni De Micheli, *Fellow, IEEE*

***Abstract*— In this paper, we propose a paradigm shift in representing and optimizing logic by using only majority (MAJ) and inversion (INV) functions as basic operations. We represent logic functions by *Majority-Inverter Graph* (MIG): a directed acyclic graph consisting of three-input majority nodes and regular/complemented edges. We optimize MIGs via a new Boolean algebra, based exclusively on majority and inversion operations, that we formally axiomatize in this work. As a complement to MIG algebraic optimization, we develop powerful Boolean methods exploiting global properties of MIGs, such as bit-error masking. MIG algebraic and Boolean methods together attain very high optimization quality. Considering the set of IWLS'05 benchmarks, our MIG optimizer (*MIGhty*) enables a 7% depth reduction in LUT-6 circuits mapped by ABC while also reducing size and power activity, with respect to similar AIG optimization. Focusing on arithmetic intensive benchmarks instead, *MIGhty* enables a 16% depth reduction in LUT-6 circuits mapped by ABC, again with respect to similar AIG optimization. Employed as front-end to a delay-critical 22-nm ASIC flow (logic synthesis + physical design) *MIGhty* reduces the average delay/area/power by 13%/4%/3%, respectively, over 31 academic and industrial benchmarks. We also demonstrate delay/area/power improvements by 10%/10%/5% for a commercial FPGA flow.**

***Index Terms*— Design methods and tools, Optimization, Majority Logic, Boolean Algebra, DAG, Logic Synthesis.**

I. Introduction

NOWADAYS, *Electronic Design Automation* (EDA) tools are challenged by design goals at the frontier of what is achievable in advanced technologies. In this scenario, extending the optimization capabilities of logic synthesis tools is of paramount importance.

In this paper, we propose a paradigm shift in representing and optimizing logic, by using only majority (MAJ) and inversion (INV) as basic operations. We use the terms inversion and complementation interchangeably. We focus on majority functions because they lie at the core of Boolean function classification [1]. Thanks to that, majority inherits the expressive power from many other function classes. Together with inversion, majority can express all Boolean functions. Based on these primitives, we present in this work the *Majority-Inverter Graph* (MIG), a logic representation structure consisting of three-input majority nodes and regular/complemented edges. MIGs include any *AND/OR/Inverter Graphs* (AOIGs), containing also the popular AIGs [2]. To provide native manipulation of MIGs, we introduce a novel Boolean algebra, based exclusively on majority and inversion operations [3]. We define a set of five transformations forming a sound and complete axiomatic system. Using a sequence of these primitive axioms, it is possible to manipulate efficiently a MIG and reach all points in the representation space. We apply MIG algebra axioms locally, to design fast and efficient MIG algebraic optimization methods. We also exploit global properties of MIGs to design slower but very effective MIG Boolean optimization methods [4]. Specifically, we take advantage of the error masking property of majority operators. By selectively inserting logic errors in a MIG, successively masked by majority nodes, we enable strong simplifications in logic networks. MIG algebraic and Boolean methods together attain very high optimization quality. For example when targeting depth reduction, our MIG optimizer, *MIGhty*, transforms a ripple carry structure into a carry look-ahead like one. Considering the set of IWLS'05 benchmarks, *MIGhty* enables a 7% depth reduction in LUT-6 circuits mapped by ABC [2] while also reducing size and power activity, with respect to similar AIG optimization. Focusing on arithmetic intensive benchmarks, *MIGhty* enables a 16% depth reduction in LUT-6 circuits, again with respect to similar AIG optimization. Employed as front-end to a delay-critical 22-nm ASIC flow *MIGhty* reduces the average delay/area/power by 13%/4%/3%, respectively, over academic and industrial benchmarks, as compared to a leading commercial ASIC flow. We demonstrate improvements in delay/area/power metrics by 10%/10%/5% for a commercial 28-nm FPGA flow.

The authors are with the Integrated Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, EPFL, 1015 Lausanne, Switzerland (e-mail: name.surname@epfl.ch).

The remainder of this paper is organized as follows. Section II gives background on logic representation and optimization. Section III presents MIGs with their properties and associated Boolean algebra. Section IV proposes MIG algebraic optimization methods and Section V describes MIG Boolean optimization methods. Section VI shows experimental results for MIG-based optimization and compares them to the state-of-the-art academic tools. Section VI also shows results for MIG-based optimization employed as front-end to commercial ASIC and FPGA design flows. Section VII is a conclusion.

II. Background and Motivation

This section presents first a background on logic representation and optimization for logic synthesis. Then, it introduces the necessary notations and definitions for this work.

A. Logic Representation

The (efficient) way logic functions are represented in EDA tools is key to design efficient hardware. On the one hand, logic representations aim at having the fewest number of primitive elements (literals, sum-of-product terms, nodes in a logic network, etc.) in order to (i) have small memory footprint and (ii) be covered by as few library elements as possible. On the other hand, logic representation forms must be simple enough to manipulate. This may require having a larger number of primitive elements but with simpler manipulation laws. The choice of a computer data-structure is a trade-off between compactness and manipulation easiness.

In the early days of EDA, the standard representation form for logic was the *Sum Of Product* (SOP) form, i.e., a disjunction (OR) of conjuctions (AND) made of literals [5]. This standard was driven by PLA technology whose functionality is naturally modeled by a SOP [6]. Other two-level forms, such as product-of-sums or EX-SOP, have been studied at that time [17]. Two-level logic is compact for small sized functions but, beyond that size, it becomes too large to be efficiently mapped into silicon. Yet, two-level logic has been supported by efficient heuristic and exact optimization algorithms. With the advent of VLSI, the standard representation for logic moved from SOP to *Directed Acyclic Graphs* (DAGs) [7]. In a DAG-based logic representation, nodes correspond to logic functions (gates) and directed edges (wires) connect the nodes. Nodes' functions can be internally represented by SOPs leveraging the proven efficiency of two-level optimization. From a global perspective, general optimization procedures run on the entire DAG. While being potentially very compact, DAGs without bounds on the nodes' functionality are not easy to optimize. This is because this kind of representation demands that optimization techniques deal with all possible types and sizes of functions which is impractical. On top of that, the cumulative memory footprint for each *functionally unbounded* node is potentially very large. Restricting the permissible node function types alleviates this issue. At the extreme case, one can focus on just one type of function per node and add complemented/regular attributes to the edges. Even though in principle, this restriction increases the representation size, in practice it unlocks better (smaller) representations because it supports more effective logic optimization simplifying a DAG. A notable example of DAG where all the nodes realize the same function is *Binary Decision Diagrams* (BDDs) [11]. In BDDs, nodes act as 2:1 multiplexers. With additional restriction on the ordering of input variables, BDDs are canonical and provide very efficient manipulation procedures. For this reason, BDDs found application in various areas of EDA, such as verification, testing, optimization, automated reasoning, etc [5]. However, the price for such an optimal manipulation efficiency is the BDD size, which is often too large for direct mapping into silicon. Even though BDDs are not usually mapped directly into silicon, they support in various ways logic manipulation tasks in some optimization algorithms [9]. Another DAG where all nodes realize the same function is the *And-Inverter Graph* (AIG) [2], [10] where nodes act as two-input ANDs. AIGs can be optimized through traditional Boolean algebra axioms and derived theorems. Iterated over the whole AIG, local transformations produce very effective results and scale well with the size of the circuits. This means that, overall, AIGs can be made remarkably small through logic optimization. For this reason, AIG is one of the current representation standards for logic synthesis.

B. Logic Optimization

Logic optimization consists of manipulating a logic representation structure in order to minimize some target metric. Usual optimization targets are size (number of nodes/elements), depth (maximum number of levels), interconnections (number of edges/nets), etc.

Logic optimization methods are closely coupled to the data structures they run on. In two-level logic representation (SOP), optimization aims at reducing the number of terms. ESPRESSO is the main optimization tool for SOP [6]. Its algorithms operate on SOP cubes and manipulate the ON-, OFF- and DC-covers iteratively. In its default settings, ESPRESSO uses fast heuristics and does not guarantee to reach the global optimum. However, an exact optimization of two level logic is available (under the name of ESPRESSO-exact) and often run in a reasonable time. The exact two-level optimization is based on Quine-McCluskey algorithm [18]. Moving to DAG logic representation (also called multi-level logic), optimization aims at reducing graph size and depth or other accepted complexity metrics. There, DAG-based logic optimization methods are divided into two groups: Algebraic methods, which are fast and Boolean methods, which are slower but may achieve better results [21]. Traditional algebraic methods assume that DAG nodes are represented in SOP form and treat them as polynomials [7], [19]. Algebraic operations are selectively iterated over all DAG nodes, until no improvement is possible. Basic algebraic operations are extraction, decomposition, factoring, balancing and substitution [20], [21]. Their efficient runtime is enabled by theories of weak-division and kernel extraction. In contrast, Boolean methods do not treat the functions as polynomials but handle their true Boolean nature using Boolean identities as well as (global) don't cares (circuit flexibilities) to get a better solution [5], [21], [24]–[26]. Boolean division and substitution techniques trade off runtime for better minimization quality. Functional decomposition is another Boolean method which aims at representing the original function by means of simpler component functions. The first attempts at functional decomposition [27]–[29] make use of decomposition charts to find the best component functions. Since the decomposition charts grow exponentially with the number of variables these techniques are only applicable to small functions. A different, and more scalable, approach to functional decomposition is based on the BDD data structure. A particular class of BDD nodes, called dominator nodes, highlights advantageous functional decomposition points [9]. BDD decomposition can be applied recursively and is capable of exploiting optimization opportunities not visible by algebraic counterparts [9], [22], [23]. Recently, disjoint support decomposition has also been considered to optimize locally small functions and speedup logic manipulation [30], [31]. It is worth mentioning that the main difficulty in developing Boolean algorithms is due to the unrestricted space of choices. This makes more difficult to take good decisions during functional decomposition.

Advanced DAG optimization methodologies, and associated tools, use both algebraic and Boolean methods. When DAG nodes are restricted to just one function type the optimization procedure can be made much more effective. This is because logic transformations are designed specifically to target the functionality of the chosen node. For example, in AIGs, logic transformations such as balancing, refactoring, and general rewriting are very effective. For example, balancing is based on the associativity axiom from traditional Boolean algebra [12], [13]. Refactoring operates on an AIG subgraph which is first collapsed into SOP and then factored out [19]. General rewriting conceptually includes balancing and refactoring. Its purpose is to replace AIG subgraphs with equivalent pre-computed AIG implementations that improve the number of nodes and levels [12]. By applying local, but powerful, transformations many times during AIG optimization it is possible to obtain very good result quality. The restriction to AIGs makes it easier to assess the intermediate quality and to develop the algorithms, but in general is more prone to local minimum. Nevertheless, Boolean methods can still complement AIG optimization to attain higher quality of results [2], [24].

In this work, we present a new representation form, based on majority and inversion, with its native Boolean algebra. We show algebraic and Boolean optimization techniques for this data structure unlocking new points in the design space.

Note that early attempts to majority logic have already been reported in the 60's [14]–[16], but, due to their inherent complexity, failed to gain momentum later on in automated synthesis. We address, in this paper, the unique opportunity led by majority logic in a contemporary synthesis flow.

C. Notations and Definitions

We provide hereafter notations and definitions on Boolean algebra and logic networks.

1) Boolean Algebra: In the binary Boolean domain, the symbol $\mathbb{B}$ indicates the set of binary values $\{0, 1\}$, the symbols $\wedge$ and $\vee$ represent the conjunction (AND) and disjunction (OR) operators, the symbol $'$ represents the complementation (INV) operator and 0/1 are the false/true logic values. Alternative symbols for $\wedge$ and $\vee$ are $\cdot$ and $+$, respectively. The standard Boolean algebra (originally axiomatized by Huntington [32]) is a non-empty set $(\mathbb{B}, \wedge, \vee, ', 0, 1)$ subject to *identity, commutativity, distributivity, associativity* and *complement* axioms over $\wedge, \vee$ and $'$ [1]. For the sake of completeness, we report these basic axioms in Eq. 1. Such axioms will be used later on in this work for proving theorems.

This axiomatization for Boolean algebra is sound and complete [33]. Informally, it means that logic arguments, or formulas, proved by axioms in Δ are valid (soundness) and all true logic arguments are provable (completeness). We refer the reader to [33] for a more formal discussion on mathematical logic. In practical EDA applications, only sound and complete axiomatizations are of interest.

Other Boolean algebras exist, with different operators and axiomatizations, such as Robbins algebra, Freges algebra, Nicods algebra, etc. [33]. Boolean algebras are the basis to operate on logic networks.

$$\Delta \begin{cases} \textbf{Identity : } \boldsymbol{\Delta.I} \\ x \vee 0 = x \\ x \wedge 1 = x \\ \textbf{Commutativity : } \boldsymbol{\Delta.C} \\ x \wedge y = y \wedge x \\ x \vee y = y \vee x \\ \textbf{Distributivity : } \boldsymbol{\Delta.D} \\ x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \\ x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \\ \textbf{Associativity : } \boldsymbol{\Delta.A} \\ x \wedge (y \wedge z) = (x \wedge y) \wedge z \\ x \vee (y \vee z) = (x \vee y) \vee z \\ \textbf{Complement : } \boldsymbol{\Delta.Co} \\ x \vee x' = 1 \\ x \wedge x' = 0 \end{cases} \tag{1}$$

2) Logic Network: A logic network is a *Directed Acyclic Graph* (DAG) with nodes corresponding to logic functions and directed edges representing interconnection between the nodes. The direction of the edges follows the natural computation from inputs to outputs. The terms logic network, Boolean network, and logic circuit are used interchangeably in this paper. A logic network is said *irredundant* if no node can be removed without altering the Boolean function that it represents. A logic network is said *homogeneous* if each node represents the same logic function and has a fixed indegree, i.e., the number of incoming edges or fan-in. In a homogeneous logic network, edges can have a regular or complemented attribute. The depth of a node is the length of the longest path from any primary input variable to the node. The depth of a logic network is the largest depth among all the nodes. The size of a logic network is the number of its nodes.

3) Self-Dual Function: A logic function $f(x, y, .., z)$ is said to be *self-dual* if $f = f'(x', y', .., z')$ [1]. By complementation, an equivalent *self-dual* formulation is $f' = f(x', y', .., z')$.

4) Majority Function: The n-input (n being odd) majority function M returns the logic value assumed by more than half of the inputs [1]. For example, the three input majority function $M(x, y, z)$ is represented in terms of $\wedge, \vee$ by $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$. Also $(x \vee y) \wedge (x \vee z) \wedge (y \vee z)$ is a valid representation for $M(x, y, z)$. The majority function is *self-dual* [1].

III. Majority-Inverter Graphs

In this section, we present MIGs and their representation properties. Then, we show a new Boolean algebra natively fitting the MIG data structure. Finally, we discuss the error masking capabilities of MIGs from an optimization standpoint.

A. MIG Logic Representation

Definition 3.1: An MIG is a homogeneous logic network with an indegree equal to 3 and each node representing the majority function. In a MIG, edges are marked by a regular or complemented attribute.

To determine some basic representation properties of MIGs, we compare them to the well-known *AND/OR/Inverter Graphs* (AOIGs) (which include AIGs). In terms of representation expressiveness, the elementary bricks in MIGs are *majority operators* while in AOIGs there are conjunctions (AND) and disjunctions (OR). It is worth noticing that a majority operator $M(x, y, z)$ behaves as the conjunction operator $AND(x, y)$ when $z = 0$ and as the disjunction operator $OR(x, y)$ when $z = 1$. Therefore, majority is actually a generalization of both conjunction and disjunction. Recall that $M(x, y, z) = xy + xz + yz$. This property leads to the following theorem.

Theorem 3.1: MIGs $\supset$ AOIGs.

Proof: In both AOIGs and MIGs, inverters are represented by complemented edge markers. An AOIG node is always a special case of a MIG node, with the third input biased to logic 0 or 1 to realize an AND or OR, respectively. On the other hand, a MIG node is never a special case of an AOIG node, because the functionality of the three input majority cannot be realized by a unique AND or OR. ■

As a consequence of the previous theorem, MIGs are at least as good as AOIGs but potentially much better, in terms of representation compactness. Indeed, in the worst case, one can replace node-wise AND/ORs by majorities with the third input biased to a constant (0/1). However, even a more compact MIG representation can be obtained by fully exploiting its node functionality rather than fixing one input to a logic constant.

Fig. 1 depicts a MIG representation example for $f = x_3 \cdot (x_2 + (x_1' + x_0)')$. The starting point is a traditional AOIG. Such AOIG has 3 nodes and 3 levels of depth, which is the best representation possible using just AND/ORs. The first MIG is obtained by a one-to-one replacement of AOIG nodes by MIG nodes. As shown by Fig. 1, a better MIG representation is possible by taking advantage of the majority function. This transformation will be detailed in the rest of this paper. In this way, one level of depth is saved with the same node count.

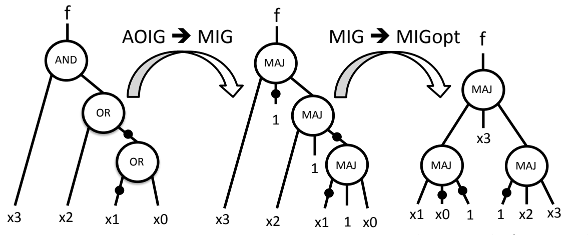

Fig. 1: MIG representation for $f = x_3 \cdot (x_2 + (x_1' + x_0)')$. Complementation is represented by bubbles on the edges.

MIGs inherit from AOIGs some important properties, like universality and AIG inclusion. This is formalized by the following.

Corollary 3.2: MIGs ⊃ AIGs.

Proof: MIGs ⊃ AOIGs ⊃ AIGs $\implies$ MIGs ⊃ AIGs ■

Corollary 3.3: MIG is an universal representation form.

Proof: MIGs ⊃ AOIGs ⊃ AIGs that are universal representation forms [10]. ■

So far, we have shown that MIGs extend the representation capabilities of AOIGs. However, we need a proper set of manipulation tools to handle MIGs and automatically reach compact representations. For this purpose, we introduce hereafter a new Boolean algebra, based on MIG primitives.

B. MIG Boolean Algebra

We present a novel Boolean algebra, defined over the set $(\mathbb{B}, M, ', 0, 1)$, where M is the majority operator of three variables and $'$ is the complementation operator. The following five primitive transformation rules, referred to as Ω, form an *axiomatic system* for $(\mathbb{B}, M, ', 0, 1)$. All variables belong to $\mathbb{B}$.

$$\mathbf{\Omega}\begin{cases}\textbf{Commutativity : } \boldsymbol{\Omega.C} \\ M(x,y,z) = M(y,x,z) = M(z,y,x) \\ \textbf{Majority : } \boldsymbol{\Omega.M} \\ \begin{cases}\text{if}(x=y)\text{: } M(x,x,z) = M(y,y,z) = x = y \\ \text{if}(x=y')\text{: } M(x,x',z) = z\end{cases} \\ \textbf{Associativity : } \boldsymbol{\Omega.A} \\ M(x,u,M(y,u,z)) = M(z,u,M(y,u,x)) \\ \textbf{Distributivity : } \boldsymbol{\Omega.D} \\ M(x,y,M(u,v,z)) = M(M(x,y,u),M(x,y,v),z) \\ \textbf{Inverter Propagation : } \boldsymbol{\Omega.I} \\ M'(x,y,z) = M(x',y',z')\end{cases} \tag{2}$$

Axiom $\Omega.C$ defines a commutativity property. Axiom $\Omega.M$ declares a 2 over 3 decision threshold. Axiom $\Omega.A$ is an associative law extended to ternary operators. Axiom $\Omega.D$ establishes a distributive relation over majority operators. Axiom $\Omega.I$ expresses the interaction between M and complementation operators. It is worth noticing that $\Omega.I$ does not require operation type change like De Morgan laws, as it is well known from self-duality [1].

We prove that $(\mathbb{B}, M, ', 0, 1)$ axiomatized by Ω is an actual Boolean algebra by showing that it induces a complemented distributive lattice [34].

Theorem 3.4: The set $(\mathbb{B}, M, ', 0, 1)$ subject to axioms in Ω is a Boolean algebra.

Proof: The system Ω embed median algebra axioms [35]. In such scheme, $M(0, x, 1) = x$ follows from $\Omega.M$. In [36], it is proved that a median algebra with elements 0 and 1 satisfying $M(0, x, 1) = x$ is a distributive lattice. Moreover, in our scenario, complementation is well defined and propagates through the operator M ($\Omega.I$). Combined with the previous property on distributivity, this makes our system a complemented distributive lattice. Every complemented distributive lattice is a Boolean algebra [34]. ■

Note that there are other possible axiomatic systems for $(\mathbb{B}, M, ', 0, 1)$. For example, one can show that in the presence of $\Omega.C$, $\Omega.A$ and $\Omega.M$, the rule in $\Omega.D$ is redundant [37]. In this work, we consider $\Omega.D$ as part of the axiomatic system for the sake of simplicity.

1) Derived Theorems: Several other complex rules, formally called theorems, in $(\mathbb{B}, M, ', 0, 1)$ are derivable from Ω. Among the ones we encountered, three rules derived from Ω are of particular interest to logic optimization. We refer to them as Ψ and are described hereafter. In the following, the symbol $z_{x/y}$ represents a replacement operation, i.e., it replaces x with y in all its appearence in z.

$$\mathbf{\Psi}\begin{cases}\textbf{Relevance – } \boldsymbol{\Psi.R} \\ M(x,y,z) = M(x,y,z_{x/y'}) \\ \textbf{Complementary Associativity – } \boldsymbol{\Psi.C} \\ M(x,u,M(y,u',z)) = M(x,u,M(y,x,z)) \\ \textbf{Substitution – } \boldsymbol{\Psi.S} \\ M(x,y,z) = \\ M(v, M(v', M_{v/u}(x,y,z), u), M(v', M_{v/u'}(x,y,z), u'))\end{cases} \tag{3}$$

The first rule, relevance ($\Psi.R$), replaces reconvergent variables with their neighbors. For example, consider the function $f = M(x, y, M(w, z', M(x, y, z)))$. Variables x and y are reconvergent because they appear in both the bottom and the top majority operators. In this case, relevance ($\Psi.R$) replaces x with y' in the bottom majority as $f = M(x, y, M(w, z', M(y', y, z)))$. This representation can be further reduced to $f = M(x, y, w)$ by using $\Omega.M$.

The second rule, complementary associativity ($\Psi.C$), deals with variables appearing in both polarities. Its rule of replacement is $M(x, u, M(y, u', z)) = M(x, u, M(y, x, z))$ as depicted by Eq. 3.

The third rule, substitution ($\Psi.S$), extends variable replacement to the non-reconvergent case. We refer the reader to Fig. 2 for an example about substitution ($\Psi.S$) applied to a 3-input parity function.

Hereafter, we show how Ψ rules can be derived from Ω.

Theorem 3.5: Ψ rules are derivable by Ω.

Proof: **Relevance** ($\Psi.R$): Let S be the set of all possible input patterns for $M(x, y, z)$. Let $S_{x=y}$ ($S_{x=y'}$) be the subset of S such that $x = y$ ($x = y'$) condition is true. Note that $S_{x=y} \cap S_{x=y'} = \emptyset$ and $S_{x=y} \cup S_{x=y'} = S$. According to $\Omega.M$, variable z in $M(x, y, z)$ is only relevant for $S_{x=y'}$. Thus, it is possible to replace x with y', i.e., (x/y'), in all its appearance in z, preserving the original functionality.

Complementary Associativity ($\Psi.C$):
$M(x,u,M(u',y,z)) = M(M(x,u,u'),M(x,u,y),z)$ ($\Omega.D$)
$M(M(x,u,u'),M(x,u,y),z) = M(x,z,M(x,u,y))$ ($\Omega.M$)
$M(x,z,M(x,u,y)) = M(x,u,M(y,x,z))$ ($\Omega.A$)

Substitution ($\Psi.S$): We set $M(x,y,z) = k$ for brevity.
$k = M(v,v',k) =$ ($\Omega.M$)
$M(M(u,u',v),v',k) =$ ($\Omega.M$)
$M(M(v',k,u),M(v',k,u'),v) =$ ($\Omega.D$)
Then, $M(v',k,u) = M(v',k_{v/u},u)$ ($\Psi.R$)
and $M(v',k,u') = M(v',k_{v/u'},u)$ ($\Psi.R$)
Recalling that $k = M(x,y,z)$, we finally obtain: $M(x,y,z) = M(v,M(v',M_{v/u}(x,y,z),u),M(v',M_{v/u'}(x,y,z),u'))$ ■

2) Soundness and Completness: The set $(\mathbb{B},M,',0,1)$ together with axioms Ω and derivable theorems form our majority logic system. In a computer implementation of our majority logic system, the natural data structure for $(\mathbb{B},M,',0,1)$ is a MIG and the associated manipulation tools are Ω and Ψ transformations. In order to be useful in practical applications, such as EDA, our majority logic system needs to satisfy fundamental mathematical properties such as soundness and completeness [33]. Soundness means that every argument provable by the axioms in the system is valid. This guarantees preserving of correctness. Completeness means that every valid argument has a proof in the system. This guarantees universal logic reachability. We show that our majority Boolean algebra is sound and complete.

Theorem 3.6: The Boolean algebra $(\mathbb{B},M,',0,1)$ axiomatized by Ω is sound and complete.

Proof: We first consider soundness. Here, we need to prove that all axioms in Ω are valid, i.e., preserve the true behavior (correctness) of a system. Rules $\Omega.C$ and $\Omega.M$ are valid because they express basic properties (commutativity and majority decision rule) of the majority operator. Rule $\Omega.I$ is valid because it derives from the self-duality of the majority operator. For rules $\Omega.D$ and $\Omega.A$, a simple way to prove their validity is to build the corresponding truth tables and check that they are actually the same. It is an easy exercise to verify that it is true. We consider now completeness. Here, we need to prove that every valid argument, i.e., $(\mathbb{B},M,',0,1)$-formula, has a proof in the system Ω. By contradiction, suppose that a true $(\mathbb{B},M,',0,1)$-formula, say α, cannot be proven true using Ω rules. Such $(\mathbb{B},M,',0,1)$-formula α can always be reduced by $\Psi.S$ rules into a $(\mathbb{B},\wedge,\vee,',0,1)$-formula. This is because $\Psi.S$ can behave as Shannon's expansion by setting $v=1$ and u to a logic variable. Using Δ (Eq. 1), all $(\mathbb{B},\wedge,\vee,',0,1)$-formulas can be proven, including α. However, every $(\mathbb{B},\wedge,\vee,',0,1)$-formula is also contained by $(\mathbb{B},M,',0,1)$, where $\wedge$ and $\vee$ are emulated by majority operators. Moreover, rules in Ω with one input fixed to 0 and 1 behaves as Δ rules (Eq. 1). This means that also Ω is capable to prove the reduced $(\mathbb{B},M,',0,1)$-formula α, contradicting our assumption. Thus our system is sound and complete. ■

As a corollary of Ω soundness, all rules in Ψ are valid.

Corollary 3.7: Ψ rules are valid in $(\mathbb{B},M,',0,1)$.

Proof: Ψ rules are derivable by Ω as shown in Theorem 3.5. Then, Ω rules are sound in $(\mathbb{B},M,',0,1)$ as shown in Theorem 3.6. Rules derivable from sound axioms are valid in the original domain. ■

As a corollary of Ω completeness, any element of a pair of equivalent $(\mathbb{B},M,',0,1)$-formulas, or MIGs, can be transformed one into the other by a sequence of Ω transformations. From now on, we use MIGs to refer to functions in the $(\mathbb{B},M,',0,1)$ domain. Still, the same arguments are valid for $(\mathbb{B},M,',0,1)$-formulas.

Corollary 3.8: It is possible to transform any MIG α into any other logically equivalent MIG β, by a sequence of transformations in Ω.

Proof: MIGs are defined over the $(\mathbb{B},M,',0,1)$ domain. Following from Theorem 3.6, all valid arguments over $(\mathbb{B},M,',0,1)$ can be proved by a sequence of Ω rules. A valid argument is then $M(1,M(\alpha,\beta',0),M(\alpha',\beta,0)) = 0$ which reads "α is never different from β" according to the initial hypothesis. It follows that the sequence of Ω rules proving such argument is also logically transforming α into β. ■

3) Reachability: To measure the efficiency of a logic system, thus of its Boolean algebra, one can study (i) the ability to perform a desired task and (ii) the number of basic operations required to perform such a task. In the context of this work, the task we care about is logic optimization. For the graph size and graph depth metrics, MIGs can be smaller than AOIGs because of Theorem 3.1. However, the complexity of Ω sequences required to reach those desirable MIGs is not obvious. In this regard, we give an insight about the majority logic system efficiency by comparing the number of Ω rules needed to get an optimized MIGs with the number of Δ rules needed to get an evenly optimized AIGs. This type of efficiency metric is often referred to as reachability, i.e., the ability to reach a desired representation form with the smallest number of steps possible.

Theorem 3.9: For a given optimization goal and an initial AOIG, the number of Ω rules needed to reach this goal with a MIG is smaller, or at most equal, than the number of Δ rules needed to reach the same goal with an AOIG.

Proof: Consider the shortest sequence of Δ rules meeting the optimization goal with an AOIG. On the MIG side, assume to start with the initial AOIG replacing node-wise AND/OR nodes with pre-configured majority nodes. Note that Ω rules with one input fixed to 0/1 behave as Δ rules. So, it is possible to emulate the same shortest sequence of Δ rules in AOIGs with Ω in MIGs. This is just an upper bound on the shortest sequence of Ω rules. Exploiting the full Ω expresiveness and MIG compactness, this sequence can be further shortened. ■

For a deeper theoretical study on majority logic expresiveness, we refer the reader to [38]. In this work, we use the mathematical theory presented so far to define a consistent logic optimization framework. Then, we give experimental evidence on the benefits predicted by the theory. Results in Section VI show indeed a depth reduction, over the state-of-the-art techniques, up to 48× thanks to our majority logic system. More details on the experiments are given in Section VI.

Operating on MIGs via the new Boolean algebra is one natural approach to run logic optimization. Interestingly enough, other approaches are also possible. In the following, we show how MIGs can be optimized exploiting other properties of the majority operator, such as bit-error masking.

C. Inserting Safe Errors in MIG

MIGs are hierarchical majority voting systems. One notable property of majority voting is the ability to correct different types of bit-errors. This feature is inherited by MIGs, where the error masking property can be exploited for logic optimization. The idea is to purposely introduce logic errors that are succesively masked by the voting resilience in MIG nodes. If such errors are advantageous, in terms of logic simplifications, better MIG representations can be generated.

In the immediate following, we briefly review hereafter notations and definitions on logic errors [5], [39]. Then, we present the theoretical grounds for "safe error insertion" in MIGs. We define what type of errors, and at what overhead cost, can be introduced. Note that, in this work, we use the word *erroneous* to highlight the presence of a logic error. Our notation do not relate to testing or other fields.

Definition The logic error in function f is defined as the difference between f and its erroneous version g and is computed as $f \oplus g$.

In principle, a logic error can be determined for any two circuits. In practical cases, a logic error is interpreted as a perturbation A on an original logic circuit f.

Notation A logic circuit f affected by error A is written f^A.

For example, consider the function $f = (a+b)\cdot c$. An error A defined as *"fix variable b to 0 "* (A: $b = 0$) leads here to $f^A = ac$. In general, an error flips k entries in the truth table of the affected function. In the above example, $k = 1$.

To insert safe (permissible) errors in a MIG we consider a node w and we triplicate the sub-trees rooted at it. In each version of w we introduce logic errors heavily simplifying the MIG. Then, we use the three erroneous versions of w as inputs to a top majority node exploiting the error masking property. Unfortunately, a majority node cannot mask all types of errors. This limits our choice of permissible errors. *Orthogonal* errors, defined hereafter, fit our purposes. Informally, two logic errors are *orthogonal* if for any input pattern they cannot happen simultaneously. In the majority voting scenario the orthogonality is important because it guarantees that no two logic errors happen at the same time which would corrupt the original functionality.

Definition Two logic errors A and B on a logic circuit f are said *orthogonal* if $(f^A \oplus f)\cdot(f^B \oplus f) = 0$.

To give an example of *orthogonal* errors consider again the function $f = (a+b)\cdot c$. Here, the two errors A: $a+b = 1$ and B: $c = 0$ are actually *orthogonal*. Indeed, by logic simplification, we get $(c \oplus f)\cdot(0 \oplus f) = (((a+b)c)'c + ((a+b)c)c')\cdot((a+b)c) = ((a+b)c)'c\cdot((a+b)c) = 0$. Instead, the errors A: $a+b = 1$ and B: $c = 1$ are not *orthogonal* for f. This is because the input $(1,1,1)$ triggers both errors.

Now consider back a generic MIG root w. Let A, B and C be three pairwise *orthogonal* errors on w. Being all pairwise *orthogonal*, a top majority node $M(w^A, w^B, w^C)$ is capable to mask A, B and C orthogonal errors restoring the original functionality of w. This is formalized in the following theorem.

Theorem 3.10: Let w be a generic node in a MIG. Let A, B and C be three pairwise *orthogonal* errors on w. Then the following equation holds: $w = M(w^A, w^B, w^C)$

Proof: The equation $w = M(w^A, w^B, w^C)$ is logically equivalent to $w \oplus M(w^A, w^B, w^C) = 0$. The $\oplus$ (XOR) operator propagates into the majority operator as $w \oplus M(w^A, w^B, w^C) = M(w^A \oplus w, w^B \oplus w, w^C \oplus w)$. Recalling that $M(a,b,c) = ab + ac + bc$ we rewrite the previous expression as $(w^A \oplus w)\cdot(w^B \oplus w) + (w^A \oplus w)\cdot(w^C \oplus w) + (w^B \oplus w)\cdot(w^C \oplus w)$. As A, B and C are pairwise *orthogonal*, we have that each term is 0, so $0+0+0=0$. So, $w \oplus M(w^A, w^B, w^C) = 0$. ■

Note that a MIG $w = M(w^A, w^B, w^C)$ can have up to three times the size and one more level of depth as the original w. This means that simplifications enabled by *orthogonal* errors A, B and C must be significant enough to compensate for such overhead. Note also that this approach resembles triple modular redundancy [40] and its approximate variants [41], but operates differently. Here, we exploit the error masking property in majority operators to enable logic simplifications rather than covering potential hardware failures. More details on how to identify advantageous *orthogonal* errors in MIGs will be given in Section V-A together with related Boolean optimization methods.

In the following sections, we present algorithms for algebraic and Boolean optimization of MIGs.

IV. MIG Algebraic Optimization

In this section, we propose algebraic optimization methods for MIGs. They exploit axioms and derived theorems of the novel Boolean algebra. Our algebraic optimization procedures target size, depth and switching activity reduction in MIGs.

A. Size-Oriented MIG Algebraic Optimization

To optimize the size of a MIG, we aim at reducing the number of its nodes. Node reduction can be done, at first instance, by applying the majority rule. In the MIG Boolean algebra domain this corresponds to the evaluation of the majority axiom ($\Omega.M$) from *Left to Right* ($L \to R$), as $M(x,x,z) = x$. A different node elimination opportunity arises from the distributivity axiom ($\Omega.D$), evaluated from *Right to Left* ($R \to L$), as $M(x,y,M(u,v,z)) = M(M(x,y,u),M(x,y,v),z)$. By applying $\Omega.M_{L\to R}$ and $\Omega.D_{R\to L}$ to all MIG nodes, in an arbitrary sequence, we can actually eliminate nodes. By repeating this procedure until no improvement exists, we designed a simple yet powerful procedure to reduce a MIG size. Embedding some intelligence in the graph exploration direction, e.g., the sequence of MIG nodes, immediately improves the optimization effectiveness. Note that the applicability of majority and distributivity depends on the particular MIG structure. Indeed, there may be MIGs where no direct node elimination is evident. This is because (i) the optimal size is reached or (ii) we are stuck in a local minimum. In the latter case, we want to reshape the MIG in order to encode new reduction opportunities. The rationale driving the reshaping process is to locally increase the number of common inputs/variables to MIG nodes. For this purpose, the associativity axioms ($\Omega.A$, $\Psi.C$) allow us to move variables between adjacent levels and the relevance axiom ($\Psi.R$) to exchange reconvergent variables. When a more radical transformation is beneficial, the substitution axiom ($\Psi.S$) replaces pairs of independent variables, temporarily inflating the MIG. Once the reshaping process has created new reduction opportunities, majority ($\Omega.M_{L\to R}$) and distributivity ($\Omega.D_{R\to L}$) are applied again over the MIG to simplify it. The reshaping and elimination processes can be iterated over a user-defined number of cycles,

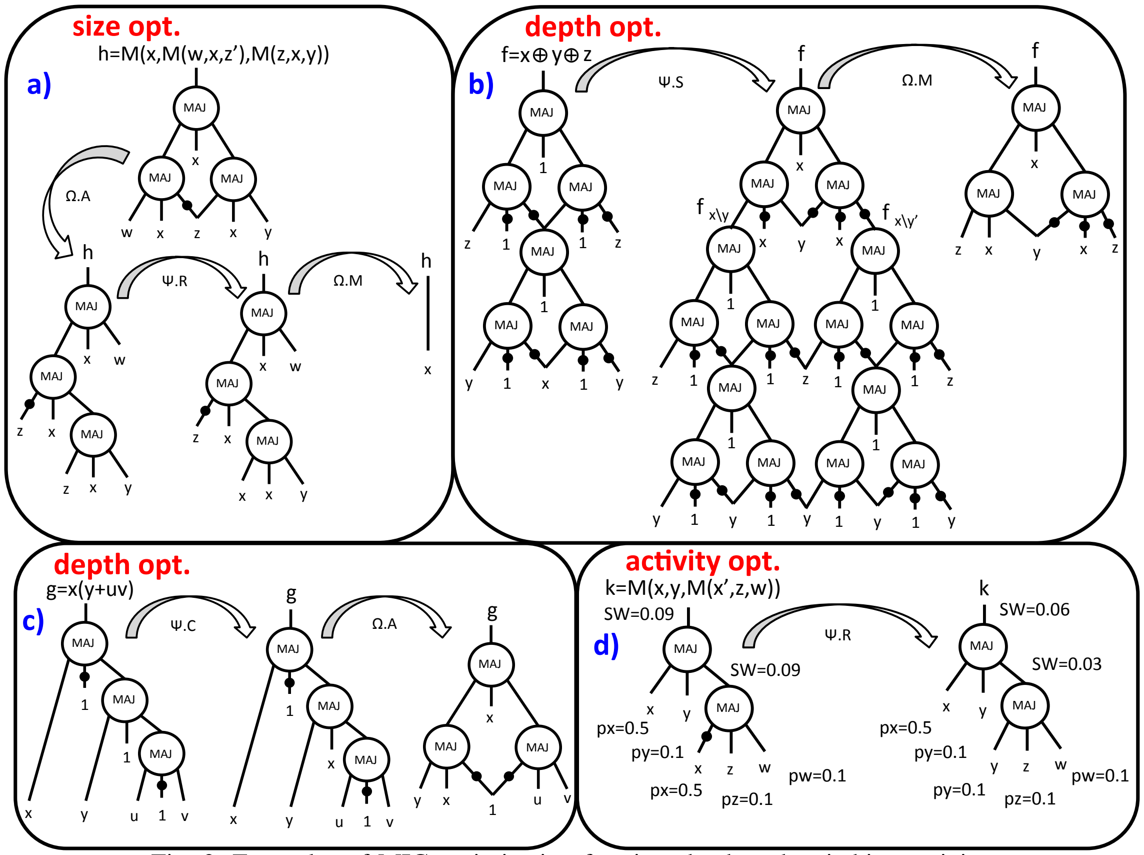

Fig. 2: Examples of MIG optimization for size, depth and switching activity.

called *effort*. Such MIG-size algebraic optimization strategy is summarized in Alg. 1.

```
Algorithm 1 MIG Algebraic Size-Optimization Pseudocode
INPUT: MIG $\alpha$                OUTPUT: Optimized MIG $\alpha$.
  for (cycles=0; cycles<effort; cycles++) do
    $\Omega.M_{L\to R}(\alpha)$;  $\Omega.D_{R\to L}(\alpha)$;
    $\Omega.A(\alpha)$;  $\Psi.C(\alpha)$;       } reshape    } eliminate
    $\Psi.R(\alpha)$;  $\Psi.S(\alpha)$;
    $\Omega.M_{L\to R}(\alpha)$;  $\Omega.D_{R\to L}(\alpha)$;
  end for
```

For the sake of clarity, we comment on the MIG-size algebraic optimization of a simple example, reported in Fig. 2(a). The input MIG is equivalent to the formula $M(x, M(x, z', w), M(x, y, z))$, which has no evident simplification by majority and distributivity axioms. Consequently, the reshape process is invoked to locally increase the number of common inputs. Associativity $\Omega.A$ swaps w and $M(x, y, z)$ in the original formula obtaining $M(x, M(x, z', M(x, y, z)), w)$, when variables x and z are close to the each other. After that, the relevance $\Psi.R$ modifies the inner formula $M(x, z', M(x, y, z))$, exchanging variable z with x and obtaining $M(x, M(x, z', M(x, y, x)), w)$. At this point, the final elimination process is applied, simplifying the reshaped representation as $M(x, M(x, z', M(x, y, x)), w) = M(x, M(x, z', x), w) = M(x, x, w) = x$ by using $\Omega.M_{L\to R}$.

B. Depth-Oriented MIG Algebraic Optimization

To optimize the depth of a MIG, we aim at reducing the length of its critical path. A valid strategy for this purpose is to move late arrival (critical) variables close to the outputs. In order to explain how critical variables can be moved, while preserving the original functionality, consider the general case in which a part of the critical path appears in the form $M(x, y, M(u, v, z))$. If the critical variable is x, or y, no simple move can reduce the depth of $M(x, y, M(u, v, z))$. Whereas, if the critical variable belongs to $M(u, v, z)$, say z, depth reduction is achievable. We focus on the latter case, with order $t_z > t_u \geq t_v > t_x \geq t_y$ for the variables arrival time (depth). Such an order can arise from (i) an unbalanced MIG whose inputs have equal arrival times, or (ii) a balanced MIG whose inputs have different arrival times. In both cases, z is the critical variable arriving later than u, v, x, y, hence the local depth is $t_z + 2$. If we apply the distributivity axiom $\Omega.D$ from left to right ($L \to R$), we obtain $M(x, y, M(u, v, z)) = M(M(x, y, u), M(x, y, v), z)$ where z is pushed one level up, reducing the local depth to $t_z + 1$. Such technique is applicable to a broad range of cases, as all the variables appearing in $M(x, y, M(u, v, z))$ are distinct and independent. However, there is a size penalty of one extra node. In the favorable cases for which associativity axioms ($\Omega.A$, $\Psi.C$) apply, critical variables can be pushed up with no penalty. Furthermore, where majority axiom applies $\Omega.M_{L\to R}$, it is possible to reduce both depth and size. As noted earlier, there exist cases for which moving critical variables cannot improve the overall depth. This is because (i) the optimal depth is reached or (ii) we are stuck in a local minimum. To move away from a local minimum, the reshape process is useful. The reshape and critical variable push-up processes can be iterated over a user-defined number of cycles, called *effort*. Such MIG-depth algebraic optimization strategy is summarized in Alg. 2.

Algorithm 2 MIG Algebraic Depth-Optimization Pseudocode

INPUT: MIG α **OUTPUT:** Optimized MIG α.

```
for (cycles=0; cycles<effort; cycles++) do
    $\Omega.M_{L\rightarrow R}(\alpha)$; $\Omega.D_{L\rightarrow R}(\alpha)$; $\Omega.A(\alpha)$;     }
    $\Omega.A(\alpha)$; $\Psi.C(\alpha)$; }reshape                                                  }push-up
    $\Psi.R(\alpha)$; $\Psi.S(\alpha)$;   }                                                         }
    $\Omega.M_{L\rightarrow R}(\alpha)$; $\Omega.D_{L\rightarrow R}(\alpha)$; $\Omega.A(\alpha)$;     }
end for
```

We comment on the MIG-depth algebraic optimization using two examples depicted by Fig. 2(b-c). The considered functions are $f = x\oplus y\oplus z$ and $g = x(y+u\cdot v)$ with initial MIG representations derived from their optimal AOIGs. In both of them, all inputs have 0 arrival time. No direct push-up operation is advantageous. The reshape process is invoked to move away from local minimum. For $g = x(y+uv)$, complementary associativity $\Psi.C$ enforces variable x to appear in two adjacent levels, while for $f = x \oplus y \oplus z$ substitution $\Psi.S$ replaces x with y, temporarily inflating the MIG. After this reshaping, the push-up procedure is applicable. For $g = x(y + u\cdot v)$, associativity $\Omega.A$ exchanges $1'$ with $M(u, 1', v)$ in the top node, reducing by one level the MIG depth. For $f = x\oplus y\oplus z$, majority $\Omega.M_{L\rightarrow R}$ heavily simplifies the structure and reduces the intermediate MIG depth by four levels. The optimized MIGs have much smaller depth than their optimal AOIGs counterparts. Note that Alg. 2 produces irredundant solutions.

C. Switching Activity-Oriented MIG Algebraic Optimization

To optimize the total switching activity of a MIG, we aim at reducing (i) its size and (ii) the probability for nodes to switch from logic 0 to 1, or vice versa. For the size reduction task, we can run the same MIG-size algebraic optimization described previously. To minimize the switching probability, we want that nodes do not change values often, i.e., the probability of a node to be logic 1 (p_1) is close to 0 or 1 [42]. For this purpose, relevance $\Psi.R$ and substitution $\Psi.S$ can exchange variables with undesirable $p_1 \sim 0.5$ with more favorable variables having $p_1 \sim 1$ or $p_1 \sim 0$. In Fig. 2(d), we show an example where relevance $\Psi.R$ replaces a variable x having $p_1 = 0.5$ with a reconvergent variable y having $p_1 = 0.1$, thus reducing the overall MIG switching activity.

V. MIG Boolean Optimization

In this section, we propose Boolean optimization methods for MIGs. They exploit the *safe error insertion schemes* presented in Section III-C. First, we introduce two techniques to identify advantageous orthogonal errors in MIGs. Second, we present our Boolean optimization technique targeting depth and size reduction in MIGs. Note that also other optimization goals are possible but are not discussed here for brevity.

A. Identifying Advantageous Orthogonal Errors in MIGs

In the following, we present two methods for identifying advantageous triplets of *orthogonal* errors in MIGs.

1) Critical Voters Method: A natural way to discover advantageous triplets of *orthogonal* errors is to analyze a MIG structure. We want to identify critical portions of a MIG to be simplified thanks to these errors. To do so, we focus on nodes[1] that have the highest impact on the final voting decision, i.e., influencing a Boolean function most. We call such nodes *critical voters* of a MIG. Critical voters can also be primary input themselves. To determine the critical voters, we rank MIG nodes based on a *criticality* metric. The *criticality* computation goes as follows. Consider a MIG node m. We label all MIG nodes whose computation depends on m. For all such nodes, we calculate the impact of m by propagating a unit weight value from m outputs up to the root with an attenuation factor of $1/3$ each time a majority node is encountered. We finally sum up all the values obtained and call this result *criticality* of m. Intuitively, MIG nodes with the highest *criticality* are critical voters.

[1]In the context of the critical voters technique we consider also the primary inputs to be a special class of nodes with no fan-in.

For the sake of clarity, we give an example of *criticality* computation in Fig. 3. Node $m5$ has *criticality* of 0, since it is the root and does not propagate to any node. Node $m4$ has *criticality* of 1/3 (a unit weight propagated to $m5$ and attenuated by $1/3$). Node $m3$ has *criticality* of 1/3 ($m4$) + (1/3+1)/3 (direct and $m4$ contribution to $m5$) which sums up to 7/9. Node $m2$ has *criticality* of 1/3 ($m3$) + 4/9 ($m4$) + 7/27 ($m5$) which sums up to 28/27. Node $m1$ has *criticality* 1/3 + *criticality* of $m2$ attenuated by factor 3 which sums up to about 2/3. Among the inputs, only $x1$ has a notable *criticality* being 1/3 ($m3$) + 1/9 ($m4$) + (1/3+1/9+1)/3 ($m5$) which sums up to 25/27. Here the two elements with highest *criticality* are $m2$ and $x1$.

We first determine two critical voters a and b and a set of MIG nodes fed directly by both a and b, say $\{c_1, c_2, ..., c_n\}$. In this context, an advantageous triplet of *orthogonal* errors is: A: $a = b'$, B: $c_1 = a, c_2 = a, ..., c_n = a$ and C: $c_1 = b, c_2 = b, ..., c_n = b$. Consider again the example in Fig. 3. There, the critical voters are $a = m2$ and $b = x1$, while $c_1 = m3$. Thus, the pairwise *orthogonal* errors are $m2 = x1'$ (A), $m3 = x1$ (B) and $m3 = m2$ (C), as shown in Fig. 3. The actual *orthogonality* of A, B and C type of errors is proved in the following theorem.

Theorem 5.1: Let a and b be two critical voters in a MIG. Let $\{c_1, c_2, ..., c_n\}$ be the set of MIG nodes fed by both a and b in the same polarity. Then, the following errors are pairwise *orthogonal*: A: $a = b'$, B: $c_1 = a, c_2 = a, ..., c_n = a$ and C: $c_1 = b, c_2 = b, ..., c_n = b$.

Proof: Starting from a MIG w, we build the three erroneous versions w^A, w^B and w^C as described above. We show that *orthogonality* holds for all 3 pairs. **Pair (w^A, w^B):** We need to show that $(w^A \oplus w)\cdot(w^B \oplus w) = 0$. The element $w^A \oplus w$ implies $a = b$, being the difference between the original and the erroneous one with $a = b'$ ($a \neq b$). The element $w^B \oplus w$ implies $c_i \neq a$ ($c_i = a'$), being the difference between the original and the erroneous one with $c_i = a$. However, if $a = b$ then c_i cannot be a' because $c_i = M(a, b, x) = M(a, a, x) = a \neq a'$ by construction. Thus, the two elements cannot be true at the same time, making $(w^A \oplus w)\cdot(w^B \oplus w) = 0$. **Pair ($w^A, w^C$):** This case is analogous to the previous one. **Pair (w^B, w^C):** We need to show that $(w^B \oplus w)\cdot(w^C \oplus w) = 0$. As we deduced before, the element $w^B \oplus w$ implies $c_i \neq a$ ($c_i = a'$). Similarly, the element $w^C \oplus w$ implies $c_i \neq b$ ($c_i = b'$). By the *transitive property of equality and congruence* in the Boolean domain $c_i \neq a$ and $c_i \neq b$ implies $a = b$. However, if $a = b$, then $c_i = M(a, b, x) = M(a, a, x) = M(b, b, x) = a = b$ which contradicts both $c_i \neq a$ and $c_i \neq b$. Thus, w^B, w^C cannot be

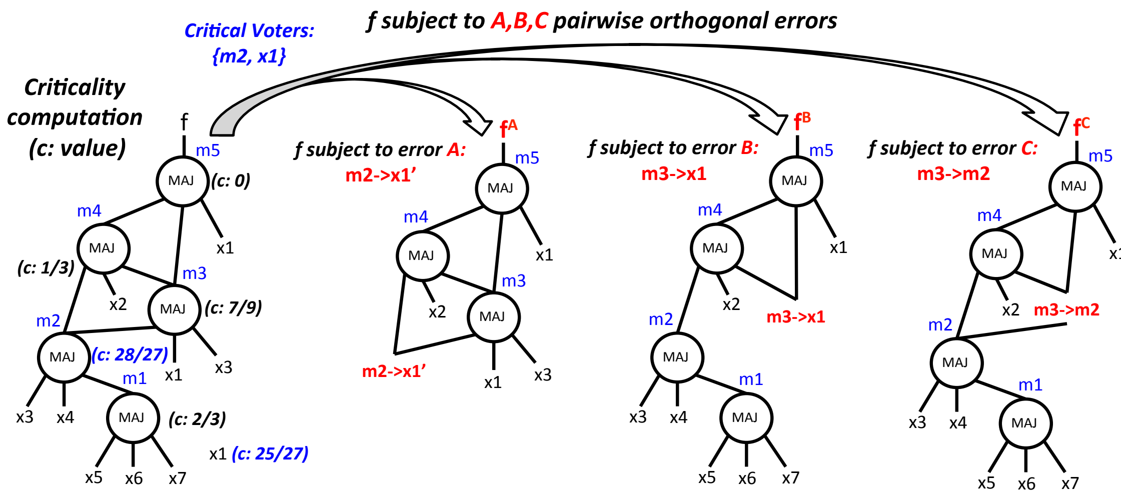
f subject to A,B,C pairwise orthogonal errors
Critical Voters: {m2, x1}
Criticality computation (c: value)
f
m5
MAJ
(c: 0)
m4
x1
MAJ
(c: 1/3)
m3
x2
MAJ
(c: 7/9)
m2
MAJ
(c: 28/27)
x1
x3
m1
x3
x4
MAJ
(c: 2/3)
x1 (c: 25/27)
x5
x6
x7
f subject to error A: m2->x1'
f^A
f subject to error B: m3->x1
f^B
f subject to error C: m3->m2
f^C
m2->x1'
m3->x1
m3->m2

Fig. 3: Example of *criticality* computation and *orthogonal* errors.

true simultaneously, making $(w^B \oplus w) \cdot (w^C \oplus w) = 0$. ■

Even though focusing on critical voters is typically a good strategy for safe error insertion in MIGs, sometimes other techniques can be also convenient. In the following, we present one of these alternative techniques.

2) *Input Partitioning Method:* As a complement to critical voters method, we propose a different way to derive advantageous triplets of *orthogonal* errors in MIGs. In this case, we focus on the inputs rather than looking for internal MIG nodes. In particular, we search for inputs leading to advantageous simplifications when erroneous. Analogously to the *criticality* metric in critical voters, we use here a decision metric, called *dictatorship* [43], to select the most profitable inputs for logic error insertion. The *dictatorship* is the ratio of input patterns over the total (2^n) for which the output assumes the same value as the selected input, in a chosen polarity [43]. For example, in the function $f = (a + b) \cdot c$, the inputs a and b have equal *dictatorship* of 5/8 while input c has an higher *dictatorship* of 7/8. The inputs with the highest *dictatorship* are the ones where we want to insert logic errors. Indeed, they have the largest influence on the circuit functionality and its structure.

Exact computation of the *dictatorship* requires exhaustive simulation of an MIG structure, which is not feasible for practical reasons. Heuristic approaches to estimate *dictatorship* involve partial random simulation and graph techniques [43].

After exact or heuristic computation of the dictatorship, we select a subset of the primary inputs with highest *dictatorship*. Next, for each selected input, we determine a condition that causes an error. We require these errors to be *orthogonal*. Since we operate directly on the primary inputs, we already divide the Boolean space into disjoint subsets that are *orthogonal*. Because we need at least three errors, we need to consider at least three inputs to be made erroneous, say x, y and z. A possible partition is the following: $\{x \neq y,\ x = y = z,\ x = y = z'\}$. The corresponding errors are A: $x = y$ for $\{x \neq y\}$, B: $z = y'$ when $x = y$ for $\{x = y = z\}$ and C: $z = y$ when $x = y$ for $\{x = y = z'\}$. We formally prove A, B and C *orthogonality* hereafter.

Theorem 5.2: Consider the input split $\{x \neq y,\ x = y = z,\ x = y = z'\}$ in a MIG. Three errors A, B and C selectively affecting one subset but not the others are pairwise *orthogonal*.

Proof: To prove the theorem it is sufficient to show that the split $\{x \neq y,\ x = y = z,\ x = y = z'\}$ is actually a partition of the whole Boolean space, i.e., a union of disjoint (non-overlapping) subsets. It is an easy exercise to enumerate all the eight possible $\{x, y, z\}$ input patterns and associate with each of them the corresponding $\{x \neq y,\ x = y = z,\ x = y = z'\}$ subset. By doing so, one can see that no $\{x, y, z\}$ pattern is associated with more than one sub-set, meaning that all subsets are disjoint. Moreover, all together, they form the whole Boolean space. ■

For the sake of clarity, we report an illustrative example on the input partitioning method. The function is $f = M(x, M(x, y', z), M(x', y, z))$. The input split is $\{x \neq y,\ x = y = z,\ x = y = z'\}$ which is affected by errors A, B and C, respectively. The first error A imposes $x = y$ leading to $f^A = M(x, M(y, y', z), M(x', x, z))$ which can be further simplified into $f^A = M(x, z, z) = z$ by $\Omega.M$. The second error B imposes $z = y'$ when $x = y$. This is the case for the bottom level majority operators $M(x, y', z)$ and $M(x', y, z)$ which are transparent when $x = y$. Therefore, error B leads to $f^B = M(x, M(x, y', y'), M(x', y, y'))$ which can be further simplified into $f^B = M(x, y', x') = y'$ by $\Omega.M$. The third error C imposes $z = y$ when $x = y$ holds. Analogously to error B, error C leads to $f^C = M(x, M(x, y', y), M(x', y, y))$ which can be further simplified into $f^C = M(x, x, y) = x$ by $\Omega.M$. A top majority node finally merges the three functions into $f = M(f^A, f^B, f^C) = M(z, y', x)$ which correctly represents the objective function but has 2 fewer nodes and 1 level less than the original representation.

B. Depth-Oriented MIG Boolean Optimization

The most intuitive way to exploit safe error insertion in MIGs is to reduce the number of levels. This is because the initial overhead in $w = M(w^A, w^B, w^C)$, where w is the initial MIG and w^A, w^B, w^C are the three erroneous versions, is just one additional level. This extra level is usually amply recovered during simplification and optimization of MIG erroneous branches. For depth-optimization purposes, the critical voters method introduced in Section III-C enables very good results. The reason is the following. Critical voters appear along the critical path more than once. Thus, the possibility to insert simplifying errors on critical voters directly enables a strong reduction in the maximum number of levels. Some-

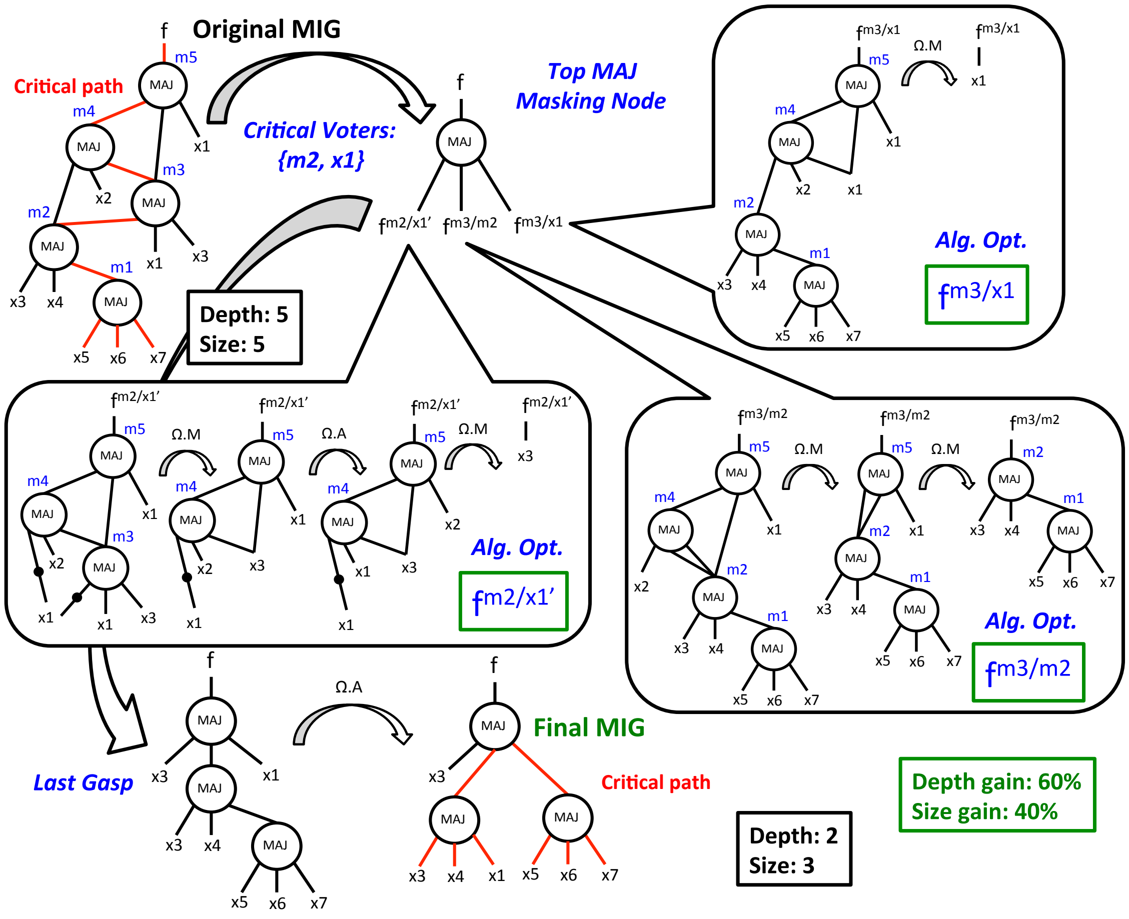

Fig. 4: MIG Boolean depth-optimization example based on critical voters errors insertion. Final depth reduction: 60%.

times, using an actual MIG root for error insertion requires an unpractical size overhead. In these cases, we bound the critical voters search to sub-MIGs partitioned on a depth criticality basis. Once the critical voters and a proper error insertion root have been identified, three erroneous sub-MIG versions are generated as explained in Section III-C. On these sub-MIGs, we want to reduce the logic height. We do so by running algebraic MIG optimization on them (Alg. 2). Note that, in principle, also MIG Boolean methods can be re-used. This would correspond to a recursive Boolean optimization. However, it turned out during experimentation that algebraic optimizations already produce satisfactority results at the local level. Thus, it makes more sense to apply Boolean techniques iteratively on the whole MIG structure rather than recursively on the same logic portion. At the end of the optimization of erroneous branches, the new MIG-roots must be given in input to a top majority voting node. This re-establishes the functional correctness. A *last gasp* of MIG algebraic optimization is applied at this point, to take advantage of the simplification opportunities arosen from the integration of erroneous branches. This Boolean optimization strategy is summarized in Alg. 3.

We comment on the MIG Boolean depth optimization with a simple example, reported in Fig. 4. First, the critical voters are searched and identified, being in this example the input $x1$ and the node $m2$ (from Fig. 3). The proper error insertion root in this small example is the MIG root itself. So, three different versions of the root f are generated with errors $f^{m2/x1'}$, $f^{m3/m2}$ and $f^{m3/x1}$. Each erroneous branch is handled by fast algebraic optimization to reduce its depth. The detailed algebraic optimization steps involved are shown in Fig. 4. The most common operation is $\Omega.M$ that directly simplifies the introduced errors. The optimized erroneous branches are then linked together by a top fault-masking majority node. A last gasp of algebraic optimization on the final MIG structure further optimizes its depth. In summary, our MIG Boolean optimization techniques attains a depth reduction of 60%.

Algorithm 3 MIG Boolean Depth-Optimization Pseudocode

```
INPUT: MIG $\alpha$                OUTPUT: Optimized MIG $\alpha$.
  for (cycles=0; cycles<effort; cycles++) do
    $\{a,b\}$=search_critical_voters($\alpha$);// Critical voters $a,b$ searched
    $c$=size_bounded_root($\alpha,a,b$);// Proper error insertion root
    $x_1^n$=common_parents($\alpha,a,b$);// Nodes fed by both $a$ and $b$
    $c^A$=$c^{b/a'}$;// First erroneous branch
    $c^B$=$c^{x_1^n/a}$;// Second erroneous branch
    $c^C$=$c^{x_1^n/b}$;// Third erroneous branch
    MIG-depth_Alg_Opt($c^A$);// Reduce the erroneous branch height
    MIG-depth_Alg_Opt($c^B$);// Reduce the erroneous branch height
    MIG-depth_Alg_Opt($c^C$);// Reduce the erroneous branch height
    $c$=$M(c^A,c^B,c^C)$;// Link the erroneous branches
    MIG-depth_Alg_Opt($c$); // Last Gasp
    if depth($c$) is not reduced then
      revert to previous MIG state;
    end if
  end for
```

C. Size-Oriented MIG Boolean Optimization

Safe error insertion in MIGs can be used for size reduction. In this case, the branch triplication overhead in $w = M(w^A, w^B, w^C)$ imposes tight simplification requirements. One way to handle this situation is to enforce stricter selection metrics on critical voters. However, the benefits deriving from this approach are limited. A better solution is to change the type of error inserted and use the *input partitioning method*. Indeed, the *input partitioning method* can focus on the most influent inputs of a MIG, and introduces selective simplification on them. The resulting Boolean optimization procedure is similar to Alg. 2 but with depth techniques replaced by size techniques, and critical voter search replaced by input partitioning methods.

VI. Experimental Results

In this section, we test the performance of our MIG optimization techniques on academic and industrial benchmarks. We run logic optimization experiments (comparing logic networks) and complete design experiments (consisting of logic synthesis and physical design) on commercial ASIC and FPGA flows. Finally, we give our vision on nanotechnology design via MIGs.

A. Methodology

We developed a majority-logic manipulation package, called *MIGhty*, consisting of about 8k lines of C code. It embeds various optimization commands based on the theory presented so far. In this work, we use a particular *MIGhty* optimization strategy targeting strong depth reduction interleaved with size recovery phases. The top-level optimization script is depicted by Alg. 4. This technique starts by reducing the depth by

Algorithm 4 Top-Level MIG-optimization Script

```
INPUT: MIG $\alpha$.                OUTPUT: Optimized MIG $\alpha$.
  MIG-depth_Alg_Opt($\alpha$);// small size overhead
  MIG-reshaping($\alpha$);// reshuffling
  MIG-size_Alg_Opt($\alpha$);// no depth overhead
  MIG-depth_Bool_Opt($\alpha$);// pronounced size overhead
  MIG-reshaping($\alpha$);// reshuffling
  MIG-depth_Alg_Opt($\alpha$);// small size overhead
  MIG-size_Bool_Opt($\alpha$);// small depth overhead
  MIG-size_Alg_Opt($\alpha$);// no depth overhead
  MIG-reshaping($\alpha$);// reshuffling
  MIG-depth_Alg_Opt($\alpha$);// small size overhead
  MIG-size_Alg_Opt($\alpha$);// no depth overhead
```

algebraic methods implying a small size overhead. After a fast reshaping step, it decreases the size of the MIG by level-bounded size reduction. At this point, Boolean MIG depth optimization is invoked to significantly reduce the number of levels at the price of a temporary MIG size inflation. Further level reduction opportunities are exploited in an algebraic depth reduction step. Then, size recovery is achieved by Boolean intertwined with algebraic size reduction. A small depth overhead is possible in this phase due to the size reduction. Finally, a last gasp of algebraic depth optimization further compacts the MIG followed by level-bounded algebraic size reduction. All optimization steps have a runtime complexity linear w.r.t. the MIG size, i.e., are imposed to consider each node at least once.

The script in Alg. 4 is a composite optimization strategy, similarly to the class of *resyn* scripts in ABC [2].

MIGhty reads files in Verilog or AIGER format and writes back a Verilog description of the optimized MIG. In order to simplify successive mapping steps, *MIGhty* reduces majority functions into AND/ORs if no size/depth overhead is implied. Thus, only the essential majority functions are written. Also, the number of inversions is minimized by $\Omega.I$ before writing.

We consider IWLS'05 Open Cores benchmarks and larger arithmetic HDL benchmarks. As a case study, we also consider various adder circuits. All the Verilog files deriving from our experiments can be downloaded at [44], for the sake of reproducibility. In all benchmarks, we assumed the input signals to be available at time 0. In total, we optimized about half a million equivalent gates over 31 benchmarks.

For the pure logic optimization experiments, we use as reference tool the ABC academic synthesizer [2], with the delay oriented script $if{-}g; iresyn$. The initial $if{-}g$ optimization strongly reduces the AIG depth by using SOP-balancing [51]. The latter $iresyn$ optimization performs fast rewriting passes on the AIG, reducing mostly the number of nodes but potentially also the number of levels.

We chose the AIG script $if{-}g; iresyn$ because its optimization rationale is close to our MIG optimization strategy and the respective runtimes are comparable. Note that ABC offers many other optimization scripts. Some of them may give better results under determinate conditions (benchmark type, size etc.). As the purpose of this work is primarily to assess the potential of MIG optimization w.r.t. to analogous AIG optimization, we neglect considerations and comparisons related to other ABC commands.

While comparing size and depth of MIGs *vs.* AIGs already gives some good intuition on a data structure and optimization effectiveness, we aim at providing results on even grounds. For this reason, we map both AIG-optimized and MIG-optimized circuits onto LUT6. We perform LUT mapping using the established ABC script $dch{-}f; if\ -m\ -K\ 6$.

For the complete design experiments, we consider a 22-nm (28-nm) commercial ASIC (FPGA) flow suite. The commercial flow consists of a logic synthesis step followed by place & route. In this case, we use the MIG-optimized Verilog file as input to the commercial tools in place of the original Verilog file. In other words, the *MIGhty* package operates as a front-end to the flows. Indeed, the efficiency of MIG-optimization helps the commercial tool to design better circuits. With the final circuit speed being our main design goal, we use an *ultra-high delay effort script* in the commercial tools.

TABLE I: Adder Optimization Results

Type	Bit	Orig. AIG		Map. AIG		Opt. MIG		Map. MIG	
Operands	Width	size	depth	lut6	depth	size	depth	lut6	depth
2-op	32	352	96	65	13	610	12	150	4
2-op	64	704	192	132	26	1159	11	272	5
2-op	128	1408	384	267	52	14672	19	3684	7
2-op	256	2816	768	544	103	7650	16	1870	7
3-op	32	760	68	127	14	1938	16	349	8
4-op	64	1336	136	391	27	2212	18	524	7

B. Optimization Case Study: Adders

We first test the MIG optimization capabilities for adders, that are known hard-to-optimize circuits [52]. Results for more general benchmarks are given in the next subsection. Table I shows the adder results. Our optimized MIG adders have 4 to

TABLE II: MIG Logic Optimization and LUT-6 Mapping Results

		MIGhty					ABC				
Benchmark	I/O	Opt. MIG		Map. MIG			Opt. AIG		Map. AIG		
Open Cores IWLS'05		Size	Depth	LUT6	Depth	Runtime (s)	Size	Depth	LUT6	Depth	Runtime (s)
DSP	4223/3953	40517	34	11077	11	7.98	39958	41	11309	12	5.39
ac97_ctrl	2255/2250	10745	8	2917	3	6.52	10497	9	2914	3	8.98
aes_core	789/668	20947	18	3902	4	11.78	20632	19	3754	5	8.22
des_area	368/72	4186	22	735	6	1.04	5043	24	1012	7	2.11
des_perf	9042/9038	67194	15	12796	3	34.22	75561	15	12814	3	25.43
ethernet	10672/10696	57959	15	18108	6	23.69	56882	22	18267	6	36.54
i2c	147/142	971	8	270	3	0.11	1009	10	268	4	0.05
mem_ctrl	1198/1225	7143	19	2333	7	0.38	9351	22	2582	7	0.33
pci_bridge32	3519/3528	18063	16	5294	6	3.28	16812	18	5424	7	2.22
pci_spoci_ctrl	85/76	932	11	276	4	0.04	994	13	287	4	0.02
sasc	133/132	621	6	152	2	0.11	657	7	152	2	0.03
simple_spi	148/147	837	8	206	3	0.05	770	10	211	3	0.01
spi	274/276	3337	19	812	6	3.71	3430	24	854	7	2.28
ss_pcm	106/98	397	6	104	2	0.01	381	6	104	2	0.01
systemcaes	930/819	9547	25	1845	7	5.26	11014	31	2060	8	4.79
systemcdes	314/258	2453	19	515	5	2.21	2495	21	623	5	1.05
tv80	373/404	7397	30	1980	11	6.43	7838	35	2036	11	2.97
usb_funct	1860/1846	12995	19	3333	5	13.45	13914	20	3394	5	9.04
usb_phy	113/111	372	7	136	2	0.11	380	7	136	2	0.05
IWLS'05 total		266613	**305**	66791	**96**	120.38	277618	**354**	68201	**103**	109.52
Arithmetic HDL		Size	Depth	LUT6	Depth	Runtime (s)	Size	Depth	LUT6	Depth	Runtime (s)
MUL32	64/64	9096	36	1852	10	2.90	8903	40	1993	11	1.90
sqrt32	32/16	2171	164	544	54	1.02	1353	292	236	55	1.22
diffeq1	354/289	17281	219	4685	45	56.32	21980	235	4939	45	16.88
div16	32/32	4374	102	818	37	4.67	5111	132	806	38	2.44
hamming	200/7	2071	61	517	14	2.01	2607	73	590	17	2.54
MAC32	96/65	9326	41	2095	11	4.30	9099	54	2044	12	7.76
metric_comp	279/193	18493	77	6202	29	16.21	21112	95	6796	31	9.51
revx	20/25	7516	143	1937	40	10.70	7516	162	2176	42	12.02
mul64	128/128	25773	109	6557	31	13.84	26024	186	6751	43	10.09
max	512/130	4210	29	1023	12	1.67	2964	113	818	20	2.23
square	64/127	17550	40	4393	13	18.66	17066	168	4278	35	12.24
log2	32/32	31326	201	8809	59	23.32	30701	272	8223	73	15.54
Arithmetic total		149727	**1222**	39432	**355**	155.62	154436	**1822**	39650	**422**	94.37

48× smaller depth than the original AIGs. In all cases, the optimized MIG structure resembles a carry-look ahead design which is known to be the most depth-efficient for adders. Considering LUT mapped results, MIG-optimization enables significantly less deep circuits, having 1.75 to 14× smaller depth than LUT6 circuits mapped from the original AIGs.

C. General Optimization Results

Table II shows general results for *MIGhty* logic optimization and LUT-6 mapping. For the IWLS'05 and HDL arithmetic benchmarks, we see a total improvement in all size, depth and switching activity metrics, w.r.t. to AIG optimized by ABC. The switching activity is computed by the ABC command *ps -p*. The same improvement trend holds also for LUT mapped circuits. Since logic depth was our main optimization target, we notice there the largest reduction.

Considering the IWLS'05 benchmarks, that are large but not deep, *MIGhty* enables about 14% depth reduction. At the LUT-level, we see about 7% depth reduction. At the same time, the size and switching activity are reduced by about 4% and 2%, respectively. At the LUT-level, size and switching activity are reduced by about 2% and 1%, respectively.

Focusing on the arithmetic HDL benchmarks, we see a better depth reduction. Here, *MIGhty* enables about 33% depth reduction. At the LUT-level, it enables about 16% depth reduction. At the same time, *MIGhty* reduces size and switching activity by 4% and 0.1%. At the LUT-level, this corresponds to about 1% size reduction and practically the same switching activity.

The switching activity numbers are not reported in Table II for space reasons but can be reproduced using the ABC command *ps -p* on the files downloadable at [44].

Table II confirms that the runtime of our tool is similar with that of $if-g; iresyn$ ABC script.

All MIG output Verilog files passed formal verification tests (ABC cec and Synopsys Formality) with success.

D. ASIC Results

Table III shows the results for ASIC design (synthesis followed by place and route) at a commercial 22-nm technology node[2]. In total, we see that by using *MIGhty* as front-end to the ASIC design flow, we obtained better final circuits, in all relevant metrics including area, delay and power. For the delay, which was our critical design constraint, we observe an improvement of about 13%. This improvement is not as large as the one we saw at the logic optimization level because some of the gain got absorbed by the interconnect overhead during physical design. However, we still see a coherent trend: We got about 4% and 3% reductions in area and power.

E. FPGA Results

Table IV shows the results for FPGA design (synthesis followed by place and route) on a commercial 28-nm technology node[3]. By employing *MIGhty* as front-end to the FPGA design flow, we obtain better final circuits, in LUT count, delay and power metrics. For the delay, that was our critical design constraint, we observe an improvement of about 10%.

[2]Design tools and library names cannot be disclosed due to our license.

TABLE III: MIG 22-nm ASIC Design Results

Benchmark	MIGhty+ASIC flow			ASIC flow		
	μm^2	ns	mW	μm^2	ns	mW
DSP	6958.23	**0.57**	1.82	1841.76	**2.95**	1.82
ac97_ctrl	2045.48	**0.12**	0.55	2070.83	**0.15**	0.56
aes_core	4599.62	**0.29**	1.75	4417.46	**0.29**	1.64
des_area	853.21	**0.31**	0.59	1084.60	**0.36**	0.53
des_perf	14417.90	**0.20**	11.21	15808.09	**0.23**	11.81
ethernet	10835.31	**0.25**	1.61	10631.93	**0.29**	1.59
i2c	210.13	**0.10**	0.04	210.04	**0.11**	0.04
mem_ctrl	1359.41	**0.30**	0.27	1372.58	**0.33**	0.27
pci_b32	3215.69	**0.26**	0.79	3259.76	**0.29**	0.79
pci_spoci	159.34	**0.16**	0.08	177.47	**0.16**	0.09
sasc	125.12	**0.08**	0.02	139.98	**0.10**	0.02
simple_spi	169.60	**0.12**	0.04	178.64	**0.14**	0.04
spi	542.22	**0.39**	0.21	503.41	**0.42**	0.18
ss_pcm	85.33	**0.08**	0.02	89.23	**0.08**	0.02
systemcaes	1328.08	**0.35**	0.65	1427.94	**0.43**	0.66
systemcdes	538.97	**0.31**	0.37	641.30	**0.33**	0.45
tv80	1299.34	**0.43**	0.37	1213.84	**0.50**	0.40
usb_funct	2269.22	**0.25**	0.72	2337.65	**0.26**	0.77
usb_phy	111.15	**0.05**	0.02	115.73	**0.07**	0.02
MUL32	1862.55	**0.55**	1.81	1748.45	**0.56**	1.90
sqrt32	498.65	**2.54**	0.62	504.76	**2.74**	0.62
diffeq1	3460.48	**3.19**	4.33	3713.87	**3.49**	4.68
div16	595.86	**1.64**	0.26	948.66	**2.06**	0.40
hamming	325.65	**0.90**	0.56	348.46	**1.04**	0.58
MAC32	2281.57	**0.58**	1.95	2194.88	**0.60**	1.89
metric_c	4274.04	**1.36**	1.68	4642.09	**1.55**	1.72
revx	1401.04	**2.23**	1.42	1451.11	**2.63**	1.48
mul64	6378.20	**1.43**	7.01	6330.08	**1.82**	6.95
max	628.23	**0.45**	0.33	631.46	**0.56**	0.33
square	4031.05	**0.46**	3.69	3895.13	**0.67**	3.57
log2	6784.70	**3.07**	7.45	7197.50	**3.59**	8.03
Total	83645.37	**23.02**	53.37	86270.09	**26.47**	55.04

TABLE IV: MIG 28-nm FPGA Design Results

Benchmark	MIGhty+FPGA flow			FPGA flow		
	LUT6	ns	W	LUT6	ns	W
DSP*	9599	**8.22**	7.76	9501	**8.54**	7.73
ac97_ctrl*	2417	**4.54**	3.91	2444	**4.67**	3.92
aes_core	4440	**5.54**	1.93	4788	**5.63**	1.94
des_area	955	**15.24**	0.96	1212	**15.73**	0.98
des_perf*	8480	**5.22**	18.56	11350	**5.40**	18.75
etherne*t	14840	**6.26**	23.89	16343	**6.74**	23.84
i2c	274	**10.58**	0.83	264	**10.38**	0.83
mem_ctrl*	1929	**6.74**	2.00	2044	**7.25**	1.99
pci_b32*	4542	**5.76**	7.77	4741	**6.39**	7.78
pci_spoci	260	**9.86**	0.81	290	**9.99**	0.81
sasc	141	**10.02**	0.88	137	**10.04**	0.88
simple_spi	192	**9.91**	0.93	200	**10.23**	0.93
spi	994	**15.72**	1.32	814	**18.57**	1.35
ss_pcm	92	**9.60**	0.78	89	**9.58**	0.78
systemcaes	1445	**6.67**	2.31	1445	**6.96**	2.32
systemcdes	667	**14.93**	1.31	798	**15.90**	1.31
tv80	1892	**16.44**	1.57	1975	**17.47**	1.57
usb_funct*	2988	**6.02**	3.25	2887	**5.79**	3.21
usb_phy	97	**10.00**	0.82	94	**10.06**	0.82
MUL32	1776	**11.05**	0.88	1867	**12.02**	0.89
sqrt32	447	**25.46**	0.68	560	**27.81**	0.70
diffeq1	5134	**22.36**	1.56	6545	**30.89**	1.82
div16	1160	**26.03**	0.72	765	**28.12**	0.70
hamming	519	**16.20**	13.16	657	**17.65**	17.81
MAC32	2220	**12.47**	0.93	2338	**15.83**	0.94
metric_c	5486	**34.57**	1.11	6416	**38.65**	1.13
revx	2010	**26.19**	0.79	2333	**31.04**	0.80
mul64	7109	**22.54**	1.77	6224	**25.07**	1.41
max	952	**20.10**	1.04	754	**22.19**	1.04
square	4327	**17.05**	1.16	3579	**17.56**	1.11
log2	9944	**44.13**	1.42	14166	**51.75**	1.79
Total	97328	**455.41**	106.81	107620	**503.97**	111.88

Also here, P&R absorbs part of the advantage predicted at the logic-level. Regarding LUT number and power, we see improvements of about 10% and 5%, respectively. Some of the values reported (marked by*) are just post synthesis results because the placement and routing on FPGA failed due to excessive number of I/Os.

In summary, MIG synthesis technology enables a consistent advantage over the state-of-the-art commercial design flows. It is worth noticing that we employed MIG optimization just as a front-end to an existing commercial flow. We foresee even better results by integrating MIG optimization inside the synthesis engine of commercial tools.

F. Nanotechnology Design via MIGs

Due to their ultra-scaled dimensions, nanotechnologies often operate on physics principles that are different from those of traditional CMOS. For example, logic switches in some nanotechnologies do not even use electron charge as the state variable [45]. This brings new logic primitives to the attention of logic synthesis. In particular, various promising nanotechnologies realize devices behaving as majority voters. Specific examples include, but are not limited to, spin-wave device [46], quantum-dot cellular automata [47], DNA-based logic [48], ReRAM device [49] and ambipolar FET nanotechnologies [50]. For these nanotechnologies, MIGs are the natural and native circuit abstraction for automated design. MIGs can fully harness the logic advantage over CMOS provided by these new switches, which is often a pivotal asset in the corresponding nanotechnologies. Preliminary experiments already shown superior results for SWD, ambipolar FET and ReRAM nanotechnologies [46], [49], [50]. Based on our studies and results so far, we foresee an even broader impact of MIGs in nanotechnology design.

VII. Conclusions

In this paper, we proposed a paradigm shift in representing and optimizing logic circuits, by using only majority (MAJ) and inversion (INV) as basic operations. We presented the *Majority-Inverter Graphs* (MIGs): a directed acyclic graph consisting of three-input majority nodes and regular/complemented edges. We developed algebraic and Boolean optimization techniques for MIGs and we embedded them into a tool, called *MIGhty*. Over the set of IWLS'05 (arithmetic intensive) benchmarks, *MIGhty* enabled a 7% (16%) depth reduction in LUT-6 circuits mapped by ABC while also reducing size and switching activity, with respect to similar AIG optimization. Employed as front-end to a delay-critical 22-nm ASIC flow, *MIGhty* reduced the average delay/area/power by about 13%/4%/3%, over 31 benchmarks. We also demonstrated improvements in delay/area/power by 10%/10%/5% for a commercial 28-nm FPGA flow.

Acknowledgements

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

References

[1] T. Sasao, *Switching Theory for Logic Synthesis*, Springer, 1999.

[2] ABC synthesis tool - available online at http://www.eecs.berkeley.edu/∼alanmi/abc/.

[3] L. Amarú, *et al.*, *Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization*, Proc. DAC'14.

[4] L. Amarú, *et al.*, *Boolean Logic Optimization in Majority-Inverter Graph*, Proc. DAC'15.

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

[6] R.L. Rudell, A. Sangiovanni-Vincentelli, *Multiple-valued minimization for PLA optimization*, IEEE TCAD, 6(5): 727-750, 1987.

[7] R.K. Brayton, *et al.*, *MIS: A Multiple-Level Logic Optimization System*, IEEE Trans. CAD, 6(6): 1062-1081, 1987.

[8] E. Sentovich, *et al.*, *SIS: A System for Sequential Circuit Synthesis*, ERL, Dept. EECS, Univ. California, Berkeley, UCB/ERL M92/41, 1992.
[9] C. Yang and M. Ciesielski, *BDS: A BDD-Based Logic Optimization System*, IEEE TCAD, 21(7): 866-876, 2002.
[10] R. Brayton, A. Mishchenko, *ABC: An Academic Industrial-Strength Verification Tool*, Proc. CAV, 2010.
[11] R.E. Bryant, *Graph-based algorithms for Boolean function manipulation*, IEEE TCOMP, C-35(8): 677-691, 1986.
[12] A. Mishchenko, S. Chatterjee, R. Brayton, *DAG-aware AIG rewriting a fresh look at combinational logic synthesis*, Proc. DAC 2006.
[13] A. Mishchenko, R. Brayton, *Scalable logic synthesis using a simple circuit structure*, Proc. IWLS 2006.
[14] H.S. Miller, R. O. Winder. *Majority-logic synthesis by geometric methods* IRE Transactions on Electronic Computers, (1962): 89-90.
[15] Y. Tohma, *Decompositions of Logical Functions Using Majority Decision Elements*, IEEE Trans. on Electronic Computers, pp. 698-705, 1964.
[16] F. Miyata, *Realization of arbitrary logical functions using majority elements*, IEEE Transactions on Electronic Computers, (1963): 183-191.
[17] N. Song, *et al.*, *EXORCISM-MV-2: minimization of ESOP expressions for MV input incompletely specified functions*, Proc. on MVL, 1993.
[18] E. J. McCluskey, *Minimization of Boolean Functions*, Bell system technical Journal 35.6 (1956): 1417-1444.
[19] R.K. Brayton, *The Decomposition and Factorization of Boolean Expressions*, Proc. ISCAS'82.
[20] R.K. Brayton, *et al.*, *MIS: A multiple-level logic optimization system*, IEEE TCAD 6.6 (1987): 1062-1081.
[21] R.K. Brayton, *Multilevel logic synthesis*, Proc. IEEE78.2(1990):264-300.
[22] N. Vemuri, *et al.*, *BDD-based logic synthesis for LUT-based FPGAs*, ACM TODAES 7.4 (2002): 501-525.
[23] L. Amaru, P.-E. Gaillardon, G. De Micheli, *BDS-MAJ: A BDD-based logic synthesis tool exploiting majority decomposition*, Proc. DAC, 2013.
[24] A. Mishchenko *at al., Using simulation and satisfiability to compute flexibilities in Boolean networks*, IEEE TCAD 25 (5): 743-755, 2006.
[25] S. C. Chang, *et al.*, *Perturb and Simplify: multilevel Boolean network optimizer*, IEEE TCAD 15.12 (1996): 1494-1504.
[26] S. C. Chang, L. P. Van Ginneken, M. Marek-Sadowska *Circuit optimization by rewiring*, IEEE TCOMP 48.9 (1999): 962-970.
[27] R. Ashenhurst, *The decomposition of switching functions*, In Proceedings of the International Symposium on the Theory of Switching, pages 74116, April 1957.
[28] J. P. Roth and R. M. Karp *Minimization over boolean graphs*, IBM Journal, pages 661664, April 1962.
[29] H. A. Curtis, *A New Approach to the Design of Switching Circuits*, Van Nostrand, Princeton, N.J., 1962.
[30] V. Bertacco and M. Damiani, *Disjunctive decomposition of logic functions*, Proc. ICCAD 97, pp. 78-82.
[31] A. Mishchenko and R. Brayton, *Faster logic manipulation for large designs*, Proc. IWLS'13.
[32] E. V. Huntington, *Sets of Independent Postulates for the Algebra of Logic*, Trans. of the American Math. Society, 5:3 (1904), 288-309.
[33] B. Jonsson, Bjarni, *Boolean algebras with operators. Part I.*, American journal of mathematics (1951): 891-939.
[34] G. Birkhoff, *Lattice Theory*, Amer. Math. Soc., New York, 1967
[35] John R. Isbell, *Median algebra*, Trans. Amer. Math. Soc., 319-362, 1980.
[36] G. Birkhoff, *A ternary operation in distributive lattices*, Bull. of the Amer. Math. Soc., 53 (1): 749752, 1947.
[37] D. Knuth, *The Art of Computer Programming*, Volume 4A, Part 1, New Jersey: Addison-Wesley, 2011
[38] M. Krause, *et al.*, *On the computational power of depth-2 circuits with threshold and modulo gates*, Theor. Comp. Sci. 174.1 (1997): 137-156.
[39] S. Muroga, *et al.*, *The transduction method-design of logic networks based on permissible functions*, IEEE TCOMP, 38.10 (1989): 1404-1424.
[40] R.E. Lyons, W. Vanderkulk., *The use of triple-modular redundancy to improve computer reliability*, IBM Journal of Research and Development 6.2 (1962): 200-209.
[41] A. AC., Gomes, et al. *Methodology for achieving best trade-off of area and fault masking coverage in ATMR*, IEEE LATW, 2014.
[42] M.Pedram *Power minimization in IC design: principles and applications*, ACM TODAES 1.1 (1996): 3-56.
[43] M. Parnas, *et al.*, *Proclaiming dictators and juntas or testing boolean formulae*, Combinatorial Optimization, Springer, 2001. 273-285.
[44] http://lsi.epfl.ch/MIG
[45] K. Bernstein *et al.*, *Device and Architecture Outlook for Beyond CMOS Switches*, Proceedings of the IEEE, 98(12): 2169-2184, 2010.
[46] O. Zografos *et al.*, *Majority Logic Synthesis for Spin Wave Technology*, Proc. DSD'14.
[47] P. D. Tougaw, C. S. Lent, *Logical devices implemented using quantum cellular automata*, J. Applied Physics, 75(3): 1811-1817, 1994.
[48] W. Li, *et al.*, *Three-Input Majority Logic Gate and Multiple Input Logic Circuit Based on DNA Strand Displacement*, Nano letters 13.6 (2013).
[49] P.-E. Gaillardon *et al.*, *Computing Secrets on a Resistive Memory Array*, Proc. DAC'15.
[50] L. Amaru *et al.*, *Efficient arithmetic logic gates using double-gate silicon nanowire FETs* Proc. NEWCAS 2013.
[51] A. Mishchenko, et al. *Delay optimization using SOP balancing*, Proc. ICCAD, 2011.
[52] J. P. Fishburn, *A depth-decreasing heuristic for combinational logic*, Proc. DAC 1990.

Luca Amarú (S'13) Luca Amarú received the B.S. degree in Electronic Engineering from Politecnico di Torino in 2009. In 2011, he received the joint M.S. degree in Electronic Engineering from Politecnico di Torino and Politecnico di Milano. Since 2011, he has been working toward his Ph.D. degree in Computer Science at EPFL, Lausanne, Switzerland. In 2014, he was a visiting researcher at Stanford University, Palo Alto, CA, USA. He has been serving as TPC member for DSD'14-15 conferences and is reviewer for several IEEE journals.

His research interests include design automation, logic in computer science and beyond CMOS technologies.

Mr. Amarú received the Best Presentation Award at FETCH 2013 conference and a Best Paper Award Nomination at ASP-DAC 2013 conference.

Pierre-Emmanuel Gaillardon (S'10-M'11) works for EPFL, Lausanne, Switzerland, as a research associate at the Laboratory of Integrated Systems (LSI). He holds an Electrical Engineer degree from CPE-Lyon, France (2008), a M.Sc. degree in Electrical Engineering from INSA Lyon, France (2008) and a Ph.D. degree in Electrical Engineering from CEA-LETI, Grenoble, France and the University of Lyon, France (2011). Starting January 2016, he will assume an assistant professor position within the Electrical and Computer Engineering (ECE) department at The University of Utah, Salt Lake City, UT, USA. Previously, he was research assistant at CEA-LETI, Grenoble, France and visiting research associate at Stanford University, Palo Alto, CA, USA. Dr. Gaillardon is recipient of the C-Innov 2011 best thesis award and the Nanoarch 2012 best paper award. He is an Associate Editor of the IEEE Transactions on Nanotechnology. He has been serving as TPC member for many conferences and is reviewer for several journals and funding agencies. His research activities and interests are currently focused on the development of reconfigurable logic architectures and circuits exploiting emerging technologies and novel EDA techniques.

Giovanni De Micheli (M'83-SM'89-F'94) received the Nuclear Engineer degree from Politecnico di Milano, Italy, in 1979, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley, CA, USA, in 1980 and 1983, respectively. He is Professor and Director of the Institute of Electrical Engineering and of the Integrated Systems Centre at EPF Lausanne, Switzerland, and is program leader of the Nano-Tera.ch program. Previously, he was Professor of Electrical Engineering at Stanford University, Stanford, CA, USA. His research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies, networks on chips and 3D integration. He is also interested in heterogeneous platform design including electrical components and biosensors, as well as in data processing of biomedical information. He is author of Synthesis and Optimization of Digital Circuits (McGraw-Hill, 1994), and co-author and/or co-editor of eight other books and of over 600 technical articles. His citation h-index is 85 according to Google Scholar.

Prof. De Micheli is a Fellow of ACM and a member of the Academia Europaea. He is a member of the Scientific Advisory Board of IMEC, CfAED and STMicroelectronics. He was the recipient of the 2012 IEEE/CAS Mac Van Valkenburg award for contributions to theory, practice and experimentation in design methods and tools, and of the 2003 IEEE Emanuel Piore Award for contributions to computer-aided synthesis of digital systems. He also received the Golden Jubilee Medal for outstanding contributions to the IEEE CAS Society in 2000, the D. Pederson Award for the best paper in the IEEE TRANSACTIONS ON CAD/ICAS in 1987, and several Best Paper Awards including DAC (1983 and 1993), DATE (2005), and Nanoarch (2010 and 2012). He has served IEEE in several capacities, including Division 1 Director (2008-2009), co-founder and President Elect of the IEEE Council on EDA (2005-2007), President of the IEEE CAS Society (2003), and Editor-in-Chief of the IEEE TRANSACTIONS ON CAD/ICAS (1997-2001). He has been Chair of several conferences, including Memocode (2014), DATE (2010), pHealth (2006), VLSI SOC (2006), DAC (2000), and ICCD (1989).