View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

LUT Mapping and Optimization for Majority-Inverter Graphs

Winston Haaswijk*, Mathias Soeken*, Luca Amarù†, Pierre-Emmanuel Gaillardon‡, Giovanni De Micheli*
*Integrated Systems Laboratory, EPFL, Lausanne, VD, Switzerland
†Design Group, Synopsys Inc., Mountain View, CA, USA
‡Laboratory for NanoIntegrated Systems, The University of Utah, Salt Lake City, UT, USA

***Abstract*—A *Majority-Inverter Graph* (MIG) is a directed acyclic graph in which every vertex represents a three-input majority operation and edges may be complemented to indicate operand inversion. MIGs have algebraic and Boolean properties that enable efficient logic optimization. They have been shown to obtain superior synthesis results as compared to state-of-the-art *And-Inverter Graph* (AIG) based algorithms. In this paper, we extend MIGs to *Functionally Reduced* MIGs (FRMIGs), analogous to the extension of AIGs to *Functionally Reduced* AIGs (FRAIGs). This enables the use of MIGs in a *lossless synthesis* design flow. We present an FRMIG based technology mapper for *lookup tables* (LUTs). Any MIG may be mapped to a k-LUT network. Using *exact synthesis* we may decompose the k-LUT network back into an equivalent MIG. We show how LUT mapping and exact k-LUT decomposition can be used to create an MIG optimization method. Finally, we present the results of applying our new optimization method and LUT mapper to both logic optimization and technology mapping.**

I. Introduction

For several decades, the performance of digital circuits has been largely dependent on the effectiveness of the tools developed by the logic synthesis community. Advancements in logic representation and optimization as well as technology mapping have enabled continued improvement to the capabilities of modern chips.

Within logic representation and optimization there has been a trend from heterogeneous logic representations towards simpler, homogeneous representations. Traditionally, nodes in multi-level logic networks represented complex Boolean functions on varying numbers of inputs [1]–[3]. While this is a rich representation that allows for powerful optimization techniques, homogeneous networks such as *And-Inverter Graphs* (AIGs) have permitted more efficient implementations. These implementations use less memory and enable better runtimes while maintaining or even improving synthesis results [4], [5]. As such, homogeneous networks appear to be the more scalable approach.

The recently introduced *Majority-Inverter Graphs* (MIGs) are another example of homogeneous networks [6]. A generalization of AIGs, MIGs are directed acyclic graphs consisting of three-input majority nodes with optionally complemented edges. MIGs include AIGs but also have other desirable algebraic and Boolean properties. Algorithms that exploit these properties have been shown to obtain superior results in both logic optimization and technology mapping, especially with respect to depth and delay reduction [6], [7]. Novel work on *exact synthesis* has shown to be another avenue for MIG optimization, enabling MIG size reduction as well as improvements in both depth and area in FPGA technology mapping [8].

Technology mapping is the problem of covering a logic network with a set of primitives. In some literature it is also referred to as *cell-library binding* [3]. Typically, technology mapping occurs after the application of technology independent optimizations, although there are systems that perform both operations simultaneously [9]. The state of the art in technology mapping uses cut enumeration techniques to find suitable covers [10]–[12]. We distinguish between two different kinds of technology mapping. In the former, we are given a library of logic primitives such as *standard cells* or *gate arrays*. The available primitives depend on the particular technology. In the latter, we are mapping to lookup table-based FPGAs. Lookup tables are powerful primitives that can implement any function on k variables. Typically, we refer to lookup tables on k variables as k-LUTs. The k-LUT FPGA mapping problem is simpler than the more general case of cell library binding because of the homogeneous nature of the logic primitives. In the remainder, we focus on k-LUT technology mapping, which is applicable to both FPGA technology mapping and resynthesis of general logic networks. We also refer to this process simply as LUT mapping.

Our goal is to improve techniques for MIG logic optimization and LUT mapping. To achieve this goal we present a number of contributions:

1) We show how MIGs can be extended to the *Functionally Reduced* MIGs (FRMIGs). This enables a *lossless synthesis* flow for MIGs.
2) We present a cut-based LUT mapper for FRMIGs.
3) We present an MIG optimization method that uses the FRMIG mapper and exact k-LUT decomposition.
4) Combining our new MIG optimization method and our FRMIG mapper we show improvements to LUT mapping results.

The results of our experiments on logic optimization and LUT mapping include:

- Improvements to MIG logic optimization. We achieve a reduction in {size, depth} of {10%, 26%}, as compared to previous MIG optimization methods.
- Significant improvements after LUT mapping for several

benchmarks, reducing {size, area} by up to {13%, 56%} as compared to the best known results.
- Significant improvements in depth after LUT mapping for several benchmarks where we do not obtain size reductions. These results allow us to provide an area/depth trade-off.

The remainder of this paper is structured as follows. In Section II we present the background on MIGs, LUT mapping, structural and functional equivalence, and exact synthesis. We show how MIGs can be extended to FRMIGs in Section III and present our FRMIG LUT mapper in Section IV. We describe our new optimization method in Section V. Our MIG optimization and LUT mapping experiments can be found in Section VI.

II. Background

In order to improve on the state of the art in MIG optimization and LUT mapping we extend a number of existing techniques. Here we briefly present the necessary background in MIG synthesis and optimization, structural and functional equivalence, technology mapping, and exact synthesis.

A. Majority-Inverter Graphs

A Majority-Inverter Graph is a data structure for the representation and optimization of Boolean functions. It is a directed acyclic graph in which every vertex corresponds to a 3-input majority operator. We denote the 3-input majority function of variables a, b, and c as $\langle abc\rangle$, following the convention of [13]. In general the n-input majority function is equal to the value expressed by more than half of its inputs. Therefore, we can express the three-input $\langle abc\rangle$ in disjunctive normal form as:

$$\langle abc\rangle = ab \vee ac \vee bc.$$

Note that by setting any operand to either 0 or 1, the above equation simplifies to a Boolean *and* or *or* operation, respectively. Hence, MIGs are a generalization of *And-Or-Inverter Graphs* (AOIGs) and thus also of AIGs.

More formally, we can define an MIG over a set $X = \{x_1, \ldots, x_n\}$ of *primary inputs* as a directed acyclic graph $M = (V, E, Y)$ where

- $V = X \cup N \cup \{1\}$ a finite set of nodes, where $N = \{o_1, \ldots, o_m\}$ represents the nodes corresponding to the majority operations, and we also include the constant 1 node;
- $E \subseteq V \times N \times \mathbb{B}$ a finite multiset of edges, where the first element in the tuple is a source node, the second is a target node, and the third is a polarity bit representing whether or not the edge is complemented; and
- $Y \subseteq V \times \mathbb{B}$ a finite multiset of outputs.

Each node $n \in N$ must have three predecessors, reflecting its correspondence to a three-input majority operator. In the remainder of this paper, if $n \in N$, we write n_1, n_2, and n_3 to denote these inputs. We write p_1, p_2, and p_3 to denote the polarity bits indicating complementation of the inputs.

The functional semantics of an MIG can be described in terms of an interpretation function f that maps MIG nodes to a Boolean function. For convenience we define $f^0(v) = v$ and $f^1(v) = \overline{v}$. We define f as

$$\begin{aligned} f(1) &= \top \\ f(x) &= x && \text{for } x \in X \\ f(n) &= \langle f^{p_1}(n_1) f^{p_2}(n_2) f^{p_3}(n_3)\rangle && \text{for } n \in N \\ f(y) &= f^p(n) && \text{for } y = (n,p) \in Y \end{aligned}$$

A *sound* and *complete* Boolean algebra based on the 3-input majority operation is presented in [6], and its extension to n-input majority in [14]. There is a direct correspondence between the MIG structure and the axioms of these algebras. This means that we can use the algebraic rules directly to manipulate MIGs in a sound and complete way. This is the basis for the algebraic MIG optimizations shown in [6]. These algebraic optimizations are shown to lead to an average reduction of 22%, 14%, 11% in delay, area, power respectively, as compared to state-of-the-art academic and commercial synthesis tools.

Boolean MIG optimizations are presented in [7]. The majority operator possesses an inherent error-correcting mechanism. This enables us to insert *orthogonal* errors into an MIG structure that enable significant optimization opportunities. Boolean MIG optimization reduces the average delay/area/power by 15.07%, 4.93%, 1.93% respectively, over 27 academic and industrial benchmarks.

This section serves as a brief overview of the MIG structure and capabilities. For more complete descriptions the interested reader is referred to [6], [8], [14].

B. FPGA Technology Mapping

Generally speaking, an FPGA is an integrated circuit that consists of an array of programmable logic blocks and interconnects. Often these logic blocks are lookup tables that can implement arbitrary logic functions. Typically an FPGA also includes a number of non-programmable *hard blocks* that implement specific functionality, such as adders, multipliers, embedded memories, and even microprocessors. As we are reaching the limits of modern CMOS scaling, FPGA are seen as an approach to enable more computational capabilities in an energy efficient way [15], [16]. In order to make use of these advantages we require good logic optimization and technology mapping algorithms for FPGAs.

LUT mapping is the special case in which we cover a logic network with k-bounded lookup tables (k-LUTs). A k-LUT is a lookup table with k inputs; a powerful logic primitive that can represent any function on k variables.

A breakthrough in delay-oriented LUT mapping came with the introduction of the *FlowMap* algorithm [17]. FlowMap was the first algorithm to show how k-feasible cuts can be used to obtain a minimum-depth k-LUT cover. Several improvements of FlowMap have since been made. Some of these improvements include generalizing the algorithm to a more general cut enumeration basis, improving the runtime

and memory requirements, as well as improving different aspects of the final cover such as area reduction [11], [18]–[21].

More recently, developments in technology mapping have aimed at reducing *structural bias* [10], [22]. Structural bias is a problem of cut-based technology mapping algorithms. It is caused by the original decomposition of the network, which is not unique. One approach to mitigate this problem has been to use the concepts of *choice nodes* and *functional reduction* to merge structurally different equivalent networks into a single representation [9], [23]. *Functionally Reduced AIGs* (FRAIGs) are an example of such a representation [24]. The concept of functional reduction naturally leads to a so-called *lossless synthesis* flow. In lossless synthesis, multiple different logic networks may be merged into a single graph that can be used for technology mapping. The different networks correspond to points in the optimization space that are no longer lost in a lossless synthesis flow [12], [25].

The state-of-the-art in LUT mapping algorithms is based on cut enumeration. However, a drawback of cut-based algorithms is that they are susceptible the problem of structural bias [22]. Given a logic network, they find a cover based on the feasible cuts in that network. However, the given logic network is typically just one of many possible representations. Other networks (with different structure) may lead to different covers. Hence, the output of the algorithm is inherently biased by the structure of the input network. This is a problem because some covers may be "better" than other in terms of size or depth.

The notion of *lossless synthesis* was proposed in [22] as a way to mitigate the problem of structural bias. In a traditional optimization and mapping flow, an input network is optimized by applying one or more optimization scripts to it. Afterwards the optimized network is then mapped. This optimization process loses information: it may be that during optimization an intermediate networks is better in some respect than the final network. In lossless synthesis, multiple equivalent (intermediate) networks are stored, so that the desirable parts of the networks are never lost.

To efficiently store the different networks generated by lossless synthesis, they may be combined into a single network graph with *choice nodes* [23]. Intuitively, a choice node may be understood as a vertex in a logic network that encodes different implementations of the same function (up to complementation). Equivalent points in different networks can be found through a combination of simulation and combinational equivalence checking (SAT). This notion of finding equivalent points in logic networks leads to the notion of *functional reduction*. *Functionally Reduced* AIGs (FRAIGs) were first introduced in [24]. In FRAIGs, equivalent AND nodes are merged into choice nodes that represent different implementations of the same logic function. The current state of the art in FPGA technology mapping combines cut enumeration with the FRAIGing of differently optimized logic networks [10], [12], [24], [25].

C. Exact Synthesis

In exact synthesis, we are interested in finding the optimum representation for a Boolean function. Of course, optimality depends on the optimization objective (such as size or depth) as well as the chosen representation form. In this paper we use exact synthesis to refer to the optimum MIG representation.

Exact synthesis can be formulated as a decision problem and may be solved by an SMT solver. SMT solvers are constraint solvers for decision problems that can be formulated in first-order logic. Typically they work on formulas within some background theory, such as a theory of bit vectors. To see how we can use an SMT solver in exact synthesis, suppose that the solver has some method of checking if a given function on n variables can be represented by an MIG with k nodes. In order to find the smallest possible MIG for that function, we start checking with $k = 0$, and we keep increasing k until we have found a k for which an MIG exists that can represent the function. Similar algorithms can be used to find the minimum depth or to optimize other criteria. The checking method can be implemented by formalizing the exact synthesis criteria (e.g. size or depth) as an SMT theory. An in-depth, formal example of how this may be done can be found in [8]. Thus, we can use an SMT solver for exact synthesis.

The computational complexity of exact synthesis as described here prevents it from being used to synthesize arbitrarily large function. SMT is a generalization of SAT and is therefore NP-hard. As such, the time complexity of this approach grows at least exponentially with n. However, for small enough n ($n \leq 4$) it has been shown to work [8]. In Section V we show how we can use exact synthesis to decompose k-LUT networks (for small enough k) into locally optimal sub-MIGs. This is the basis for our optimization algorithm.

D. Structural and Functional Equivalence

MIGs are not canonical representation forms: a Boolean function may have structurally different but *functionally equivalent* MIG representations. See for example Fig. 1 in which two different MIGs represent the function $f = \overline{abc}$. Other homogeneous networks, such as AIGs, have similar limitations with different structures representing equivalent functions.

Despite the non-canonical nature of MIGs, there are still methods to remove redundancies and simplify MIG structures. One such method is *structural hashing*. With structural hashing, before adding a node to an MIG, we check if a node with the same fan-in already exists. This can be done efficiently with a hash table. Typically there is an order on the fan-ins of the node so that swapping them does not create a different node. Thus, structural hashing merges nodes that are *structurally equivalent*. Fig. 2 shows how structural hashing can be used to reduce the node count and remove redundant nodes.

In order to reduce structural bias we would like to be able to find and merge equivalent nodes, even if they are not structurally equivalent. Nodes with different fan-ins may still be *functionally equivalent*. The process of merging functionally

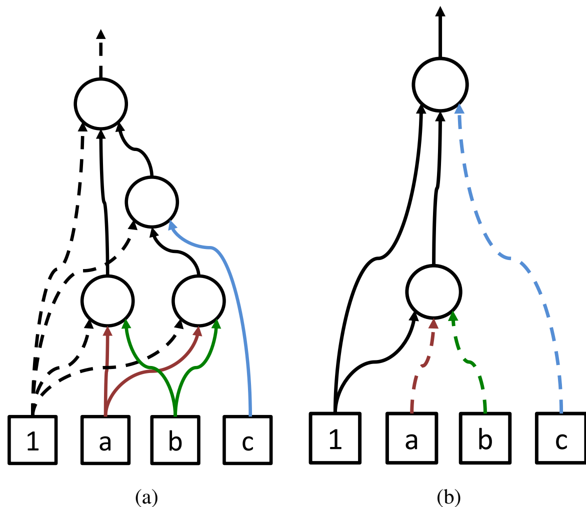

Fig. 1. An example of two structurally different, but functionally equivalent MIGs for the function $f = \overline{abc}$. Complementation is indicated by dashed edges.

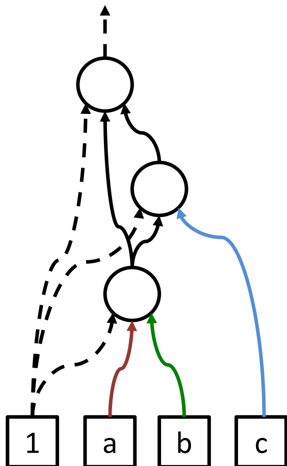

Fig. 2. Structural hashing can be used on the MIG in Fig 1 (a) to remove one redundant node.

equivalent nodes is commonly referred to as functional reduction. Functional reduction is typically achieved through BDD or SAT sweeping [26], but can also be achieved incrementally during construction [24].

Storing functionally equivalent nodes efficiently can be achieved by *choice nodes*. Choice nodes were developed in the 90s as a way to efficiently encode sets of logic networks to be considered in technology mapping [9], [23]. Intuitively, one can think of a choice node as representing an equivalence class of functions (up to complementation). Fig. 1 shows two structurally different MIGs that are equivalent up to complementation. Using only structural hashing we would not be able to find this equivalence. However, through a combination of simulation and SAT sweeping we can efficiently find these equivalence points. We refer to an MIG in which functionally equivalent nodes have been merged into choice nodes as a *Functionally Reduced MIG* (FRMIG). An FRMIG reduces structural bias by containing different equivalent structures at once.

III. Functionally Reduced MIGs

In Section II-D we reviewed the concepts of structural and functional equivalence. Applying these concepts to MIGs leads to the concept of the *Functionally Reduced* MIG (FRMIG). Here, we define FRMIGs as well as a procedure to construct FRMIGs from MIGs.

Given an MIG $M = (V, E, Y)$, we define a *functional equivalence* relation on the nodes in V. A relation $\simeq$ is a functional equivalence relation if for all $v_1, v_2 \in V$

$$v_1 \simeq v_2 \Rightarrow (f(v_1) = f(v_2)) \lor (f(v_1) = \neg f(v_2))$$

Let $V_{\simeq}$ denote the set of equivalence classes of V under $\simeq$. We say that a subset $\mathcal{V}$ is a *set of functional representatives* iff it contains exactly one node v_C from each equivalence class $C \in V_{\simeq}$. We say that v_C is the *functional representative* of C.

Let $\widehat{M} = (V, E, Y)$ be an MIG, $\simeq$ a functional equivalence relation, and $\mathcal{V}$ a set of functional representatives. Then, $\widehat{M}$ is a *Functionally Reduced* MIG w.r.t. $\simeq$ and $\mathcal{V}$, iff $(v, n, b) \in E \Rightarrow v \in \mathcal{V}$. In other words, only equivalence class representatives have outgoing edges. This means that for any $n \in N$ its predecessors n_1, n_2, n_3 are in $\mathcal{V}$.

We sometimes loosely refer to equivalence class representatives as *choice nodes*. This follows the intuition that each choice node represents a set of alternative implementations for an equivalence class of functions.

Note that according to this definition, any MIG is trivially an FRMIG. Suppose we have an arbitrary MIG $\widehat{M}_T = (V, E, Y)$. Let $\simeq_T$ be the relation $n_1 \simeq_T n_2$ iff $n_1 = n_2$. One can easily verify that this is a functional equivalence relation. Let $\mathcal{V}_T = V$. Then $\widehat{M}_T = (V, E, Y)$ is an FRMIG under $\simeq_T$ and $\mathcal{V}_T$. However, this trivial definition is not very interesting. In the remainder of this paper we assume that for every FRMIG the functional equivalence relation is defined as: *equality up to complementation*.

Fig. 3 shows how the functionally equivalent MIGs from Fig. 1 can be merged in the same FRMIG structure. Note how only one of the equivalent nodes is chosen as the equivalence class representative and has an outgoing edge.

We now sketch a procedure for constructing an FRMIG from an MIG. The procedure starts by simulating the MIG on a number of random bit vectors. We then traverse the MIG in topological order, creating new nodes only after structural hashing. When a new node is created, we check its simulation vector to see if there are any potentially equivalent nodes. If an equivalence between two nodes is found, the node which appears earlier in the topological order is chosen to be the equivalence class representative. Note that this means that primary inputs are always chosen to be the representatives of their equivalence classes. One can intuitively think of the

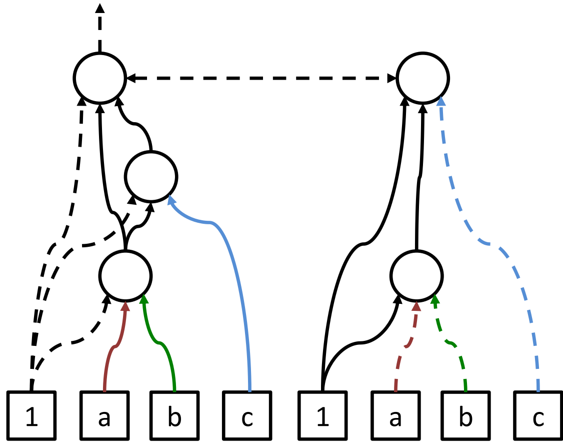

Fig. 3. This figure demonstrates the use of *choice nodes* in FRMIGs. Two equivalent (up to complementation) MIG structures are represented by one equivalence class representative. This representative is implemented as a choice node with one outgoing edge. The horizontal dashed line indicates equivalence between the MIGs. It is dashed because both MIG structures represent the complement of the other.

equivalence class representative as a choice node containing MIG structures that are equivalent up to complementation.

Equivalence checking of nodes is done using the SMT solver Z3 [27]. As the MIG is traversed we incrementally add clauses to the solver that represent the functionality of the nodes, similar to how the Tseitin transformation is used for combinational equivalence checking. Suppose that we have two nodes n_1 and n_2 that have an identical simulation vector. In order to check their equivalence we proceed as follows:

1) Run the solver with the assumption $\neg n_1 \wedge n_2$. This checks if there is any interpretation under which n_1 is false and n_2 is true. If so, then the two nodes are clearly not equivalent and we return false. If not, the nodes may be equivalent.
2) We then run the solver with the assumption $n_1 \wedge \neg n_2$, checking the converse. Again, if this assumption is SAT, we return false. If this is UNSAT, we have now shown that the nodes are equivalent and we return true.

This procedure can be extended in a simple way to check for equivalence up to complementation, as well as to support an *undefined* result if we want to bound the running time of equivalence checks.

A procedure that is similar to the one above can be used to enable the merging of multiple MIGs into a single FRMIG structure. Merging MIGs together enables a lossless MIG synthesis and optimization flow, in which we can take advantage of different optimization points during technology mapping.

IV. FRMIG LUT Mapping

FRMIGs are an extension of MIGs that reduce structural bias by combining multiple MIG structures. To take advantage of this new representation we present a LUT mapper that maps FRMIGs to k-LUTs. The architecture of this mapper is based on the work in [25]. As LUT mapping is based on cut enumeration, we first show how to enumerate the cuts of an MIG. We then show how cut enumeration for MIGs may be extended to FRMIGs.

Let n be a node in an MIG. A *cut* of n can be defined as a set c of nodes in its transitive fan-in such that every path from a primary input to n passes through a node in c. We say a cut c of n is redundant is there is some $c' \subset c$ such that c' is also a cut of n. A k-feasible cut of n is an irredundant cut c such that $|c| \leq k$. When mapping a MIG we are only interested in finding k-feasible cuts, since any larger cuts cannot be covered by k-LUTs. For any node n, we consider the *trivial cut* $\{n\}$ to be a valid k-feasible cut.

Given an MIG $M = (V, E, Y)$ on primary inputs X, we use $\Phi(v)$ to denote the set of k-feasible cuts for $v \in V$. We may define Φ recursively as:

$$\begin{aligned} &\Phi(1) = \{\{\}\} \\ &\Phi(x) = \{\{x\}\} && \text{for } x \in X \\ &\Phi(n) = \{\{n\}\} \cup (\Phi(n_1) \otimes \Phi(n_2) \otimes \Phi(n_3)) && \text{for } n \in N \end{aligned}$$

where $\otimes$ is an an operation that gives an over-approximation of the k-feasible cuts of a node. It is defined as

$$A \otimes B = \{a \cup b \mid a \in A, b \in B, |a \cup b| \leq k\}$$

This definition of Φ may lead to the inclusion of some redundant cuts, but these can be easily filtered out during cut enumeration. Details of efficient cut computation are discussed in [25] and [10].

In order to support FRMIGs, we need to extend the above cut enumeration procedure to work with choice nodes. Let $M = (V, E, Y)$ be an FRMIG under $\simeq$ and $\mathcal{V}$. We compute cuts for both equivalence classes in $\mathcal{V}$ and nodes in V as follows. The set of cuts for an equivalence class $C \in \mathcal{V}$ is

$$\Phi_{\simeq}(C) = \bigcup_{n \in C} \Phi(n)$$

In other words, the set of cuts for an equivalence class is simply the union of the cuts of all the nodes in that class.

We now alter the definition of Φ to

$$\begin{aligned} &\Phi(1) = \{\{\}\} \\ &\Phi(x) = \{\{x\}\} && \text{for } x \in X \\ &\Phi(n) = \{\{n\}\} \cup (\Phi_{\simeq}(n_1) \otimes \Phi_{\simeq}(n_2) \otimes \Phi_{\simeq}(n_3)) && \text{for } n \in \mathcal{V} \\ &\Phi(n) = \Phi_{\simeq}(n_1) \otimes \Phi_{\simeq}(n_2) \otimes \Phi_{\simeq}(n_3) && \text{otherwise} \end{aligned}$$

Note that we include the trivial cut only for equivalence class representatives. As a consequence, for any cut c, $n \in c \Rightarrow n \in \mathcal{V}$.

After extending cut enumeration to work with choice nodes, our mapper can essentially be defined as a traditional cut-based FPGA mapper. Suppose we want to map an FRMIG $\widehat{M}$ to k-LUTs. We can do so by traversing $\widehat{M}$ in topological order, enumerating k-feasibly cuts for all $n \in \mathcal{V}$. We compute a cost function for all cuts we find. The specific cost of a cut depends on the optimization objective. Typically, in depth-oriented mapping, the cost is a function of the cut depth. After

computing the cuts and their costs, a k-LUT cover is easily found by traversing the FRMIG in reverse topological order from outputs to inputs.

V. Exact MIG Optimization

In Section II-C we reviewed the notion of *exact synthesis*. We may use exact synthesis on small functions in order to synthesize MIGs that are optimal with respect to some optimization objective. We show here how this idea may be used in MIG optimization.

We first introduce some notation that will be useful in our discussion. Let $\mathcal{N}$ be a logic network. We use $d(n)$ to indicate the depth of nodes n in $\mathcal{N}$. If n is a primary input node, then $d(n) = 0$. Otherwise, let $\text{inputs}(n)$ denote the set of inputs of n. The depth of n is then $d(n) = \max\{d(i) \mid i \in \text{inputs}(n)\} + 1$. Let $\text{outputs}(\mathcal{N})$ be the set of outputs of $\mathcal{N}$. We use $d(\mathcal{N}) = \max\{d(o) \mid o \in \text{outputs}(\mathcal{N})\}$ to denote the depth of the logic network itself.

Let M be a MIG that computes some Boolean function $f(X)$ over $|X| = n$ variables. Suppose we map M to a depth-optimal k-LUT network L_k ($k \geq 3$). Then $d(L_k)$ is a lower bound on $d(M)$, since the k-LUTs can compute any function on k variables. Note that we do not mean that $d(L_k) \leq d(M')$ for *any* MIG M' that computes $f(X)$. The lower bound is just on the specific structure of M. We call this a *structural lower bound*. To see why L_k gives a structural lower bound, note that any k-feasible cuts that contain multiple levels of majority nodes may be merged into a single k-LUT, thereby decreasing the depth of L_k relative to M. This means that for any depth-optimal cover L_k we have $d(L_k) \leq d(M)$. As k increases, $d(L_k)$ decreases, since we are able to merge more functionality into the k-LUTs. In other words

$$\lim_{k \to \infty} d(L_k) = 1$$

since, as k approaches ∞, the entire logic network may be merged into a single k-LUT.

In addition to giving a structural lower bound, L_k also removes structural bias. In the extreme case, $k \geq n$ and all structure is removed, because M can be represented by single k-LUT. The LUT can be viewed as a specification of $f(X)$. Different structural representations can be extracted from the network by decomposing the LUT in different ways. Suppose we could use exact synthesis to decompose the LUT into a new MIG M'. We would then have found an optimal MIG representation M' of $f(X)$.

In practice, we cannot map most MIGs into a single k-LUT, because the LUT would require an efficient way of representing $f(X)$. Presumably, the most efficient representation available for $f(X)$ is the MIG itself. Suppose we could somehow map the MIG into a single LUT and efficiently represent $f(X)$. Using exact synthesis to decompose the LUT would still be infeasible, since n could be arbitrarily large. However, for smaller k, this method of mapping MIGs into k-LUTs becomes feasible. If k is "small enough", we can represent the different k-LUT functions, and we can use exact synthesis to decompose them. We do not remove all of the structural bias, but still enough to provide alternative optimization points. This is the basis for our exact k-LUT optimization procedure.

Algorithm 1: An MIG size optimization procedure using LUT mapping and exact synthesis.

```
function size-optimize(M, k) :=
Input  : MIG M
Output: Optimized MIG M'
M' ← M;
do
    M ← M';
    M' ← new_mig();
    Perform area-oriented mapping of M into k-LUTs;
    foreach primary input i in M do
        create_input(M', i);
    end
    foreach LUT l in the cover in topological order do
        f ← function computed by l;
        opt_mig ← exact_mig(f);
        create_nodes(M', opt_mig);
    end
while size(M') < size(M);
return M';
```

This optimization method is not limited to reducing MIG depth. Instead of mapping into depth-optimal k-LUT networks, we could for instance focus on finding small k-LUT covers. Decomposing small k-LUT covers into locally minimum-sized MIGs can then be used to decrease MIG size.

The results of the method depend not only on the k-LUT cover, but also on the exact synthesis optimality criteria. For example, we may choose to decompose k-LUTs into locally size-optimal MIGs or into locally depth-optimal MIGs. Different choices lead to different optimization results. In a lossless synthesis flow these different optimization points may then be merged into a single FRMIG structure.

We present a MIG size optimization procedure based on this exact optimization method. It takes an MIG M and proceeds as follows. First, we map M into a k-LUT network while heuristically minimizing area. We then use exact synthesis to decompose the LUTs into minimum-size sub-MIGs. These are connected to produce a functionally equivalent MIG M'. This process is iterated until the area of M' is the same as that of M. The pseudocode for this procedure can be found in Algorithm 1.

Depth optimization techniques for MIGs are well developed. For example, the best known results for depth-optimal LUT mapping on the EPFL benchmark suite[1] were obtained by combining MIG optimization with the iterative application of ABC's technology mapper. MIG size optimization techniques, on the other hand, are less well developed. The current state of the art in MIG size optimization is based on *functional hashing* [8]. However, this approach results in only modest size reductions. One drawback to this approach is that it has only a local view of size reductions. Furthermore, the best size

[1] lsi.epfl.ch/benchmarks

reductions come at the cost of increasing MIG depth.

Our aim is to use MIGs as a general purpose logic synthesis representation. As such, we require methods that work well for both depth and size optimization. In Section VI we show how our new size optimization procedure improves MIG size optimization, thereby extending the range of logic synthesis and optimization objectives to which MIGs can be applied.

VI. Experiments

We show here that our new optimization method enables significant improvements in both logic optimization and LUT mapping.

1) Methodology: We have developed a C++ logic manipulation package that implements the size optimization procedure described in Section V. It uses the methods presented in [8] to perform exact synthesis. The largest k for which exact MIGs were available at the time of these experiments was $k = 4$. Therefore, in these experiments our procedure maps MIGs into 4-LUTs. We run our experiments on the EPFL arithmetic benchmark suite. The choice for this suite was motivated by the fact that it contains relatively large circuits, as compared to other well known benchmarks suites. Note that our procedure is not limited to just arithmetic benchmarks.

2) Results: We first compare our new MIG size optimization procedure to the functional hashing approach described in [8] The results of the comparison can be found in Table I. We refer to our new procedure as *LUT-based size optimization*.

LUT-based size optimization reduces both depth and area by about 50% for the adder benchmark as compared to the functional hashing algorithm. On average functional hashing reduces MIG size by 3%, whereas LUT-based size optimization reduces MIG size by an average of 13%. Furthermore, LUT-based size optimization reduces MIG depth by 19% on average. Functional hashing, on the other hand, adds an average of 7% in depth to obtain its area reductions.

We can see that functional hashing has an inversely proportional relation between area and depth: it *increases* depth by 2.33% for every 1% in area reduction. LUT-based size optimization turns this into a directly proportional relation: it *decreases* depth by 1.46% for every 1% reduction in size.

In the next experiment we apply our LUT mapper to the size-optimized MIGs. We compare the results with the state-of-the-art LUT count results for the EPFL arithmetic benchmark suite. These best results were obtained by using ABC's AIG size optimization and LUT mapping algorithms. The results can be found in Table II.

Our procedure performs significantly better on three of the benchmarks. It reduces area/depth by 4.5%/13.3% for the Adder, by 8.7%/15% for the Multiplier, and by 13%/56% for the Square. In other words, it is able to significantly reduce area, while simultaneously reducing depth.

On the Log2 and Max benchmarks, our procedure increases area by 3.3% and 30%, respectively. However, it reduces depth by 7.8% and 41.2%, respectively. These results present an interesting area/depth trade-off.

Our mapper performs worse than ABC in the Sine benchmark, with an increase in area and depth of 4% and 6.4%, respectively. ABC performs significantly better on the Divisor and Square-root benchmarks. We do not know which specific techniques were used to generate ABC's best results. As such, it is difficult to determine what specifically allowed it to perform so much better on these particular benchmarks. We do know that, in order to obtain these results, a variety of ABC commands were used. These commands were iterated, always storing the best global result, until no further improvement was found [28]. Our results, on the other hand, were obtained by a single LUT mapping pass.

VII. Conclusions and Future Work

The goal of this paper was to improve techniques for MIG logic optimization and LUT mapping. In order to compete with the state-of-the-art in LUT mapping we have extended MIGs to FRMIGs. This allows us to reduce structural bias and to use MIGs in a lossless synthesis flow.

Using our FRMIG mapper, we have introduced a general MIG optimization method based on k-LUT mapping and exact k-LUT decomposition. This method can be used in both depth and area optimization. We have shown how we can obtain up to 50% better results in both MIG size and area as compared to previous approaches in MIG size optimization. When compared to functional hashing, our approach reduces area and depth by 10% and 26%, respectively.

Using our new optimization method and LUT mapper we also obtain significantly better results in both LUT count and LUT depth for 3 of the heavily optimized EPFL arithmetic benchmarks, reducing area and depth by up to 13% and 56%, respectively.

In our experiments we were limited to use exact synthesis for functions on up to 4 variables. We expect that increasing the number of variables will significantly improve our optimization method. Preliminary results also show that our optimization procedure has promise in MIG depth optimization and improved depth-optimal LUT mapping.

Acknowledgment

This research was supported by SNSF-200021-146600 and H2020-ERC-2014-ADG 669354 CyberCare.

References

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

[2] A. L. Sentovich, E.M. and Singh, K.J. and Lavagno, L. and Moon, C. and Murgai, R. and Saldanha, A. and Savoj, H. and Stephan, P.R. and Brayton, Robert K. and Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. May 1992, 1992.

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

[4] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," *43rd ACM/IEEE Design Automation Conference*, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1146909.1147048

[5] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Computer Aided Verification*, vol. 6174, 2010, pp. 24–40.

TABLE I
COMPARISON OF MIG SIZE OPTIMIZATION METHODS

Benchmark	I/O	Size	Depth	**Functional Hashing** Size	**Functional Hashing** Depth	**LUT-based Size Optimization** Size	**LUT-based Size Optimization** Depth
Adder	256/129	1020	255	1020	255	513	130
Barrel shifter	135/128	3336	12	3240	15	3336	12
Divisor	128/128	57247	4372	57247	4403	35993	3607
Hypotenuse	256/128	214335	24801	214307	24878	196842	11317
Log2	32/32	32060	444	29795	461	32060	444
Max	512/130	2865	287	2865	337	2479	276
Multiplier	128/128	27062	274	24903	271	22634	165
Sine	24/25	5416	225	5254	230	5416	255
Square-root	128/64	24618	5058	24618	6021	24170	6073
Square	64/128	18484	250	17893	250	17562	130
Average improvement (new/old):				0.97	1.07	**0.87**	**0.81**

TABLE II
COMPARISON OF 6-LUT AREA MINIMIZATION

Benchmark	I/O	Size	Depth	**ABC** Size	**ABC** Depth	**MIG Mapper** Size	**MIG Mapper** Depth	**Size** (MIG/ABC)	**Depth** (MIG/ABC)
Adder	256/129	1020	255	201	73	192	64	0.96	0.87
Barrel shifter	135/128	3336	12	512	4	512	4	1.00	1.00
Divisor	128/128	57247	4372	3813	1542	10328	1915	2.70	1.24
Log2	32/32	32060	444	7344	142	7592	131	1.03	0.92
Max	512/130	2865	287	532	192	692	113	1.30	0.59
Multiplier	128/128	27062	274	5681	120	5192	102	0.91	0.85
Sine	24/25	5416	225	1347	62	1402	66	1.04	1.06
Square-root	128/64	24618	5058	3286	1180	6810	2070	2.07	1.75
Square	64/128	18484	250	3800	116	3309	74	0.87	0.64

[6] L. Amarú, P.-E. Gaillardon, and G. D. Micheli, "Majority-Inverter Graph : A Novel Data-Structure and Algorithms for Efficient Logic Optimization," in *DAC*, 2014.

[7] L. Amarú, P.-E. Gaillardon, and G. D. Micheli, "Boolean Logic Optimization in Majority-Inverter Graphs," in *Proc. of Design Automation Conf. (DAC)*, 2015.

[8] M. Soeken, L. G. Amarù, P.-e. Gaillardon, and G. D. Micheli, "Optimizing Majority-Inverter Graphs With Functional Hashing," in *Design Automation and Test in Europe*, 2016, pp. 1030–1035.

[9] G. Chen and J. Cong, "Simultaneous Logic Decomposition with Technology Mapping in FPGA Designs," in *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, 2001, pp. 48–55. [Online]. Available: http://portal.acm.org/citation.cfm?doid=360276.360298

[10] S. Chatterjee, "On Algorithms for Technology Mapping," Ph.D. dissertation, University of California at Berkeley, 2007.

[11] A. Mishchenko and R. Brayton, "Combinational and Sequential Mapping with Priority Cuts," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 354–361.

[12] A. Mishchenko, R. Brayton, and S. Jang, "Global Delay Optimization Using Structural Choices," in *FPGA '10*, 2010, pp. 181–184. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1723112.1723144

[13] D. E. Knuth, *The Art of Computer Programming, Volume 4A, Part 1*, 1st ed. Addison-Wesley Professional, 2011.

[14] L. Amarú, P.-E. Gaillardon, A. Chattopadhyay, and G. De Micheli, "A Sound and Complete Axiomatization of Majority-n Logic," 2016.

[15] C. Märtin, "Multicore Processors: Challenges, Opportunities, Emerging Trends," in *Embedded World Conference*, 2014, pp. 25–27.

[16] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, G. Jan, G. Michael, H. Scott, H. Stephen, A. Hormati, J.-y. K. Sitaram, L. James, and L. Eric, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services - Microsoft Research," in *41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[17] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[18] J. Cong and Y. Ding, "On Area/Depth Trade-Off in LUT-Based FPGA Technology Mapping," in *IEEE Transactions on Very Large Scale Integration Systems*, 1994, pp. 137–148.

[19] J. Cong and Y.-Y. Hwang, "Simultaneous Depth and Area Minimization in LUT-based FPGA Mapping," in *Third International ACM Symposium on Field-Programmable Gate Arrays*, 1995, pp. 68–74.

[20] J. Cong, C. Wu, and Y. Ding, "Cut Ranking and Pruning: Anabling A General And Efficient FPGA Mapping Solution," in *FPGA '99*, 1999, pp. 29–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=296425

[21] D. Chen and J. Cong, "DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs," in *International Conference on Computer Aided Design*. IEEE, 2004, pp. 752–759.

[22] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing Structural Bias in Technology Mapping," *IEEE/ACM International Conference On Computer Aided Design*, pp. 512–526, 2005.

[23] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic Decomposition During Technology Mapping," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, 1997, pp. 813–834.

[24] A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton, "FRAIGs : A Unifying Representation for Logic Synthesis and Verification," Tech. Rep., 2005.

[25] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, 2007, pp. 240–253.

[26] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[27] L. 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.

[28] R. K. Brayton and A. Mishchenko, "Private Communication," 2016.