Abstract—Given a set of logic primitives and a Boolean function, exact synthesis finds the optimum representation (e.g., size or depth) of the function in terms of the primitives. Due to its high computational complexity, exact synthesis is limited to small networks. Some logic rewriting algorithms use exact synthesis to replace small subnetworks by their optimum representations. However, conventional approaches have two major drawbacks. First, their scalability is limited, as Boolean functions are enumerated to precompute their optimum representations. Second, the strategies used to replace subnetworks are not satisfactory. We show how the use of exact synthesis for logic rewriting can be improved. To this end, we propose a novel method that includes various improvements over conventional approaches: (i) we improve the subnetwork selection strategy, (ii) we show how enumeration can be avoided, allowing our method to scale to larger subnetworks, and (iii) we introduce XOR Majority Graphs (XMGs) as compact logic representations that make exact synthesis more efficient. We show a 45.8% geometric mean reduction (taken over size, depth, and switching activity), a 6.5% size reduction, and depth size reductions of 8.6%, compared to the academic state-of-the-art. Finally, we outperform 3 out of 9 of the best known size results for the EPFL benchmark suite, reducing size by up to 11.5% and depth up to 46.7%.

I. INTRODUCTION

Boolean and algebraic methods have been the driving force in multi-level logic optimization [1]. Compared to two-level logic, exact optimization for multi-level logic networks has turned out to be more difficult due to the high computational complexity [2]–[4]. Despite this, the combination of Boolean and algebraic methods has enabled sufficient optimizations to increasingly complex logic networks.

In recent years, we have seen a shift from complex heterogeneous logic representations to simpler homogeneous networks such as And-Inverter Graphs (AIGs) [5], [6] and Majority-Inverter Graphs (MIGs) [7]–[9]. These simpler representations enable more efficient logic representation and optimization, requiring less memory and allowing better runtimes [10]. They also permit logic rewriting algorithms, which work by locally replacing subnetworks by their precomputed optimized representations [5], [11]. One drawback of conventional rewriting algorithms is that they require enumeration of the space of Boolean functions in order to precompute the optimum networks. However, this space is so large that enumeration quickly becomes intractable. Therefore, in practice this rewriting approach is limited to functions on 4 variables, or has to give up exactness for heuristic results.

Exact synthesis is the problem of finding an optimum network for a given function. The cost criteria are typically (although not limited to) size or depth of the network. Due to its high computational complexity, exact synthesis has not been the main driver behind many logic optimization algorithms. However, the combination of logic rewriting and exact synthesis has recently led to improvements in both AIG and MIG size optimization [5], [12]–[14]. By precomputing size-optimum AIGs for a subclass of NPN classes of 5-variable functions, the area of highly optimized large networks may be reduced by 5.57% on average [12]. A similar method has recently been introduced for MIG size optimization [13].

Our goal is to improve the use of exact synthesis in logic rewriting, with a focus on size optimization. The major problems of state-of-the-art approaches are: (i) the enumeration of Boolean functions to precompute optimum networks, thus limiting the approach to small subnetworks, and (ii) the non-satisfactory strategies to select subnetworks for rewriting.

We propose a number of solutions to overcome these problems. We show how enumeration of the Boolean space can be avoided by computing optimum representations only for functions that occur in practice. We find these functions by using LUT mapping and NPN canonization as filters. This also allows us to construct a database of (classes of) Boolean functions that occur in practice. We refer to this process as mining for Boolean functions. Additionally, LUT covers turn out to contain better selections of subnetworks than those used by previous approaches, thus improving the subnetwork selection strategy. Finally, we introduce XOR Majority Graphs (XMGs) and use them as underlying logic representation for exact synthesis. XMGs enable compact logic representation. This decreases the runtime of exact synthesis, especially when combined with improvements to the exact synthesis algorithm introduced in [13].

Our experiments on the EPFL benchmark suite1 demonstrate the improvements enabled by our method:

- We show a 45.8% reduction in geometric mean (taken over size, depth, and switching activity), a 6.5% average reduction in size, and a 8.6% improvement in the depth·size measure for area-oriented k-LUT technology mapping, as compared to the academic state-of-the-art ABC package.
- We improve the currently best known results for area optimized networks in 3 out of 9 cases, showing improvements in size and depth up to 11.5% and 46.7%, respectively.

1http://lsi.epfl.ch/benchmarks
We discuss and outline the possibilities of such a framework VI. Finally, our proposed approach opens up a new field of research in which parallel and distributed computing power is invested to re-synthesize networks using exact synthesis methods. Distributed systems and compute clusters can be used to search and mine for optimum network representations. We discuss and outline the possibilities of such a framework in Section VII.

II. BACKGROUND

In this section, we introduce some useful notation, as well as the concepts behind NPN classification and LUT mapping.

A. Boolean Functions

We consider Boolean functions \( f(x_1, \ldots, x_n) \) over \( n \) variables. In an expression for \( f \) a variable \( x_i \) can either appear as positive literal \( x_i \) or as a negative literal \( \bar{x}_i \). The central Boolean operations in this paper are the exclusive OR (XOR):

\[
    x \oplus y = x\bar{y} \lor \bar{x}y = (x \lor y)(\bar{x} \lor \bar{y})
\]

(1)

and the majority of three (MAJ):

\[
    \langle xyz \rangle = x\lor z \lor yz = (x \lor y) (x \lor z) (y \lor z).
\]

(2)

The Boolean operations AND and OR can be obtained from MAJ by setting one of its arguments to 0 or 1, respectively:

\[
    \langle 0xy \rangle = x \land y \quad \text{and} \quad \langle 1xy \rangle = x \lor y
\]

(3)

The MAJ operation is self-dual [15] since \( \langle \bar{x}y\bar{z} \rangle = \langle xy\bar{z} \rangle \). We refer to this property as inverter propagation. Several other interesting properties for the MAJ operation exist [16]–[18], but are not important in the course of this paper.

The MAJ and XOR operators interact in a natural way. XOR operators propagate through MAJ operators, much like inverters do:

\[
    a \oplus \langle xyz \rangle = \langle (x \oplus a)(y \oplus a)(z \oplus a) \rangle
\]

(4)

The XOR operation inverts one of its operands if the other one is set to 1, i.e., \( x \oplus 1 = \bar{x} \). The operation is not self-dual but also allows to propagate inverters, since:

\[
    x \oplus y = \bar{x} \oplus \bar{y} = \bar{x} \oplus y = x \oplus y
\]

(5)

and:

\[
    \bar{x} \oplus y = x \oplus \bar{y} = \bar{x} \oplus \bar{y} = x \oplus \bar{y}.
\]

(6)

B. NPN Classification

Two functions \( f(x_1, \ldots, x_n) \) and \( g(x_1, \ldots, x_n) \) are NPN-equivalent, if there exists a permutation \( \sigma \in S_n \) and polarities \( p_1, p_2, \ldots, p_n \in \mathbb{B} \) such that

\[
    f(x_1, \ldots, x_n) = g(p_\sigma(x_1), \ldots, p_\sigma(x_n)),
\]

(7)

i.e., \( g \) can be made equivalent to \( f \) by negating inputs, permuting inputs, or negating the output. NPN-equivalence is an equivalence relation that partitions the set of all Boolean functions over \( n \) variables into a smaller set of NPN classes. As an example, all \( 2^n \) Boolean functions over \( n \) variables can be partitioned into 2, 4, 14, 222, 616,126 NPN classes for \( n = 1, 2, 3, 4, 5 \). For a detailed introduction into NPN classification, the reader is referred to [19], [20].

C. LUT Mapping

LUT mapping is the special case of technology mapping in which we cover a logic network with \( k \)-LUTs (\( k \)-input lookup tables). The state-of-the-art in LUT mapping is based on \( k \)-feasible cut enumeration [21]. Depth-optimal LUT mapping was first made tractable with the introduction of the FlowMap algorithm [22]. 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 [23]–[26]. Although depth-optimal LUT mapping is a solved problem, area-optimal LUT mapping is NP-hard and remains an open problem [27]. However, different effective area-recovery heuristics have been proposed [25], [28].

III. OPTIMIZATION METHOD OVERVIEW

Fig. 1 gives an overview of our proposed method. It is applicable to any \( k \)-bounded network, i.e., a network in which each gate has at most \( k \) inputs. Note that, if a network is not \( k \)-bounded, it may be decomposed to obtain a functionally equivalent \( k \)-bounded network [22]. Therefore, in the sequel, we will assume, without loss of generality, that input networks are \( k \)-bounded. In Section V, we describe in detail our specialization of this method for XMG size optimization.

The input to our method is a parameter \( k \) and a \( k \)-bounded logic network \( N \). We first perform LUT mapping on \( N \) in order to find a suitable \( k \)-LUT cover. As our goal is size optimization, a suitable cover is one that minimizes the number of LUTs, and we use the appropriate heuristics to obtain it. After finding a cover, we compute the NPN classes for the functions of the LUTs in the cover. We then invoke exact synthesis for these NPN classes, producing locally optimum subnetworks. The results of exact synthesis are saved in a database that stores the optimum representations of the NPN classes we have encountered. These results may be reused in subsequent iterations. Finally, the locally optimum networks are merged together to create an optimized, functionally equivalent, network \( N' \). This optimization process may be iterated on
Our optimization method has some similarities to earlier AIG rewriting optimizations [5], [11], [12]. These methods also find \( k \)-feasible cuts to obtain replacement subnetworks. One difference is that our method does not rely on the enumeration of Boolean functions and their optimum subnetworks. This is one of the key differences which allows our method to scale. Enumeration of functions becomes unpractical for \( k > 4 \). For example, there are \( 2^{2^6} \) 6-variable functions, with 200,253,952,527,185 corresponding NPN classes. Suppose that the average computation time required to find the optimum representation for these functions is 0.002 seconds\(^2\). Even if we were to obtain, through some oracle, a list of the NPN classes, it would still take over 12,700 years to synthesize all their optimum representations. Therefore, avoiding enumeration is crucial to obtain tractable runtimes. We avoid it by computing optima only for those NPN classes that occur in a cover. Thus, we only examine a small portion of the total number of Boolean functions. In other words, we mine the space of “useful” Boolean functions that occur in practice. This greatly reduces the computation time required by our approach, and makes exact synthesis tractable for \( k > 4 \). For example, when mining the EPFL benchmarks for 6-variable functions we only find 286 unique NPN classes.

Another difference between our method and previous approaches is in the subnetwork selection heuristic. We select those subnetworks that appear in the LUT cover. This turns out to be a selection heuristic that compares favorably to the heuristics used by [5], [12]. Those approaches rely on purely local information, whereas LUT mapping may use heuristics with a global view. Thus, LUT mapping improves subnetwork selection.

The approach in [11] also mines for useful circuit structures. However, it is aimed at depth optimization, whereas we focus on size. Additionally, the results in [11] are not exact, but are rather based on heuristic optimization.

Finally, our approach is distinct from remapping methods [30], [31]. For example, the method in [30] iteratively improves mapped circuits, by symbolically optimizing Boolean relations with a specified cell library. In contrast, ours is a technology independent logic optimization method that uses SAT or SMT for optimization. Additionally, the runtime of our method may be improved by mining circuits for useful functions in advance.

### IV. XOR Majority Graphs

We will present an instance of the optimization flow presented in the previous section based on XOR Majority Graphs (XMGs) in the remainder of this paper. An XMG is a logic network in which each gate corresponds to either a MAJ or a XOR operator. The connections between gates can be inverted. XMGs are an extension of the Majority Inverter Graphs (MIGs) introduced in [7]. They are more expressive, and therefore more compact, than AIGs or MIGs. This makes them well suited for use in an optimization flow based on exact synthesis, as small representations can be found more quickly. Thus, the use of XMGs reduces the overall runtime.

Fig. 2 shows an XMG representation for a full adder. MAJ nodes are represented as nodes with 3 and 2 outgoing edges, respectively. Note that the arrows of the edges are in the direction of computation.

### V. Method Implementation

#### A. Size Optimization

In our implementation, we have adapted our general optimization method to focus specifically on size optimization for

\(^2\)This number is based on experiments determining the runtime of our algorithm on 4-variable functions. On 6-variable functions the average runtime would likely be higher.
XMGs. Broadly speaking, given an input XMG $N$, our size optimization algorithm consists of the following stages:

1) Area-oriented $k$-LUT mapping of $N$
2) NPN canonization of the functions in the $k$-LUT cover
3) Decomposing the $k$-LUTs into locally optimum XMGs
4) Merging the locally optimum XMGs into an optimized XMG $N'$

These steps are iterated until $N'$ no longer improves.

In the first step of our algorithm, we use our LUT mapper to generate an area-oriented cover. We use the area-flow and exact-area heuristics [28]. The reason for creating a LUT cover is that it turns out to be a superior subnetwork selection strategy as compared to previous approaches. Area-oriented selection using area-flow and exact-area selects a minimal number of LUTs to cover the entire network, using both a global and local view of the network. Thus, LUT mapping is a subnetwork selection strategy that takes both local and global information into account. It is also a good starting point for size minimization. The fewer LUTs (cuts) we need to decompose, the fewer nodes the resulting optimized XMG will have. Finally, by mapping into a minimal number of LUTs, we minimize the number of functions on which we have to invoke our exact synthesis algorithm.

After generating a cover, we extract an optimized XMG. We do so through a topological traversal of the nodes selected in the cover. We compute the NPN canonization of the cut functions, and obtain its optimum XMG. If the optimum XMG is not already present in the database, we compute it and store the results in the NPN class database. Computation of optimum XMGs is done through a generalization of the exact synthesis algorithm proposed in [13]. The pseudocode for this procedure can be found in Algorithm 1.

VI. EXPERIMENTAL EVALUATION

We have integrated the proposed algorithm into our C++ logic synthesis frameworks. The experiments have been carried out Intel E5-2680 CPU with 2.50 GHz with 64 GB of main memory running Linux 3.13.

A. XMG Size Optimization

In this experiment, we compare XMG size optimization to AIG size optimization. Our implementation reads the description of a combinational circuit, reduces the size of the circuit by using the techniques described in Section V-A, and writes back an optimized circuit. We compare our results to those obtained by the state-of-the-art academic logic synthesis package ABC 1.01 [10]. Using ABC, we iteratively apply its resyn2 script until results no longer improve. We measure the size, depth, and switching activity of the resulting optimized networks. The benchmarks are taken from the EPFL benchmark suite, which contains combinational circuits in AIGER format. All results have been formally verified with ABC’s cec command. Table I shows the results.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size Improvement</th>
<th>Depth Improvement</th>
<th>Activity Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMG</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We show the results for our procedure with $k = 4, k = 5$, and $k = 6$. On average, the {size, depth, activity} of XMGs is smaller by {21.7%, 32.1%, 6.1%}, {22.6%, 33.9%, 3.8%}, and {39.4%, 42.2%, 27.7%} for $k = 4$, $k = 5$, and $k = 6$, respectively. Using a $size \cdot depth \cdot activity$ figure of merit, XMG optimization performs 50.1%, 50.8%, and 75.3% better than AIGs for $k = 4$, $k = 5$, and $k = 6$, respectively. We also compute the geometric mean, taken over the sizes, depths, and switching activity of the networks. Both our method and ABC start with the same input networks, containing only AND gates. However, by doing exact synthesis, our method is able to more effectively compress subnetworks, due to the expressive logic primitives in the XMG representation. In other words, our algorithm effectively takes advantage of XMG expressivity. Furthermore, these results confirm our intuition that synthesizing larger subnetworks leads to a better result overall. Higher $k$ lead to better results in logic optimization. Finally, one might suppose that the more expressive XMG primitives are bound to result in smaller representations. However, as the following experiments show, the XMG size optimization advantage also carries over into LUT mapping improvements.

B. LUT Mapping

Our previous experiment compares XMGs to AIGs in a logic optimization context. In order to further investigate the potential of our size optimization method, we evaluate the results after $k$-LUT technology mapping. We compare the results of 6-LUT mapping of the optimized networks from Table I. As the networks are optimized for size, we focus on area-oriented technology mapping. All networks were mapped with ABC, using the command if -a -K 6.

Table II summarizes the results of $k$-LUT technology mapping. Compared to the AIG flow, XMG flow reduces

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

function optimize($N, k$) :=
Input :XMG $N$
Output: Optimized XMG $N'$
$N' \leftarrow N$;

do
$N \leftarrow N'$;
$N' \leftarrow$ new_xmg();
Perform area-oriented mapping of $N$ into $k$-LUTs;
foreach primary input $i$ in $N$ do
create_input($N', i$);
end
foreach LUT $l$ in the cover in topological order do
$f \leftarrow$ function computed by $l$;
$npn \leftarrow$ NPN_canonization($f$);
$opt$ _xmg $\leftarrow$ function_store_get($npn$);
if $opt$ _xmg $\equiv$ nil then
$opt$ _xmg $\leftarrow$ exact_xmg($npn$);
function_store_save($npn, opt$ _xmg);
end
create_node($N', i, opt$ _xmg);
end
while size($N'$) $< size(N)$;
return $N'$;
```
We now turn to a comparison with the best known results for area-oriented 6-LUT mapping. As these have been obtained in various ways, we do not limit ourselves to one type of mapping. We use ABC’s if command to obtain both area-oriented and depth-optimal results. We then collect the best mappings from these and compare them to the best known results. The results can be seen in Table III.

In this paper, we propose a novel algorithm for logic rewriting. Our focus here was on rewriting for size/area. However, with some small modifications, this algorithm could be adapted for depth/delay rewriting as well.

C. Comparison To Best Known Results

The previous experiments show that XMGs compare favorably to AIGs in an optimization and synthesis flow for k-LUTs. We now turn to a comparison with the best known results for these benchmarks. Published alongside the benchmarks of the EPFL benchmark suite, are two sets of best known results. These are the best known 6-LUT covers (for both size and depth) for the benchmarks in the suite. These results may be obtained by any method. As such, the techniques used to obtain these results were not limited one method, but consist of a combination of advanced ABC scripts. In fact, we do not know for each benchmark exactly which method was used to obtain its best known 6-LUT cover. However, all covers have been formally verified. Hence, they serve as a good point of reference.

VII. Discussion on Future Research

Re-writing for depth/delay: In this paper, we propose a novel algorithm for logic rewriting. Our focus here was on rewriting for size/area. However, with some small modifications, this algorithm could be adapted for depth/delay rewriting as well.

Exact synthesis as a service: Exact synthesis results can be shared and computed in the cloud. Although there exist a lot of k-input Boolean functions, we expect that only a small fraction of them occur in practice. Different users can access the same database to query for optimum networks. If
the network has already been computed it can be returned immediately, otherwise, it is scheduled for computation.

Exact synthesis aware mapping: The LUT mapping step in the optimization flow (Fig. 1) decides for which subnetworks optimum representations need to be computed. If only a single of the optimum networks for these functions requires a large amount of runtime, exact synthesis becomes a bottleneck of the optimization. In such cases we could stop the computation and retry with another subnetwork selection strategy. For example, ranking functions by synthesis difficulty could be used to skip those functions that might be a bottleneck.

VIII. CONCLUSIONS

In this paper, we have extended the capabilities of exact synthesis for use in size optimization. We have done so by introducing Boolean function mining, a process which reveals what Boolean functions occur in practice, thus eliminating the need to (pre)compute and store all exact solutions. Additionally, we introduced XOR Majority Graphs: a logic representation that enables smaller networks, and hence faster exact synthesis. These improvements allowed us to design a novel algorithm that meets our goal of scaling up the use of exact synthesis in size optimization. Our size optimization algorithm enables smaller networks, and hence faster exact synthesis. For library binding, ranking functions by synthesis difficulty could be used to skip those functions that might be a bottleneck.

REFERENCES


TABLE III

COMPARING BEST XMGs TO BEST KNOWN 6-LUT MAPPING RESULTS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Best Known Results</th>
<th>XMG Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder</td>
<td>Size</td>
<td>Depth</td>
</tr>
<tr>
<td>Barrel shifter</td>
<td>512</td>
<td>4</td>
</tr>
<tr>
<td>Divisor</td>
<td>3813</td>
<td>1542</td>
</tr>
<tr>
<td>Log2</td>
<td>7344</td>
<td>142</td>
</tr>
<tr>
<td>Max</td>
<td>533</td>
<td>192</td>
</tr>
<tr>
<td>Multiplier</td>
<td>5681</td>
<td>120</td>
</tr>
<tr>
<td>Sine</td>
<td>1347</td>
<td>62</td>
</tr>
<tr>
<td>Square-root</td>
<td>3286</td>
<td>1180</td>
</tr>
<tr>
<td>Square</td>
<td>3800</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>Average:</td>
<td>2946.2</td>
</tr>
<tr>
<td>Geomean:</td>
<td>480</td>
<td>437</td>
</tr>
<tr>
<td></td>
<td>Size · depth:</td>
<td>1125157.9</td>
</tr>
</tbody>
</table>