A Novel Basis for Logic Rewriting

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*—Given a set of logic primitives and a Boolean function, *exact synthesis* finds the optimum representation (e.g., depth or size) of the function in terms of the primitives. Due to its high computational complexity, the use of 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 over 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 run times [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 suite[1] 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 $\mathrm{depth} \cdot \mathrm{size}$ 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.

[1]http://lsi.epfl.ch/benchmarks

The remainder of paper is organized as follows. In Section II we introduce relevant concepts and notation. In Section III we give a high-level technology independent overview of our general optimization method. Section IV introduces XMGs. In Section V we describe our proof-of-concept implementation: a size optimization algorithm based on XMGs. We then evaluate our implementation through several experiments in Section 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 $f(x_1, \dots, x_n)$ 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):

$$x \oplus y = x\bar{y} \vee \bar{x}y = (x \vee y)(\bar{x} \vee \bar{y}) \tag{1}$$

and the majority of three (MAJ):

$$\langle xyz \rangle = xy \vee xz \vee yz = (x \vee y)(x \vee z)(y \vee z). \tag{2}$$

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

$$\langle 0xy \rangle = x \wedge y \quad \text{and} \quad \langle 1xy \rangle = x \vee y \tag{3}$$

The MAJ operation is self-dual [15] since $\langle \bar{x}\bar{y}\bar{z} \rangle = \overline{\langle xyz \rangle}$. 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:

$$a \oplus \langle xyz \rangle = \langle (x \oplus a)(y \oplus a)(z \oplus a) \rangle \tag{4}$$

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:

$$x \oplus y = \bar{x} \oplus \bar{y} = \overline{\bar{x} \oplus y} = \overline{x \oplus \bar{y}} \tag{5}$$

and:

$$\bar{x} \oplus y = x \oplus \bar{y} = \overline{x \oplus y} = \overline{\bar{x} \oplus \bar{y}}. \tag{6}$$

B. NPN Classification

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

$$f(x_1, \dots, x_n) = g^p(x_{\sigma(1)}^{p_1}, \dots, x_{\sigma(n)}^{p_n}), \tag{7}$$

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^{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

N' to improve results. Applying this method with larger k increases the size of the subnetworks that we optimize. Larger k enable better optimization results, on average. To see why, note that in the extreme case k is equal to the number of primary inputs of the network. The result would then be the optimum representation of that network. Hence, we would like to apply this method to the largest possible values for k.

In order to save both storage space and computation time we exploit NPN canonization (see, e.g., [29]). The number of NPN classes is orders of magnitude smaller than the number of functions. Thus, by invoking exact synthesis only for NPN classes, and by storing those results, we compute and store only a small fraction of the total number of functions.

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 run times 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.

[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.

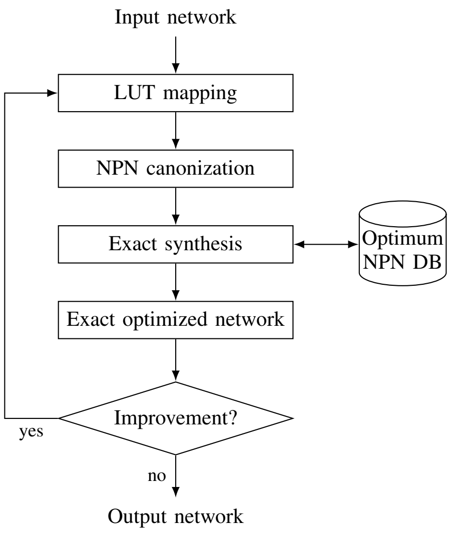

Fig. 1. An overview of the optimization flow.

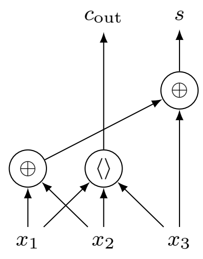

Fig. 2. XMG for a full adder, consisting of 1 majority node and 2 XOR nodes. The XOR node have an $\oplus$ label and the majority node has a $\langle\rangle$ label.

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 and XOR nodes are represented by nodes with 3 and 2 outgoing edges, respectively. Note that the arrows of the edges are inverse to 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

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.

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 $opt\_xmg$ = nil then
            $opt\_xmg \leftarrow$ exact_xmg($npn$);
            function_store_save($npn, opt\_xmg$);
        end
        create_node($N', n, opt\_xmg$);
    end
while size($N'$) < size($N$);
return $N'$;
```

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.

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 $\text{size} \cdot \text{depth} \cdot \text{activity}$ 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

TABLE I
COMPARING XMG AND AIG SIZE OPTIMIZATION

Size Optimization		**XMG** ($k = 4$)			**XMG** ($k = 5$)			**XMG** ($k = 6$)			**AIG**		
Benchmark	I/O	Size	Depth	Activity	Size	Depth	Activity	Size	Depth	Activity	Size	Depth	Activity
Adder	256/129	639	130	888.5	575	131	794.8	383	129	508.9	1019	255	981.7
Barrel shifter	135/128	3281	16	3269.5	2932	18	3294.2	2858	17	3625.4	3141	12	2546.4
Divisor	128/128	29607	4371	21649.5	29607	4371	21649.5	39768	4310	31997.9	40698	4361	31431.1
Hypotenuse	256/128	155349	12507	157711.6	143282	12845	153816.8	99927	9017	100709.1	211262	24670	169609.8
Log2	32/32	27936	275	21862.4	30574	267	26228.5	23006	219	17626.5	29238	375	19046.5
Max	512/130	2296	296	2593.9	2183	258	2569.9	1982	254	2442.3	2831	151	2630.3
Multiplier	128/128	17508	154	17300.7	18771	150	19702.0	16575	136	16675.0	24554	262	20169.1
Sine	24/25	5100	176	3956.0	5419	225	3521.6	3825	121	2620.5	5010	160	3127.6
Square-root	128/64	20130	6031	19456.0	24570	5058	21184.1	17369	6149	17441.92	19437	4968	16977.8
Square	64/128	15070	130	13632.3	15724	132	16021.9	8527	155	8335.0	16568	247	12779.5
Average:		27691.6	2408.6	26232.0	27363.7	2345.5	26878.3	**21422**	**2050.7**	**20198.3**	35375.8	3546.1	27929.9
Geom. Mean:		3555.5			3610.5			**3009.0**			3736.8		

TABLE II
COMPARING 6-LUT MAPPING FOR XMGS AND AIGS

	XMG ($k = 4$)		**XMG** ($k = 5$)		**XMG** ($k = 6$)		**AIG**	
Bench	Size	Depth	Size	Depth	Size	Depth	Size	Depth
Adder	251	131	192	64	250	122	249	121
Bar	888	6	532	5	512	4	512	4
Div	12094	2123	12094	2123	12640	2087	8190	2058
Hyp	50835	7964	52376	8506	48772	8401	47508	8339
Log2	8438	162	8965	152	7961	157	7721	152
Max	745	118	741	121	710	122	771	66
Mult	5700	127	5498	127	5685	126	5689	126
Sine	1655	78	1450	71	1487	76	5615	73
Sqrt	6595	2144	8084	3957	6366	2237	5130	2211
Square	3969	122	3839	121	3930	120	16057	122
Average:	9117.0	**1296.7**	9377.1	1524.7	**8831.3**	1345.2	9744.2	1327.2
Geomean:	904		1055		**851**		1669	
Size · depth:	**11822013.9**		14297264.4		11879864.8		12932502.2	

mapped network size by 6.5%, 3.8%, and 9.4% for $k = 4$, $k = 5$, and $k = 6$, respectively.

Table II also shows two other figures of merit. First, the geometric mean, taken over the sizes and depths of the mapped networks. The geometric means of the XMG mapped networks are lower by 45.8% for $k = 4$, 36.8% lower for $k = 5$, and 48.9% lower for $k = 6$, as compared to the mapped AIGs. Second, it shows the $\text{size} \cdot \text{depth}$ measure. With this measure, XMGs optimized with $k = 4$ show an 8.6% improvement as compared to AIGs, while $k = 6$ gives a 8.1% improvement. This is caused by the fact that the mapping heuristics work out such that $k = 4$ leads to smaller depth. As depth is not our main objective here, we do not consider this to be an issue.

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 [3]. 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.

In this experiment we again map our optimized XMGs to 6-LUTs using ABC. However, we are now comparing against 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.

We show significant improvements on three benchmarks, reducing the {size, area} of the Adder, Multiplier, and Square by {4.5%, 12.4%}, {7.7%, 46.7%}, and {11.5%, 26.8%}, respectively. For most other benchmarks, we are quite close in size, while substantially reducing depth. The main outlier to this trend is the Divisor benchmark. It is not obvious why this benchmark performs so poorly. One interesting observation is that our algorithm appears to work especially well on networks that do addition and multiplication. Networks such as Divisor and Square-root correspond to the inverse of these operations, and our algorithm performs less well on these.

We again calculate the geometric means over the sizes and depths. Our results improve on the mean by 9% as compared to the best results. Using the $\text{size} \cdot \text{depth}$ measure, we show a 10.1% improvement.

VII. DISCUSSION ON FUTURE RESEARCH

Rewriting 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

[3]As of July 15th 2016

TABLE III
COMPARING BEST XMGS TO BEST KNOWN 6-LUT MAPPING RESULTS

	Best Known Results		**XMG Mappings**	
Benchmark	Size	Depth	Size	Depth
Adder	201	73	**192**	**64**
Barrel shifter	512	4	512	4
Divisor	**3813**	1542	10670	**864**
Log2	**7344**	142	7893	**87**
Max	**532**	192	846	**72**
Multiplier	5681	120	**5245**	**64**
Sine	**1347**	62	1488	**48**
Square-root	**3286**	1180	5014	**1032**
Square	3800	116	**3364**	**85**
Average:	**2946.2**	381.2	3914.8	**257.8**
Geomean:	480		**437**	
Size $\cdot$ depth:	1123157.9		**1009151.9**	

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 a 48.9% reduction in the geometric mean, a 9.4% average reduction in size, and a 8.6% reduction in LUT $\mathrm{depth} \cdot \mathrm{size}$, as compared to the state-of-the-art ABC academic tool. It also outperforms 3 over 9 of the best known results for the EPFL benchmark suite, showing reductions of up to 11.5% in size and 46.7% in depth.

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] E. L. Lawler, “An approach to multilevel Boolean minimization,” *J. ACM*, vol. 11, no. 3, pp. 283–295, 1964.

[3] E. S. Davidson, “An algorithm for NAND decomposition under network constraints,” *IEEE Trans. Computers*, vol. 18, no. 12, pp. 1098–1109, 1969.

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

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

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

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

[8] L. Amarù, P.-E. Gaillardon, and G. De Micheli, “Boolean logic optimization in majority-inverter graphs,” in *Design Automation Conference*, 2015, pp. 1–6.

[9] L. Amarù, P.-E. Gaillardon, A. Chattopadhyay, and G. De Micheli, “A sound and complete axiomatization of majority-n logic,” *IEEE Trans. Computers*, 2016.

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

[11] W. Yang, L. Wang, and A. Mishchenko, “Lazy Man’s Logic Synthesis,” in *IWLS*, 2012.

[12] N. Li and E. Dubrova, “AIG rewriting using 5-input cuts,” in *Int’l Conf. on Computer Design*, 2011, pp. 429–430.

[13] M. Soeken, L. G. Amarù, P. Gaillardon, and G. De Micheli, “Optimizing majority-inverter graphs with functional hashing,” in *Design, Automation and Test in Europe*, 2016, pp. 1030–1035.

[14] W. Haaswijk, M. Soeken, L. Amarú, P.-E. Gaillardon, and G. D. Micheli, “LUT Mapping and Optimization for Majority-Inverter Graphs,” in *IWLS*, 2016.

[15] S. B. Akers Jr., “Synthesis of combinational logic using three-input majority gates,” in *Foundations of Computer Science*, 1962, pp. 149–157.

[16] G. Birkhoff and S. A. Kiss, “A ternary operation in distributed lattices,” *Bull. of the Amer. Math. Soc.*, pp. 749–752, 1947.

[17] M. Cohn and R. Lindaman, “Axiomatic majority-decision logic,” *IRE Trans. on Electronic Computers*, vol. 10, pp. 17–21, 1961.

[18] J. R. Isbell, “Median algebra,” *Trans. of the Amer. Math. Soc.*, vol. 260, no. 2, 1980.

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

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

[21] A. Mishchenko, S. Chatterjee, and R. K. Brayton, “Improvements to technology mapping for LUT-based FPGAs,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 240–253, 2007.

[22] J. Cong and Y. Ding, “FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[23] J. Cong and Y. Ding, “On area/depth trade-off in LUT-based FPGA technology mapping,” *IEEE Trans. VLSI Syst.*, vol. 2, no. 2, pp. 137–148, 1994.

[24] J. Cong, C. Wu, and Y. Ding, “Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution,” in *Int’l Symp. on Fied-Programmable Gate Arrays*, 1999, pp. 29–35.

[25] D. Chen and J. Cong, “DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs,” in *Int’l Conf. on Computer-Aided Design*, 2004, pp. 752–759.

[26] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton, “Combinational and sequential mapping with priority cuts,” in *Int’l Conf. on Computer-Aided Design*, 2007, pp. 354–361.

[27] A. H. Farrahi and M. Sarrafzadeh, “Complexity of the lookup-table minimization problem for FPGA technology mapping,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 13, no. 11, pp. 1319–1332, 1994.

[28] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, “Heuristics for area minimization in LUT-based FPGA technology mapping,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 11, pp. 2331–2340, 2006.

[29] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, “Fast Boolean matching based on NPN classification,” in *Int’l Conf. on Field-Programmable Technology*, 2013, pp. 310–313.

[30] L. Benini, P. Vuillod, and G. De Micheli, “Iterative Remapping for Logic Circuits,” *TCAD*, vol. 17, no. 10, pp. 948–964, 1998.

[31] S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan, “Efficient SAT-based Boolean Matching for FPGA Technology Mapping,” in *DAC*, 2006, pp. 466–471.