Multi-level Logic Benchmarks: An Exactness Study

Luca Amarú*, Mathias Soeken†, Winston Haaswijk†, Eleonora Testa†, Patrick Vuillod*,
Jiong Luo*, Pierre-Emmanuel Gaillardon‡, Giovanni De Micheli†

*Synopsys Inc., Design Group, Sunnyvale, California, USA
†Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
‡LNIS, University of Utah, Salt Lake City, Utah, USA

***Abstract*—In this paper, we study exact multi-level logic benchmarks. We refer to an exact logic benchmark, or exact benchmark in short, as the optimal implementation of a given Boolean function, in terms of minimum number of logic levels and/or nodes. Exact benchmarks are of paramount importance to design automation because they allow engineers to test the efficiency of heuristic techniques used in practice. When dealing with two-level logic circuits, tools to generate exact benchmarks are available, e.g., *espresso-exact*, and scale up to relatively large size. However, when moving to modern multi-level logic circuits, the problem of deriving exact benchmarks is inherently more complex. Indeed, few solutions are known. In this paper, we present a scalable method to generate exact multi-level benchmarks with the optimum, or provably close to the optimum, number of logic levels. Our technique involves concepts from graph theory and joint support decomposition. Experimental results show an asymptotic exponential gap between state-of-the-art synthesis techniques and our exact results. Our findings underline the need for strong new research in logic synthesis.**

I. Introduction

The *Electronic Design Automation* (EDA) community heavily relies on logic benchmarks to evaluate the performance of academic and commercial design tools [1]–[5]. Benchmarking of practical (heuristic) design methods in EDA involves comparing current results with immediately previous results or best attained results [5]. In this way, it is possible to measure the *relative efficiency* of a heuristic technique. As it is unknown whether the existing results are optimum, an *absolute* measure cannot be provided. Consequently, although the relative efficiency may be large, the results can still be far from the optimum. A better evaluation is possible if the exact results are also available. With the exact results, it is possible to determine the *absolute efficiency* of a heuristic technique by measuring how close it gets to the known optimum. However, generating (meaningful) benchmarks with known exact results is quite a challenging problem [6]. This is especially true for logic synthesis and logic optimization [6].

When considering only two-level logic, it is possible to generate fairly large exact circuits. Tools such as *espresso-exact* [8], based on the Quine-McCluskey algorithm, are actively used in design flows and solve the problem of two-level logic minimization exactly.

When moving to modern multi-level logic, the problem of exact synthesis becomes inherently more difficult [6]–[8]. This is because the space of possible solutions is much larger as there is no limitations on the number of logic levels. Few solutions exist to solve the problem of exact multi-level synthesis [9]. Among the few existing solutions, none of them attained the same performance as exact two-level synthesis, limiting the possible applications of the corresponding tools. For this reason, it is interesting to generate benchmarks that are provably optimum, against which heuristic approaches can be compared.

Exact multi-level benchmarks can be generated in two ways: (i) by using exact synthesis algorithms on a function specification, and (ii) by providing a construction algorithm, with relaxed or no function specification, and prove that it always generates an optimum circuit. The weak scalability of exact synthesis algorithm rules out the first approach to generate large exact benchmarks.

In this paper, we address the problem of creating exact multi-level benchmarks using a constructive approach. Our construction generates *non-trivial*, non-treelike, depth-optimal polynomial size multi-level circuits in polynomial time. The key concepts enabling this result come from graph theory and joint support decomposition. Experimental results show an asymptotic exponential gap between state-of-the-art synthesis techniques and our exact results. Our exact results, publicly available at [10], serve as common yardstick for future synthesis work and open the next big challenges in logic synthesis. In fact, our findings demonstrate the enduring need for efficient and effective logic synthesis tools.

The remainder is organized as follows. Section II surveys previous works on logic benchmarks, with an emphasis on exact circuits. Section III describes the proposed construction techniques for exact multi-level circuits with optimal depth. Section IV shows synthesis experiments on our set of depth-optimal benchmarks. Section V discusses open challenges and future work on exact benchmarks and exact synthesis. Section VI concludes the paper.

II. Background and Motivation

A. Previous Benchmark Suites

1) MCNC Benchmarks: The first set of combinational benchmark circuits was reported at the *International Symposium on Circuits and Systems* (ISCAS) in 1985. After four

years, sequential circuits were added to ISCAS'85 generating the ISCAS'89 benchmark suite. In 1991, these benchmarks plus others presented at past workshops and conferences were collected and distributed under the maintainance of the *Microelectronics Center of North Carolina* (MCNC) [1]. The MCNC suite was published in the same year at the *International Workshop on Logic Synthesis* (IWLS). Even though quite outdated, MCNC benchmarks are still popular in academic research.

2) IWLS Benchmarks: In 2005, a new set of benchmarks for logic synthesis was presented at the IWLS workshop under the name of IWLS05 benchmark suite [2]. It consisted of 84 designs collected from various websites (OpenCores, Faraday, etc.) and previous benchmark suites (MCNC, ITC, etc.).

3) EPFL Benchmarks: The EPFL combinational benchmark suite has been introduced at the IWLS workshop in 2015 [3]. It consists of 23 combinational circuits designed to challenge modern logic optimization tools. The benchmark suite is divided into arithmetic, random/control, and MtM (*more-then-ten-million*) benchmarks. The arithmetic part includes 10 benchmarks, e.g., square-root, hypotenuse, divisor, and multiplier. The random/control part consists of another 10 benchmarks, e.g., round-robin arbiter, lookahead XY router, alu control unit, and memory controller. The MtM part contains 3 very large benchmarks, featuring more than ten million gates each. In addition to providing the benchmarks, the EPFL suite also keeps track of the best optimization results.

B. Exact-Size Benchmarks

It is unknown whether any benchmark of the previously discussed benchmark suits is optimum—in fact, for most of the benchmarks it is quite unlikely. Other benchmark suites have been presented which only contain exact benchmarks that are optimum with respect to size.

1) LEKO Benchmarks: Logic synthesis Examples with Known Optimal (LEKO) have been introduced in [11] with application to FPGA synthesis. They target area-optimal mappings so they can be classified as exact-size benchmarks. The core idea of LEKO is to replicate a small circuit with known optimal size. If the replication follows a specific strategy [11], the final results preserve size-optimality. Size-optimality is measured in terms of the number of 4-LUTs rather than number of binary operations.

2) LEKU Benchmarks: Logic synthesis Examples with Known Upper Bounds (LEKU) are derived from LEKO by collapsing them into two level logic and successively decomposing them into primitive gates. LEKU circuits serve as suboptimal starting points for the heuristic techniques under test.

C. Motivation

In this work, we are addressing the construction of exact-depth benchmarks, which has not been proposed so far. These benchmarks are of particular interest to today's design flows that often optimize for depth as it correlates with the delay of the final system.

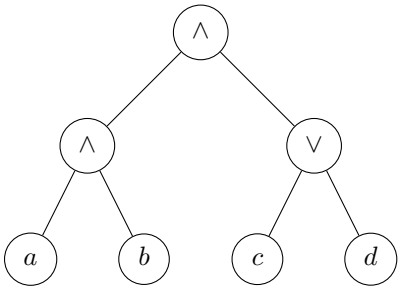

Fig. 1. Depth-optimal balanced-tree realization for $f = abc + abd$.

III. Construction of Depth-Optimal Benchmarks

In this section we present a construction method for multi-level logic circuits with optimal depth. Our construction is based on balanced binary tree circuits, i.e., circuit in which each node that is not a leaf has two children and every path from the root to a leaf has the same length. We further assume that each leaf represents a unique variable. In Section III.A, we show that balanced binary tree circuits with unique inputs are depth-optimal. However, the simple construction has two major drawbacks: (i) there are assignments of binary functions to the nodes that lead to trivial functions, and (ii) each subcircuit is disjoint support decomposable, which makes them easy to optimize using disjoint support decomposition techniques [15], [16]. In Sections III-B and III-C we address these drawbacks with more advanced algorithms.

A. Depth-Optimality of Balanced Binary Tree Circuits

As binary operations for the nodes in balanced binary tree circuits we consider the 10 binary Boolean operators of two variables[1] $a \wedge b$, $\bar{a} \wedge b$, $a \wedge \bar{b}$, $\bar{a} \wedge \bar{b}$, $a \vee b$, $\bar{a} \vee b$, $a \vee \bar{b}$, $\bar{a} \vee \bar{b}$, $a \oplus b$, and $\bar{a} \oplus b$. Recall, that leaves are labeled by a unique variable.

Theorem 3.1: A balanced binary tree as described above is a depth-optimal realization for the Boolean function it realizes.

Proof: (*Reductio ad absurdum*) It is easy to see that a function represented by a balanced binary tree of depth k depends on all 2^k variables obtained from the tree's leaves. Let us assume it can be realized by a circuit of depth $k-1$. A circuit of depth $k-1$ in which each node represents one of the 10 Boolean operators can have at most 2^{k-1} unique leaves and hence depend at most on 2^{k-1} variables which contradicts our assumption. ■

Fig. 1 shows an example for a depth-optimal realization using a balanced binary tree structure. The represented function is $f = abc + abd$.

Although balanced binary trees are provably depth-optimal, sometimes they represent trivial functions. For example, if all

[1]There are 16 possible binary Boolean operators, but operators 0, 1, a, $\bar{a}$, b, $\bar{b}$ do not functionally depend on both variables.

nodes represent AND operations, the tree simply represents a multi-input AND function. Most EDA tools would easily derive such representation. Similar considerations hold for the OR operation. In this work, we are interested in non-trivial depth-optimal multi-level circuits.

B. Populating the Binary Tree with Operators

In order to avoid trivial functions, we suggest to just pick them randomly using a linear distribution over the 10 binary operations. Since 2 of the 10 operations are binate, there is a very high probability to avoid circuits that represent unate functions [12], which again can be handled as a special case by logic synthesis algorithms [13], [14]. In order to completely rule out the possibility of generating unate circuits, it is sufficient to enforce the presence of at least one binate operator in the tree. This can happen right after the random assignment. Algorithm 1 shows pseudo code to generate circuits in such a manner.

Algorithm 1 Generation of depth-optimal multi-level circuits with disjoint support.

INPUT: Complexity measure n
OUTPUT: Depth-optimal circuit with 2^n inputs.

```
create empty balanced binary tree with n levels;
for each node n do
    assign n a random binary operator;
end for
enforce the presence of at least one binate operator;
```

C. Breaking the Disjoint Support Property

The circuits generated by Algorithm 1 have optimal depth and are binate. However, there is still an important property potentially making these benchmarks *trivial*. At any node of the tree, the function represented has disjoint support [15]. This means that disjoint support decompositon techniques [15], [16] can be quite effective here. In order to overcome this limitation, in this section we show how to break the disjoint support property.

We present two techniques to break the disjoint support property. The first one merges two disjoint full trees. The second one uses non-disjoint building blocks in place of binary operators.

1) Merging Two Disjoint Full Trees: This method starts with the creation of two balanced binary trees using Algorithm 1. These two trees have the same number of primary inputs but they need to be functionally different. Since the generation of Algorithm 1 is random, it is very unlikely that the two circuits represent the same function, especially for large n. As a second step, the primary inputs are shared between the two circuits. The roots of the two subtrees are then combined with a random binary operator into a single primary output. Finally, the functional support of the composite circuit is verified. Algorithm 2 briefly depicts the procedure.

Algorithm 2 Generation of depth-optimal multi-level circuits with joint support.

INPUT: Complexity measure n
OUTPUT: Depth-optimal circuit with 2^n inputs.

```
A = Algorithm 1(n);
B = Algorithm 1(n);
share primary inputs of A and B;
create node n that joins roots of A and B;
assign node n to a random binary operator;
set n as primary output of the composite circuit;
verify the functional support of the composite circuit;
```

Algorithm 2 generates a multi-level circuit with 2^n inputs, $n+1$ levels and $2^{n+1}-1$ nodes. The dependency on all 2^n inputs is not guaranteed by the construction method and therefore we verify the functional support as a last step. Although verifying the functional support is intractable, modern SAT-based techniques can handle fairly large problem instances [17]. It is worth noticing that in all our experiments the functional support has not been altered by Algorithm 2.

With the functional support verified, we can determine the properties of the benchmarks generated by Algorithm 2. The following theorem holds.

Theorem 3.2: A multi-level logic circuit generated by Algorithm 2, with functional support verified and equal to 2^n, is (i) binate and non-fully disjoint support and (ii) at most one unit far from the optimum depth.

Proof We start by proving the binate and non-full disjoint support properties. Binateness follows from Algorithm 1. The non-disjointness follows by merging of two circuits with shared primary inputs: at least one node will not admit a disjoint support decomposition [16]. Now consider the exactness property: the circuit is at most one level far from the optimum. This means the circuit cannot be realized with less than n levels. Let us proceed by contradiction and suppose instead it is possible to build the same circuit with $n-1$ levels. With similar reasoning as in the proof of Theorem 3.1, we can show that such circuit would depend on fewer inputs than the original one, hence the contradiction.

The circuits generated by Algorithm 2 can be simplified by low-effort synthesis scripts. By doing this it is possible to recover the level off and hit the absolute optimum. Also, the joint support nature of the circuit (reconvergence) is more visible once processed by a synthesis pass. Fig. 2 shows the output of Algorithm 2, with $n = 4$, after a simple AIG rewriting pass [18], [19]. The depth optimality is reached.

2) Using non-DSD Blocks as Primitive Operators: A different way of addressing the disjoint support issue, is to use larger primitive blocks in the tree construction. These large primitive blocks would be optimal implementations of function with no disjoint support decomposition (*prime*) [15], [16]. Algorithm 1 can be easily adapted for this purpose. However,

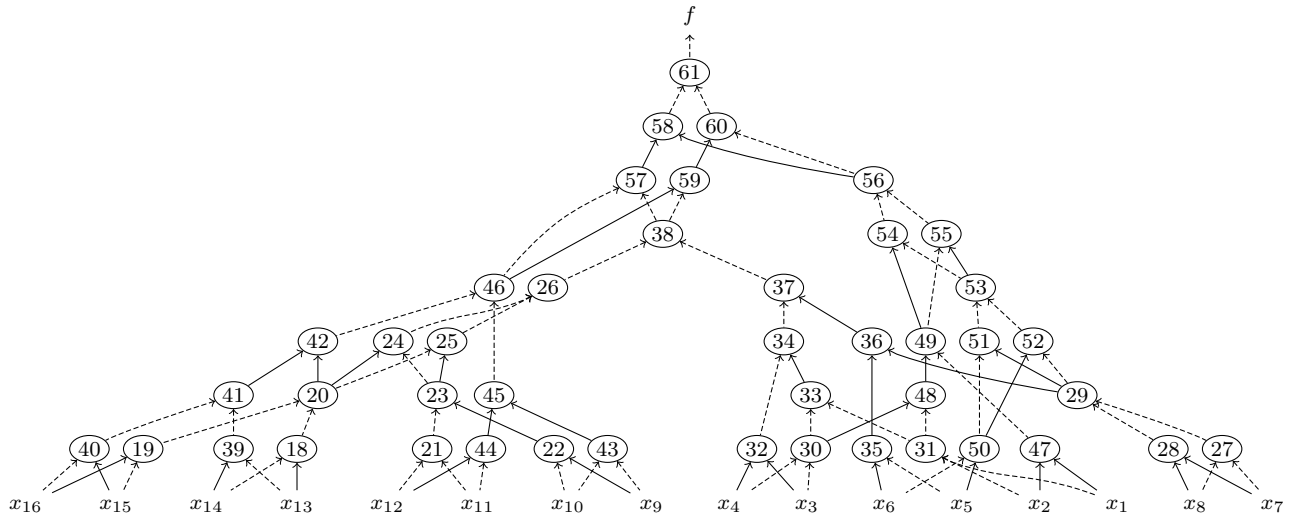

Fig. 2. Exact benchmark with complexity 4 (2^4 inputs). Depth-optimal AIG realization with 8 levels. Nodes are AND-2 functions and dotted edges are INV.

the problem of using k-arity blocks is that the number of leaves, i.e., primary inputs, of the tree grows much faster. As an example, when using primitive blocks with 4 inputs, the number of primary inputs grows as 4^n, with n being the complexity (depth) of the benchmark considered.

In this work, we use Algorithm 2 to generate depth-optimal multi-level benchmarks.

D. Building a Suboptimal Circuit

Once the depth-optimal benchmark has been generated, we also need to create a suboptimal starting point for the synthesis tools under test. We discuss hereafter two possible approaches to generate such suboptimal circuit.

1) BDD-based Collapsing: One possibility is to create a *Binary Decision Diagram* (BDD) [20] for the exact benchmark. Since the functional support of the benchmark has been verified, the corresponding BDD is guaranteed to have as many levels as many inputs. The circuits generated by Algorithm 2, with $n + 1$ levels but 2^n inputs, would be collapsed into 2^n levels. This guarantees an exponential gap between the exact solution and suboptimal starting point. In this way, the logic synthesis tools under test have an ample design space to explore, making the benchmarking more effective. The only drawback stands in the BDD complexity. It is known that BDD may be exponentially sized [20] and they only scale up to relatively small/medium values. Our experimental evaluation has shown this approach is viable up to 1024/2048 inputs ($n = 10/11$). After that, BDD collapsing becomes too expensive in terms of runtime and alternative approaches are required. Please note that circuits with up to 2048 inputs and 11 levels are already relatively large and relevant to most contemporary synthesis scenarios.

2) SOP-based Collapsing: SOP-based collapsing is quite popular in traditional logic synthesis and consists in generating a two-level SOP for the circuit under test. Recent approaches for SOP collapsing make use of SAT engines [21] and showed quite efficient results. In our context, SOP collapsing must be succeded by a decomposition step, in order to create a suboptimal multi-level circuit. SOP collapsing also has scalability limitations, mainly related to the two-level logic representation.

In this study, we use BDD-based collapsing to generate the suboptimal starting point.

IV. Experimental Results

In this section, we test state-of-the-art synthesis techniques over a set of exact benchmarks. We first describe the experimental methodology. Then, we present the synthesis results.

A. Methodology

Our algorithms have been implemented in C programming language and the tools have been tested in a Linux environment. Our C progam counts about 1k lines.

1) Benchmark Generation: We generated 7 exact benchmarks, using Algorithm 2, with complexity ranging from 4 to 10. The corresponding circuits have support size ranging from 16 to 1024. The runtime for generating the optimal benchmarks is negligible, less than 4 seconds in the worst case. The suboptimal starting implementations have been created with BDD-based collapsing, as explained in the previous section. The BDD for the exact circuit with complexity 10 (1024 inputs) has more than 600 000 nodes. Even though it is still possible to collapse exact benchmarks with complexities 11 and 12, their size would be rather out of scale and would bias the comparison with previous synthesis results.

2) Synthesis Setup: We considered three state-of-the-art synthesis techniques, namely AIG-optimization performed by ABC [19], MIG-optimization performed by MIGhty [23]–[25],

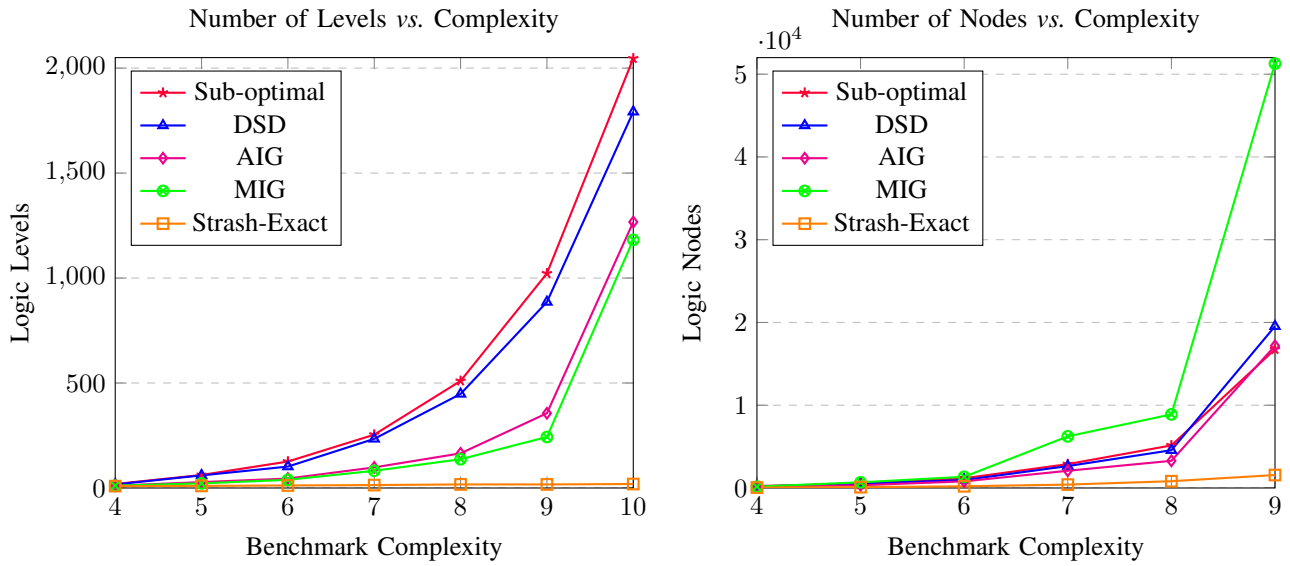

Fig. 3. Synthesis experiments for the exact depth benchmarks. Benchmark complexity of 10 is not reported in the size plot because the value is out of scale.

TABLE I
SYNTHESIS EXPERIMENTS: LOGIC DEPTH

Complexity	Sub-opt.	DSD	AIG	MIG	St-Exact
4	19	17	11	11	**9**
5	62	60	28	22	**10**
6	126	102	45	39	**12**
7	254	234	98	82	**14**
8	510	448	165	137	**17**
9	1022	886	356	243	**17**
10	2046	1792	1267	1182	**19**

TABLE II
SYNTHESIS EXPERIMENTS: LOGIC SIZE

Complexity	Sub-opt.	DSD	AIG	MIG	St-Exact
4	235	175	154	164	**51**
5	524	517	331	686	**97**
6	1176	961	801	1351	**204**
7	2881	2648	2081	6236	**411**
8	5145	4563	3218	8893	**818**
9	16752	19541	17184	51278	**1564**
10	634806	468735	269432	275052	**3110**

and DSD-optimization performed by ABC [16]. Since our study targets depth-minimality, we considered depth-oriented synthesis scripts. For AIG-optimization, we used the script *resyn2rs; if -g* iterated 10 times [22]. For MIG-optimization, we used the synthesis script described in [25]. For DSD-optimization, we used the *dsd* command available in ABC. On top of these three techniques, we also report the sub-optimal results and the exact results strashed into AIGs. These *strashed-exact* results serve as upper bounds on the optimum.

Please note that, in this study, we do not aim at comparing existing synthesis tools against each other. We are rather interested in determining the gap between standard synthesis results and the exact results. For this reason, we do not optimize further the synthesis scripts but we use their default versions. Although we do not provide an extensive evaluation of all contemporary synthesis tools, our work enables other researchers to pursue this direction.

B. Results

The synthesis results are reported in Tables I and II and graphically shown by Fig. 3.

1) Depth results: As a first observation, the depth of all DSD-, AIG-, and MIG-optimized circuits sensibly decreases w.r.t. the suboptimal implementations. DSD-optimization is less effective than AIG and MIG. This is expected because the exact benchmarks have been explictly designed to have joint support. This implies that disjoint support decomposition will not find many optimization opportunities. Considering instead AIG and MIG optimizations, they both strongly reduce the number of levels, between 2× and 3×, as compared to the suboptimal starting points. However, with respect to to the strashed exact realizations, all three synthesis techniques are far from achieving the optimum. For the first benchmarks, with complexity 4–6, AIG and MIG heuristics are between 1.2× and 3.6× far from the exact results. When moving to higher complexity, the gap between the exact results and the synthesis heuristics becomes larger, up to 66×. In Fig. 3 it is possible to appreciate the trend of this gap, which resembles an exponential curve.

2) Size results: Table II shows the size results for the considered synthesis techniques. DSD and AIG techniques reduce the number of nodes for most benchmarks. The MIG technique increases the number of nodes as the depth-optimization is more aggressive. Please note that more size optimization is possible if interleaving MIG depth optimization [25] with MIG size techniques in [26]. This would result in a similar script as the one used for AIG optimization. However, as the purpose of this work is not size optimization, we did not improve further

the size results.

For all techniques, the size of the exact results remains much smaller, from about 3× to 90×. In Fig. 3, the size for complexity 10 is not reported because out of scale and would compromise the readability of the plot. Nevertheless, all the values are included in Table III for the sake of completeness.

V. Discussion

Our benchmark generation methods provide, for the first time, a scalable way to generate *non-trivial* exact circuits in the multi-level realm. This opens the opportunity to measure the efficienty of multi-level optimization heuristics with an *absolute* metric.

This section briefly discusses some limitations of our methods and outlines future research.

A. Scalable Collapsing

Collapsing an exact benchmark is key to create the suboptimal circuit for the tools under test. In this work, we employed BDD collapsing for this purpose. However, the use of BDDs precludes the scalability to *very large* benchmarks. A hybrid BDD [20] and SAT [21] collapsing approach can alleviate the issue. A fundamentally different approach would be to use logic maximization techniques rather than logic minimization techniques. In our context, we would look for depth-increasing transformations. Rewriting techniques [18], [26] can have their internal goal easily modified for this purpose. While this approach can be more scalable, the corresponding suboptimal circuits have weaker properties w.r.t. BDD collapsing.

B. Non-DSD Native Construction

As mentioned in Section III-C2, using non-DSD primitive blocks is an alternative approach to generate exact benchmarks. By choosing the right non-DSD (*prime*) function block, it is possible to tune the tree n-arity *vs.* benchmark complexity. A clear advantage of non-DSD native construction is that the functional support verification is not anymore necessary. This is because the original properties of the circuit-tree are preserved and merging is not required to break the DSD feature. We believe that this approach would pave the way for *very large* exact benchmarks, where verifying the functional support may be the runtime bottleneck.

VI. Conclusions

In this paper, we studied exact multi-level logic benchmarks. We presented efficient methods to generate exact multi-level benchmarks with optimum, or provably close to the optimum, number of logic levels. The exact benchmarks generated by our techniques are *non-trivial* and scale up to large size. The key concepts enabling our results come from graph theory and joint support decomposition. Experimental results showed an asymptotic exponential gap between state-of-the-art synthesis techniques and our exact results. Our exact results, publicly available at [10], will serve as common yardstick for future synthesis work and prove that logic synthesis is a field where innovation is still possible.

References

[1] S. Yang, *"Logic Synthesis and Optimization Benchmarks, Version 3.0"*, Tech. Report, Microelectronics Center of North Carolina, 1991.

[2] IWLS 2005 Benchmarks. http://iwls.org/iwls2005/benchmarks.html

[3] L. Amaru, P.-E. Gaillardon, G. De Micheli, *"The EPFL Combinational Benchmark Suite"* International Workshop on Logic & Synthesis (IWLS), 2015.

[4] P. Verplaetse, J. Campenhout, D. Stroobandt, *"On synthetic benchmark generation methods"*, International Symposium on Circuits and Systems (ISCAS), 2000.

[5] A. Ng, I. L. Markov, *"Toward quality EDA tools and tool flows through high-performance computing"*, International Symposium on Quality Electronic Design (ISQED). IEEE, 2005.

[6] D. Knuth, *"The Art of Computer Programming"*, Volume 4A, Part 1

[7] K. Keutzer, D. Richards, *"Computational complexity of logic synthesis and optimization"*, International Workshop on Logic & Synthesis (IWLS), 1989.

[8] O. Coudert, T. Sasao. *"Two-level logic minimization"*, Logic Synthesis and Verification. Springer US, 2002. 1-27.

[9] E. Ernst, *"Optimal combinational multi-level logic synthesis"*, PhD Disseration, Univ. of Michigan, 2009.

[10] Exact benchmarks section, available for download at: http://lsi.epfl.ch/benchmarks

[11] J. Cong, K. Minkovich, *"Optimality study of logic synthesis for LUT-based FPGAs"* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26.2 (2007): 230-239.

[12] N. Alon, R. Boppana, *"The monotone circuit complexity of Boolean functions"*, Combinatorica 7.1 (1987): 1-22.

[13] T. Thorp, G. Yee, C. Sechen, *"Design and synthesis of monotonic circuits"* International Conference on Computer Design (ICCD), 1999.

[14] G. De Micheli, *"Synthesis and optimization of digital circuits"*, McGraw-Hill Higher Education, 1994.

[15] V. Bertacco, M. Damiani, *"Disjunctive decomposition of logic functions"*, International Conference On Computer Aided Design (ICCAD), 1997.

[16] A. Mishchenko, R. Brayton, *"Faster logic manipulation for large designs"*, International Workshop on Logic & Synthesis (IWLS), 2013.

[17] M. Soeken, P. Raiola, B. Sterin, M. Sauer, *"SAT-based Functional Dependency Computation"*, International Workshop on Logic & Synthesis (IWLS), 2016.

[18] A. Mishchenko, S. Chatterjee, R. Brayton, *"DAG-aware AIG rewriting a fresh look at combinational logic synthesis"*, Design Automation Conference (DAC), 2006.

[19] ABC synthesis tool - available online.

[20] Randal E. Bryant. *"Graph-Based Algorithms for Boolean Function Manipulation"*. IEEE Transactions on Computers, C-35(8):677691, 1986.

[21] A. Petkovska, A. Mishchenko, D. Novo, M. Owaida, P. Ienne, *"Progressive generation of canonical sums of products using a SAT solver"*, International Workshop on Logic & Synthesis (IWLS), 2016.

[22] A. Mishchenko, R. Brayton, S. Jang, V. Kravets, *"Delay optimization using SOP balancing"*, International Conference On Computer Aided Design (ICCAD), 2011.

[23] L. Amaru, P.-E. Gaillardon, G. De Micheli, *"Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization"*, Design Automation Conference (DAC), 2014.

[24] L. Amaru, P.-E. Gaillardon, G. De Micheli, *"Boolean Logic Optimization in Majority-Inverter Graphs"*, Design Automation Conference (DAC), 2015.

[25] L. Amaru, P.-E. Gaillardon, G. De Micheli, *"Majority-Inverter Graphs: A New Paradigm for Logic Optimization"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 35.5, 2015.

[26] M. Soeken, L. Amaru, P.-E. Gaillardon, G. De Micheli, *"Optimizing Majority-Inverter Graphs With Functional Hashing"*, Design, Automation & Test in Europe Conference (DATE), 2016.