A Scalable Mixed Synthesis Framework for Heterogeneous Networks

Max Austin[1], Scott Temple[1], Walter Lau Neto[1], Luca Amarù[2], Xifan Tang[1], Pierre-Emmanuel Gaillardon[1]
[1]LNIS, University of Utah, Salt Lake City, Utah, USA
[2]Synopsys Inc., Design Group, Sunnyvale, California, USA

Abstract— We present a new logic synthesis framework which produces efficient post-technology mapped results on heterogeneous networks containing a mix of different types of logic. This framework accomplishes this by breaking down the circuit into sections using a hypergraph *k-way* partitioner and then determines the best-fit logic representation for each partition between two Boolean networks, *And-Inverter Graphs* (AIG) and *Majority-Inverter Graphs* (MIG), which have been shown to perform better over each other on different types of logic. Experimental results show that over a set of *Open Piton Design Benchmarks* (OPDB) and OpenCores benchmarks, our proposed methodology outperforms state-of-the-art academic tools in *Area-Delay Product* (ADP), *Power-Delay Product* (PDP), and *Energy-Delay Product* (EDP) by 5%, 2%, and 15% respectively after performing *Application Specific Integrated Circuits* (ASIC) technology mapping as well as showing a 54% improvement in runtime over conventional MIG optimization.

I. Introduction

Logic synthesis refers to the process producing a compact and fast gate-level implementation from a *Register-Transfer-Level* (RTL) design description and is a crucial step in *Electronic Design Automation* (EDA) tools [1]. In general, this process is dependent on circuit structure, and finding the best approach to optimization is difficult without years of experience. Furthermore, many technologies today are heterogeneous meaning a single approach may not yield the best results for each block. Therefore, an optimization approach that considers multiple types of logic simplification strategies is necessary to solve this problem.

This paper presents *LSOracle*, a novel logic synthesis framework that implements mixed logic optimization using both *And-Inverter Graph* (AIG) and *Majority-Inverter Graph* (MIG) logic representations to achieve better area and delay optimization results. Our main contribution is a three-stage logic optimization flow consisting of (i) partitioning the network (ii) determining the best-fit representation for each partition between either AIGs or MIGs, and (iii) optimizing these partitions and merging them back. An added benefit for this approach is that it opens up the possibility to integrate parallelism to offset the higher computational cost of partitioning and using multiple representations.

With the proposed framework, over a set of Open Piton Design Benchmarks (OPDB) [2] and OpenCores benchmarks [3], we perform two sets of evaluation: (i) technology independent and (ii) post technology mapping. After technology independent optimization, we see an average improvement in the area and delay when compared to the original, pre-optimized benchmarks of about 17% and 41% respectively demonstrating that we are leveraging the benefits of using a mix of AIGs and MIG. We see significant improvements in *Energy-Delay Product* (EDP), *Power-Delay Product* (PDP), and *Area-Delay Product* (ADP) after *Application Specific Integrated Circuits* (ASIC) technology-mapping using the ASAP 7nm standard-cell library [4]. When comparing our framework against the state-of-the-art tool *ABC* [5], we can see improvements on average of 5% in ADP, 2% in PDP, and 15% in EDP. The improvements for PDP and EDP are similar to those achieved with the MIG-based optimization using the tool *CirKit* [6] but when comparing runtime, *LSOracle* shows, on average, a $6.76\times$ speedup.

The remainder of this paper is organized as follows: Section II gives an overview of the different Boolean networks being considered and the motivation of mixed synthesis. Section III discusses how these topics are implemented in our framework. Section IV presents and analyzes the experimental results and Section V concludes this work.

Acknowledgement: The work is funded by DARPA, under the grant FA8650-18-2-7849. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory and Defense Advanced Research Projects Agency or the U.S. Government.

II. Background

This section presents the basic concepts used throughout this work. We first briefly review some widely used Boolean networks and then discuss the motivation behind using multiple logic synthesis flows for optimization.

A. Boolean Networks

Boolean networks are *Directed Acyclic Graphs* (DAGs) where each internal vertex corresponds to a Boolean function (logic gates), and the edges represent connections between these functions. Also, vertices may represent *Primary Inputs* (PI), *Primary Outputs* (PO), registers, and constants. Edges may be regular or inverted, representing the *NOT* operator. The edge direction is analogous to the data flow, *i.e.,* from PIs to POs. When internal nodes represent the same logic function and have the same number of *fanin*, the DAG is defined as homogeneous. Homogeneous DAGs are simple and enable efficient logic manipulation [1]. The outputs of a node may be connected to other nodes, being called the node *fanout*. On the other hand, the inputs of a node are known as its *fanin*. In a DAG, the *fanin* (*fanout*) cone of a node n is the set of all nodes reachable through the *fanin* (*fanout*) of such a node. This work focuses specifically on AIGs and MIGs where the nodes of the DAG represent the 2-input AND function and the 3-input Majority function respectively and edges can be either regular or complemented. In general control blocks are likely optimized by AIG-based techniques, whereas arithmetic logic is likely optimized by MIG-based techniques, which have recently proven to be powerful in arithmetic logic [7]. More details on these Boolean networks can be found in [5] and [8].

B. Mixed Synthesis Motivation

A mixed logic synthesis approach, named *MIXSyn*, has been recently introduced in [9]. *MIXSyn*'s motivation relies on the

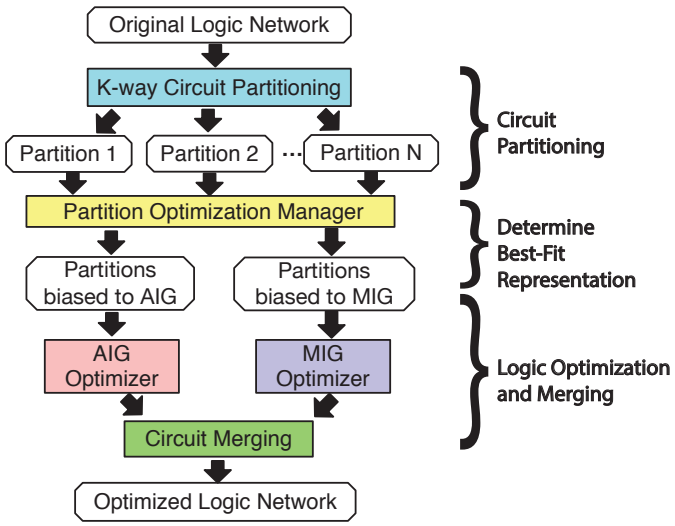

Fig. 1. Proposed logic synthesis flow

fact that efficient methods exist for two important classes of functions, *i.e.,* AND/OR and XOR, but usually, just one method is applied, leading to suboptimal results for the other class. *MIXSyn* AND/OR optimization is carried by ABC, whereas XOR decomposition and optimization is handled by BDS-pga [10]. While *MIXSyn* presents encouraging results, it does not present an integrated framework which handles different types of logic. It also relies on *Binary Decision Diagrams* (BDDs) for XOR optimization, which tends to be less scalable than DAGs. Therefore, in this work, we propose an integrated framework capable of efficiently handling different logic blocks with the appropriate DAG. The idea is that modern integrated circuits are likely composed of many *Intellectual Property* (IP) blocks, that have different logic attributes. Therefore, the right optimizer is used according to the logic performed by each block.

III. Proposed Framework: *LSOracle*

This section describes our proposed logic synthesis framework. As depicted in Fig. 1, our framework consists of three stages: (i) **circuit partitioning**, (ii) **determining the best-fit logic representation for optimization** and (iii) **logic optimization and merging**. The pseudocode for this mixed synthesis framework is shown in Alg. 1. In the remainder of this section, we will detail the technicality of each stage.

A. Circuit Partitioning (Alg.1-α)

The initial step of *LSOracle* is network partitioning to separate complex logic into blocks that can be manipulated individually with the best-fit method. This partitioning brings benefits in two areas: (i) it ensures minimal dependency in partitions to minimize the loss in global optimization and (ii) it easily allows a parallel infrastructure for optimization. The partitioning methodology used in *LSOracle* takes a hypergraph as its input. An edge in a typical undirected or directed graph connects only two vertices. A hyperedge in a hypergraph, however, can connect multiple vertices making it ideal for representing circuit netlists [11]. This also makes homogeneous Boolean networks [e.g. AIG] preferable as input since DAGs give a clear understanding of the size and structure of the network as well as logic attributes. Each node in the DAG is assigned to a single partition, and this information is stored in a partition manager. This manager contains all of the partition information and also allows efficient viewing and manipulation of individual partitions.

Algorithm 1 Mixed Logic Synthesis

```
   Input: AIG network (AIG)
          Number of Partitions (num_parts)
   Output: MIG Network (MIG)
1: partitions ← KWayPartitioning(num_parts)                ▷ (α)
2:
3: #pragma omp parallel for
4: for each part in partitions do                          ▷ (γ)
5:     mig_opt ← MIG_optimization(part)
6:     aig_opt ← AIG_optimization(part)
7:     compare mig_opt and aig_opt node×depth
8:     if mig_opt best-fit algorithm then
9:         add part to mig_parts
10:    else
11:        add part to aig_parts
12:    end if
13: end for
14: #pragma omp parallel for
15: for each part in aig_parts do                          ▷ (δ)
16:    part_opt ← AIG_optimization(part)
17:    synchronize_network(AIG, part_opt)
18: end for
19: convert AIG to MIG
20: #pragma omp parallel for
21: for each part in mig_parts do                          ▷ (δ)
22:    part_opt ← MIG_optimization(part)
23:    synchronize_network(MIG, part_opt)
24: end for
```

B. Determine Best-Fit Representation (Alg.1-γ)

After circuit/network partitioning, *LSOracle* applies logic optimization and determines the logic representation that leads to the best results for each partition. In this work, we consider two state-of-the-art methods: an AIG-based and a MIG-based logic optimization recipe [5; 8]. Note that even though we are considering just these two methods in this work, our method is rather general and can support a large portfolio of logic optimizers such as XAG/XMG, BDS, etc. [9; 12]. To ensure a fair comparison between AIG and MIG optimizations, the same recipe of interleaved area and depth algorithms were used. To determine the best-fit recipe to use for each partition, we apply a heuristic that determines which algorithm will simplify the partition most effectively. The method which minimizes the product of the number of nodes and the logic depth of the DAG is chosen as the optimizer to use for this partition. We consider this figure of merit because, in order to perform depth optimization, nodes may need to be duplicated or created. The product shows the trade-off between the area and delay optimization. Because the partitions are optimized independently of each other, determining the best-fit representation is well-suited for parallelism.

C. Logic Optimization and Merging (Alg.1-δ)

The final step in the *LSOracle* flow is to perform final logic optimization on each partition with the best-fit representation that was just determined. While the body of the partition may change in the optimization, its inputs and outputs will always remain constant. This consistency means that the original partition connections in the partition manager can be redirected to this new optimized circuit and the original partition can be removed. To unify the data structure, the AIG network is converted to a MIG after all AIG partitions have been optimized and synchronized. This aims to keep the

compactness of the resulting network without losing generality since $MIGs \supset AIGs$ [8], meaning that any AND2 operation can be represented using a single MAJ3, but a single MAJ3 cannot be represented with a single AND2. In this work, we synchronize partitions serially as each partition is optimized.

IV. Experimental Results

This section first introduces the experimental methodology and then presents initial technology independent results followed by results post technology mapping and runtime.

A. Implementation Details

LSOracle is built on the top of the École Polytechnique Fédérale de Lausanne (EPFL) logic synthesis libraries [13] and is tested on a set of OPDB [2] and OpenCores benchmarks [3]. Partitioning is done using the library *KaHyPar* [14], which is similar to partitioning tools used in previous works [15; 16]. The input to *KaHyPar* is a hypergraph representation of the network. The number of partitions has a significant impact on optimization results since it influences the complexity of the partitions to optimize. In this paper, we constrained the number of partitions to give approximately 300 nodes per partition. This empirical partition size shows a good trade-off between global and local optimization. Parallelization was accomplished with OpenMP, a widely used API for explicit parallelism in C++. All programs were run on a server with two Xeon 6142 processors and 512GB RAM. Using OpenMP's default behavior for the hardware, *LSOracle* is parallelized using a max of 64 threads. For comparison, we consider the AIG optimization of *ABC* [5], a leading academic logic synthesis tool and MIG optimization using *Cirkit* [6], another state-of-the-art academic tool, against the mixed AIG and MIG optimization with *LSOracle* using the approach described in this work. The reference flow for AIG optimization is given by *ABC's resyn2*, which interleaves multiple depth and area optimization algorithms within the tool. A *resyn2*-like flow for MIGs is used by *CirKit*. In *LSOracle*, MIG partitions are also optimized through a *resyn2*-like recipe. AIG partitions in the mixed synthesis flow are optimized through an equivalent of *ABC*s rewrite (*rw*) optimization command.

B. Technology Independent Optimization

The average technology independent results shown in Table I for each benchmark are normalized to the original, pre-optimization size to allow easy comparison between benchmarks. In general, AIG optimization gives a more compact network than MIG optimization, at the price of more logic levels. *LSOracle* provides better node reduction than *CirKit* and better depth reduction than *ABC*, leveraging the strength of both approaches. This is most prominent on circuits that are heavily heterogeneous with respect to arithmetic *vs.* control logic, such as the Aquarius benchmark from OpenCores, a RISC CPU that supports the superH-2 instruction set. In that benchmark, AIG optimization with *ABC* led to a 21.6% reduction in network size and 25.4% reduction in depth; MIG optimization with *CirKit* had an impressive 64.9% decrease in network depth but at the cost of a 9.1% increase in total node count. *LSOracle's* mixed optimization showed both a 17.6% reduction in network size and a 64.5% reduction in depth, outperforming *ABC* and *CirKit* in terms of nodes×depth by 50% and 23.7% respectively.

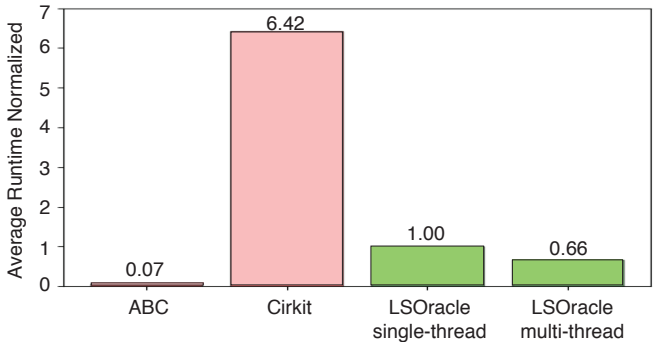

Fig. 2. Average runtime comparison normalized to LSOracle running with a single thread

C. Post-Technology Mapping Results

While technology independent results provide a generic understanding of the efficiency of the graph manipulation techniques, technology mapped results give more practical insights for the various logic optimization techniques. This section, therefore, presents technology mapped results of the logic synthesis methods shown in Section B. For a fair comparison, the technology mapping was conducted using a commercial tool with high optimization effort. Area, delay, and power results were achieved at the first positive slack.

Table II compares *CirKit* and *LSOracle* to the baseline *ABC* in terms of area, delay and power. On average, *LSOracle* achieves 5% better ADP than *ABC* and 2% better than *CirKit* and 2% better PDP and 15% better EDP than *ABC*. Especially for the Aquarius benchmark we discussed in Section B outperforms both *ABC* and *CirKit* achieving 16.99%, 12.25%, and 43.34% improvements in ADP, PDP, and EDP respectively compared to *ABC* and 7.05%, 3.60%, and 3.76% compared to *CirKit* showing that the improvements in optimization with out mixed synthesis approach remains at the post-technology mapping stage.

D. Runtime Discussion

Runtimes for the four main methodologies being compared were determined during the generation of the optimized netlists, beginning when the benchmark was read and ending when the optimized Verilog file was written. The average runtimes over all benchmarks are shown in Fig. 2. Results are the average of each benchmark normalized to *LSOracle* using a single thread.

In our current implementation, performance gains are capped by the partitioning and partition merging steps, which are single-threaded, and account for approximately 50% of the total runtime, although the exact value depends on the complexity of the network topology and the number of partitions chosen. Even bearing that in mind, multi-threading results in an average 34% improvement in runtime compared to executing *LSOracle* with one thread. Multi-threaded *LSOracle* offers substantially faster execution than *CirKit*, on average 6.76× faster, while delivering comparable average results post-technology mapping, and substantial benefits for some circuit types. *LSOracle* remains slower than *ABC* but delivers average improvements in every figure of merit post-tech-mapping at a pace that does not severely disrupt the designer's workflow.

TABLE I
TECHNOLOGY INDEPENDENT RESULTS

Circuit	Original Network		ABC		Cirkit		LSOracle Mixed	
	No. Nodes	Logic Depth	No. Nodes	Logic Depth	No. Nodes	Logic Depth	No. Nodes	Logic Depth
chip_bridge	124,565	39	72,190	26	132,857	22	70,848	34
dynamic_node	21,216	37	19,002	29	32,518	32	18,212	36
fpga_bridge	265,956	37	265,661	34	278,416	36	265,971	38
fpu	64,814	145	61,424	145	83,677	45	62,431	56
aquarius	25,058	276	19,653	206	27,346	97	20,653	98
des_perf	29,905	17	20,970	14	21,821	14	23,172	15
aes_core	21,522	26	20,418	20	19,387	23	20,223	28
ethernet	86,726	32	56,152	27	59,392	21	59,344	30
vga_lcd	126,708	24	88,816	18	94,062	19	90,269	31
DMA	24,393	27	22,395	20	21,859	21	22,626	26
Average:	79,086	66	64,668	54	77,134	33	65,375	39
Ratio:	1.00	1.00	0.82	0.82	0.98	0.50	0.83	0.59

TABLE II
TECHNOLOGY MAPPING RESULTS

Circuit	ABC			Cirkit			LSOracle		
	Arrival (ps)	Area (μm^2)	Power (mW)	Arrival (ps)	Area (μm^2)	Power (mW)	Arrival (ps)	Area (μm^2)	Power (mW)
chip_bridge	198.99	42094.91	1070.50	194.00	45864.95	1035.40	183.00	42017.93	1073.80
dynamic_node	211.00	14993.61	325.16	213.99	14941.35	334.47	208.98	15336.29	342.01
fpga_bridge	232.98	181926.67	1765.90	225.00	186347.56	1614.70	232.00	173551.92	1821.40
fpu	206.00	47573.26	1671.60	197.99	48318.35	1725.50	202.00	47759.65	1664.50
aquarius	730.98	13685.84	372.37	529.99	17135.12	472.59	531.98	15860.24	452.61
des_perf	115.00	16982.55	1105.30	122.00	16778.90	1066.70	117.50	17515.60	1134.30
aes_core	166.00	17976.79	644.90	165.00	18359.14	653.20	168.00	18146.85	649.80
ethernet	152.00	44300.57	384.32	155.99	37835.22	377.67	149.99	37588.17	373.08
vga_lcd	172.00	61773.01	510.00	174.00	61418.19	532.35	178.00	62140.19	531.00
DMA	161.00	14297.73	282.11	166.00	13890.66	279.08	164.99	14350.22	282.92
Avg.	234.60	45560.49	813.22	214.40	46088.94	809.17	213.64	44426.71	832.54
Ratio:	1.0000	1.0000	1.0000	0.9139	1.0116	0.9950	0.9107	0.9751	1.0238

V. CONCLUSION

This paper proposed a novel logic synthesis framework, *LSOracle*, which shows the potential of applying mixed logic optimization to complex networks by (i) partitioning the network to separate the different logic types, (ii) determining the best-fit recipe between an AIG and MIG based flow, and (iii) performing logic optimization using these best-fit flows and merging back the optimized partitions. Over a set of OPDB and OpenCores benchmarks, the multi-threaded structure of *LSOracle* shows about a 34% speed up in runtime when compared to a single-threaded structure and about a 6.76× speed up when compared to using MIG based optimization using *CirKit*. *LSOracle* also shows about a 5%, 2%, and 15% improvement in ADP, PDP, and EDP respectively compared to using AIG based optimization with *ABC*. Through this proposed mixed synthesis framework, this paper shows that better results can be achieved than when using a single type of logic representation for optimizing complex, heterogeneous circuits.

REFERENCES

[1] G. de Micheli, *Synthesis and optimization of digital circuits*. Tata McGraw-Hill Education, 2003.

[2] P. University, "Openpiton design benchmark," https://github.com/PrincetonUniversity/OPDB.

[3] C. Albrecht, "Iwls 2005 benchmarks," Tech. Rep., Jun. 2005.

[4] L. Clark, V. Vashishtha *et al.*, "Asap7: A 7-nm finfet predictive process design kit," 7 2016.

[5] B. L. Synthesis and V. Group, "Abc: A system for sequential synthesis and verification," 2018. [Online]. Available: http://www.eecs.berkeley.edu/ alanmi/abc/

[6] M. Soeken, "A circuit toolkit," https://github.com/msoeken/cirkit.

[7] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2016.

[8] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[9] ——, "Mixsyn: An efficient logic synthesis methodology for mixed xor-and/or dominated circuits," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2013, pp. 133–138.

[10] N. Vemuri, P. Kalla, and R. Tessier, "Bdd-based logic synthesis for lut-based fpgas," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 7, no. 4, pp. 501–525, 2002.

[11] P. Miettinen, M. Honkala, and J. Roos, *Using METIS and hMETIS algorithms in circuit partitioning*. Helsinki University of Technology, 2006.

[12] C. Yang and M. Ciesielski, "Bds: A bdd-based logic optimization system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 7, pp. 866–876, 2002.

[13] M. Soeken, H. Riener, W. Haaswijk, and G. De Micheli, "The EPFL logic synthesis libraries," May 2018, arXiv:1805.05121.

[14] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz, "k-way hypergraph partitioning via *n*-level recursive bisection," in *18th Workshop on Algorithm Engineering and Experiments, (ALENEX 2016)*, 2016, pp. 53–67.

[15] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, March 1999.

[16] R. Andre, S. Schlag, and C. Schulz, "Memetic multilevel hypergraph partitioning," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '18, 2018, pp. 347–354.