LSOracle: Using Mixed Logic Synthesis in an Open Source ASIC Design Flow

Scott Temple, Walter Lau Neto, Max Austin, Xifan Tang, Pierre-Emmanuel Gaillardon
LNIS, University of Utah, Salt Lake City, Utah, USA
pierre-emmanuel.gaillardon@utah.edu

LSOracle is a free software logic synthesis tool that leverages several types of underlying data structures and manipulation methods, including And-Inverter Graphs and Majority-Inverter Graphs to optimize highly heterogeneous circuit designs automatically. It divides large designs at the module level and selects the appropriate data structure and optimizer for each module. It is available as a standalone tool, or as a Yosys plugin, allowing easy integration with existing open source EDA toolchains, and is capable of both ASIC and FPGA synthesis. To demonstrate its capabilities, we present results using LSOracle with the OpenROAD flow for ASIC synthesis, targeting the BlackParrot benchmark and the Nangate45 library, both available in the OpenROAD repository. Using LSOracle, delay is improved by up to 46% with negligible cost in area and power. The tool is freely available at our GitHub repository [1]

***Index Terms*—EDA, Logic Synthesis, VLSI , Open Source**

I. Introduction

Realizing maximum performance in modern, complex circuit designs requires both experienced engineers writing efficient *Register Transfer Level* (RTL) code and advanced tools for every portion of the *Electronic Design Automation* (EDA) workflow. Logic synthesis, the conversion of a design from RTL to an efficient gate level implementation, stands at the top of the EDA toolchain, and is crucial to downstream tools' performance.

Logic synthesis may broadly be divided into two steps: technology independent, which optimizes the logic of a design, and technology dependent, which maps that logic onto a library of gates while optimizing the mapping for some cost function. Technology independent optimization typically consists of transforming the RTL into a homogeneous *Directed Acyclic Graph* (DAG) which describes the logic in terms of a single boolean function and potentially inverted connections between them. A variety of optimization approaches can then be applied to the graph.

The most well known DAG for logic synthesis is the *And-Inverter Graph* (AIG), which is the primary data structure used in Berkeley ABC [2]. AIGs consist only of AND-of-two nodes and edges connecting them. The edges may be inverted, which is equivalent to a NOT gate. Research has shown that the choice of Boolean function in the DAG impacts the attainable optimization [3]. For example, *Majority-Inverter Graphs* (MIGs), where each node represents the majority of three function, have advantages over AIGs for optimizing arithmetic logic [4]. *XOR-And Graphs* (XAGs) similarly have benefits for arithmetic circuits and have applications in hardware security [5]. However, every choice of data structure is a compromise, and no single data structure is ideal for every application.

This tension between optimization methods and circuit type can be resolved by having an experienced engineer select the right tool and settings to optimize each part of a design. However, experienced engineers are at a premium in small projects and academic settings, and manual intervention by the designer adds considerable time to a project.

In this paper we demonstrate LSOracle, an open source tool that allows simultaneous optimization using a variety of DAGs for each design module automatically, and supports using multiple DAGs even within a single design module. This reduces the burden on the designer and speeds the synthesis process. In most cases, LSOracle requires no designer intervention and is transparent to the user.

The remainder of this paper is organized as follows: Section II presents the details of the LSOracle architecture. Section III discusses our integration of LSOracle with the OpenROAD project and gives results for the BlackParrot benchmark using our tool. Section IV concludes and gives a brief overview of our development roadmap.

II. LSOracle Flow

LSOracle is available both as a standalone tool and as a Yosys [6] plugin. In order to limit the discussion to LSOracle itself, this section focuses on the standalone tool. The Yosys plugin uses the same fundamental functionality, but passes individual design modules into LSOracle for optimization and techmapping, in an analagous way to how ABC is employed in Yosys. Please see the Yosys documentation for details [7].

The fundamental LSOracle architecture is presented in Fig. 1. In broad terms, it consists of three steps: first, partitioning the circuit (optional in the Yosys integration), second classifying partitions for AIG or MIG optimization and performing the optimization, and third, merging back into a cohesive network.

A. Step 1: Partitioning

To partition the DAG, it is first represented as a hypergraph; *i.e.* a generalization of a graph where edges may connect an arbitrary number of nodes, rather than exactly two, as in a traditional graph. This is a convenient representation for a logic network; nodes in the DAG are nodes in the graph, and graph edges are the connections between them. Partitioning is performed with a k-way hypergraph partitioning algorithm; LSOracle currently supports KaHyPar[8] and a parallel algorithm, GMetis[9]. Each node may belong to only one partition, and partitions are chosen to be as independent as possible, with minimal connection between them. This helps alleviate the loss of global optimization that can occur if related logic is separated across multiple partitions.

B. Step 2: Classification and Optimization

After partitioning, LSOracle determines the DAG to use for each partition, using either a technology independent heuristic or an experimental neural network based classifier. AIG and MIG data structures are fully supported, with XAG currently

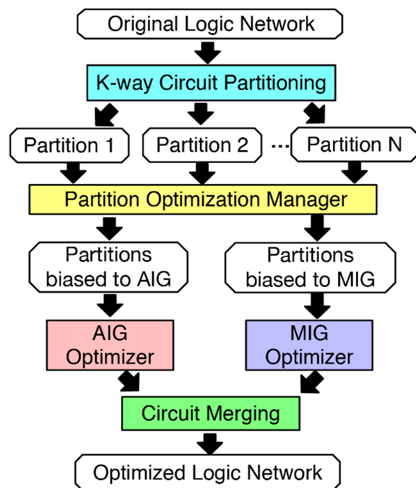

Fig. 1. LSOracle structure, showing partitioning, classification, optimization, and merging steps. In Yosys integration, each design module is passed from Yosys to this flow as the "original logic network." Using the Yosys flow, partitioning is optional because the design has already been broken into design modules.

an experimental feature. Results in this work were obtained using the heuristic classifier: Each partition is duplicated into a small, independent network, and optimization for each method is performed with a series of *rewriting*, *refactoring*, and *balancing* steps, similar to the *resyn2* command in ABC. The product of the number of nodes and the logic depth of the DAG after optimization is used as the metric to determine which classifier to use, although there are options for the user to weight network size or network depth more heavily. We have observed empirically that technology independent network size and depth correlate to area and delay, respectively, later in the design process. This heuristic is chosen to attempt to provide a balanced optimization, which does not allow network size to grow too much in order to improve clock speed, or vice versa.

C. Step 3: Merging

After selecting the best network in the classification step, the unused network is discarded, and the chosen partition is merged back into the network. LSOracle keeps track of the inputs and outputs of each partition, so to merge the optimized partition, the inputs and outputs of the original network can simply be redirected to the duplicate, optimized network. This will leave the unoptimized nodes unconnected, and a cleanup function can remove them. The optimization and merging is done in two operations: AIG partitions are merged, the merged network is converted to MIG, and MIG partitions are merged. This is done because the AND operation can be represented as a MAJ gate by adding a constant input, but not vice-versa, making it impossible to merge MIG partitions into an AIG network without adding gates.

III. OpenROAD Integration

In this section we demonstrate an integration of LSOracle with other open source tools to create a complete ASIC flow. We then test the integration with a modern, complex benchmark and evaluate the impact of LSOracle on *Power, Performance, and Area* (PPA) after placement and routing.

TABLE I
POST PLACEMENT AND ROUTING PPA FOR UNMODIFIED BLACKPARROT DESIGN USING INDICATED OPTIMIZATION

	Area (μm2)	Delay (ps)	Power (mW)
Original	20003518	8.29	5.52e+02
With ABC resyn2	20004068	10.61	5.52e+02
With ABC resyn2rs	20004586	10.62	5.52e+02
With LSOracle	20004708	6.61	5.52e+02
Improvement over Original	-0.01%	22.55%	0.00%
Improvement over resyn2	-0.003%	46.46%	0.00%
Improvement over resyn2rs	-0.004%	46.55%	0.00%

TABLE II
POST PLACEMENT AND ROUTING PPA FOR MODIFIED BLACKPARROT DESIGN ALLOWING FUNCTIONAL VERIFICATION

	Area (μm2)	Delay (ps)	Power (mW)
Original	20003450	8.33	5.52e+02
With ABC resyn2	20004326	10.68	5.52e+02
With ABC resyn2rs	20004512	10.55	5.52e+02
With LSOracle	20004326	7.16	5.52e+02
Improvement over Original	-0.004%	15.11%	0.00%
Improvement over resyn2	0.00%	39.46%	0.00%
Improvement over resyn2rs	0.00093%	38.28%	0.00%

A. Toolchain and Benchmarks

To demonstrate interoperability with other open source tools, we integrated LSOracle within the OpenROAD flow. OpenROAD is an open source suite for ASIC synthesis from RTL to GDS, including static timing analysis, placement, routing, clock tree synthesis, *etc* [10]. The OpenROAD flow uses Yosys for verilog parsing, logic synthesis, and technology mapping. In order to demonstrate the interoperability of LSOracle with other open source tools, and the performance benefits available, we integrated LSOracle into the OpenROAD flow to improve the logic synthesis.

OpenROAD's default flow performs no logic synthesis, using only trivial optimizations available in Yosys and using ABC within Yosys only for technology mapping. In order to show the benefits of adding more sophisticated optimization methodologies, we modified the OpenROAD flow to use an ABC optimization script (*resyn2* and *resyn2rs*) and to use the LSOracle plugin's high effort approach after Yosys' generic synthesis but prior to tech-mapping. ASIC tech mapping was performed using ABC in all tests.

B. Results

Tables I and II show the results of this integration using the BlackParrot benchmark [11] and Nangate45 library included in the OpenROAD public repository after placement and routing. In the original BlackParrot design, shown in Table I, we see improvements in delay of 22.55% over the original flow and of about 46% over both ABC optimization scripts, while showing negligible difference in area or power after placement and routing. Note that adding additional optimization scripts with ABC degrades performance; this is unusual, and a longer discussion follows. However, it is likely because only relatively small changes were made to the DAG, and post-technology mapping results do not correlate perfectly to technology independent network size.

TABLE III
TECHNOLOGY INDEPENDENT RESULTS FOR ORIGINAL BLACKPARROT DESIGN

	Before		After	
	Nodes	Depth	Nodes	Depth
ABC resyn2	243049 AND	1031	215721 AND	1023
ABC resyn2rs	243049 AND	1031	214258 AND	1023
LSOracle high effort	243049 AND	1031	221700 MAJ	520

To verify these results, we successfully performed functional verification using a testbench and simulation script provided to us by the BlackParrot team. This simulation required a slightly modified BlackParrot design, however, which was also supplied to us. This updated design had ports moved up in the design hierarchy so that the testbench could have access to them.

The same test described above was run on this updated design, and the results are shown in table II. With this modified, testable design, we were able to achieve a 15.11% improvement in delay over the original flow and an approximately 39% improvement over ABC optimization post placement and routing. This broadly agrees with the results in the original design. Some loss of performance is expected because the modifications both increase the total size of the network and complicate the topology, changing both partitioning and optimization.

Because it was unexpected that optimization with ABC did not improve performance after placement and routing, we investigated technology independent performance of each optimizer on the unmodified design; these results appear in Table III. AIG optimization using ABC reduced network size by up to 12% and depth by 1%. LSOracle improved network size by 9% and depth by 49%. Because LSOracle uses MAJ, rather than AND nodes, a one to one comparison is not possible. However, it is clear that the large reduction in network depth correlates with a reduction in delay. This correlation is not perfect, however, as shown by the AIG results and the slight increase in area seen with LSOracle. The technology mapper's performance has a dependence on network structure that cannot be completely reduced to node count and logic depth. In this case, the original network has a structure which performs well with the technology mapper, outweighing the benefits seen by AIG optimization alone.

IV. CONCLUSION

In this paper, we presented LSOracle and demonstrated the benefits that it can bring to an open source toolchain. We integrated our tool into an ASIC flow using OpenROAD, with more than 20% improvement in clock speed, and minimal cost in area and power on the BlackParrot benchmark. This was accomplished with no manual designer intervention; usage of the modified OpenROAD flow is identical from an end user perspective.

LSOracle is available on GitHub under the MIT license [1]. It is available both as a stand alone tool and as a Yosys plugin. Although this work showcased ASIC synthesis, LSOracle can also be used in FPGA toolchains, and has been tested with VTR [12] and OpenFPGA [13]. Ongoing work includes a cloud based version, full support for timing driven synthesis and resynthesis, and a variety of hardware security features.

REFERENCES

[1] "Lsoracle." [Online]. Available: https://github.com/LNIS-Projects/LSOracle

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

[3] 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.

[4] 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.

[5] E. Testa, M. Soeken, L. Amarù, and G. D. Micheli, "Reducing the multiplicative complexity in logic networks for cryptography and security applications," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, June 2019, pp. 1–6.

[6] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[7] C. Wolf, "Yosys manual." [Online]. Available: http://www.clifford.at/yosys/files/yosys_manual.pdf

[8] 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.

[9] X. Sui, D. Nguyen, M. Burtscher, and K. Pingali, "Parallel graph partitioning on multicore architectures," vol. 6548, 12 2010, pp. 246–260.

[10] T. Ajayi, V. A. Chhabria *et al.*, "Toward an open-source digital flow: First learnings from the openroad project," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[11] Z. Azad, L. Delshadtehrani *et al.*, "The blackparrot processor: An open-source industrial-strength rv64g multicore processor," 2019.

[12] J. Luu, J. Goeders *et al.*, "Vtr 7.0: Next generation architecture and cad system for fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2617593

[13] X. Tang, E. Giacomin, B. Chauviere, A. Alacchi, and P.-E. Gaillardon, "Openfpga: An opensource framework for agile prototyping customizable fpgas," *IEEE Micro*, vol. PP, pp. 1–1, 05 2020.

ACKNOWLEDGEMENTS

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number 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.