LSOracle 2.0: Capabilities, Integration, and Performance

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

LSOracle is a tool for logic synthesis which uses several directed acyclic graph representations concurrently when optimizing a design; for example a combination of And-Inverter Graphs and Majority-Inverter Graphs. Version 2.0 of LSOracle has recently been released. This paper describes the changes made in version 2.0 and presents recent benchmark results compared to earlier versions of the tool. Execution time has been reduced by a factor of ten with an accompanying 5-10% improvement in area-delay product.

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

I. Introduction

Logic synthesis is the conversion of a circuit design from *Register Transfer Level* (RTL) to a technology dependent netlist representation, such as standard cells or FPGA look-up tables. Because all downstream *Electronic Design Automation* (EDA) tools operate on the netlist generated by the logic synthesis tool, its performance is critical to the overall performance of the EDA toolchain.

In general, logic synthesis consists of two steps: technology independent optimization, which minimizes the logic of a given design in an abstract sense, and technology dependent, which maps the logic onto a library of elements such as manufacturable standard cells or FPGA resources while optimizing the mapping for some cost function. The focus of this work is on technology independent optimization.

The most common technique for technology independent logic optimization is the use of Boolean networks, *Directed Acyclic Graphs* (DAGs) which describe the logic using nodes representing Boolean functions and edges which describe connections between them. The most common DAG for logic synthesis is the *And-Inverter Graph* (AIG), which consists of AND-of-two nodes connected by edges which may optionally be inverted [1]. AIG synthesis is well known, and is the basis for the popular open-source synthesis tool ABC [2]. Optimization with other types of DAG is possible, however, and research has shown that the choice of DAG impacts the achievable optimization results. For example, *Majority-Inverter Graphs* (MIGs), structures similar to AIGs in which nodes represent the majority of three function, have benefits when optimizing arithmetic logic because they efficiently represent the carry operator [3]. Similarly, *XOR-And Graphs* (XAGs) have benefits for hardware security, where minimizing the number of and gates may be desirable [4]. Each type of DAG is a compromise, performing well for some categories of circuit and less well for others. When a circuit is highly heterogeneous, no single DAG will be ideal for all regions.

LSOracle is a logic synthesis tool designed to automatically partition circuits and use the appropriate DAG when optimizing each partition, improving performance on heterogeneous circuits where no single representation is ideal [5]. It has recently been updated to version 2.0, featuring improved *quality of results* (QoR), greatly reduced runtime, and a variety of new features which may be of interest to researchers and advanced users.

The remainder of this paper is organized as follows: Section II presents the LSOracle architecture. Section III discusses recent changes made in version 2.0 of LSOracle and presents recent benchmark results and a comparison of version 2 with the previous release. Section IV gives a brief overview of our development roadmap, and section V concludes.

II. LSOracle Flow

The fundamental mixed synthesis flow used in LSOracle is presented in Fig. 1. This is the flow used in the default mode for the software. When used manually, many other workflows are possible.

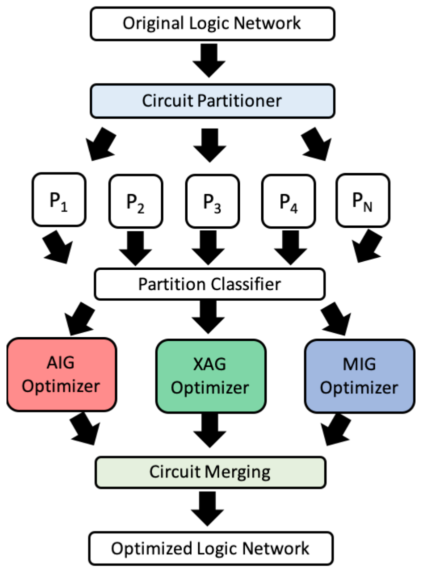

Fig. 1. LSOracle's mixed synthesis architecture, showing the three major steps: partitioning, classification/optimization, and circuit merging.

A. Step 1: Partitioning

The first step in LSOracle's mixed synthesis is to partition the circuit into smaller sub-networks. By default, this is done with K-way hypergraph partitioning using KaHyPar [6]. In a traditional graph, each edge connects exactly two vertices; a hypergraph is a generalization in which each edge can connect an arbitrary number of vertices, making them a convenient representation of a Boolean network. The number of partitions can be chosen in several ways, but the default is to fix the average partition size at a constant.

TABLE I
RUNTIME COMPARISON, LSORACLE VERSION 2.0 VS PREVIOUS STABLE

Benchmark	Previous (AIG)	V2.0 (AIG)	Reduction	Previous (MIG)	V2.0(MIG)	Reduction
adder	6.2	1.0	83%	1.5	0.4	74%
arbiter	144.6	2.4	98%	9.4	5.1	45%
bar	25.5	1.1	96%	3.2	1.1	65%
log2	3383.0	32.8	99%	3292.6	120.4	96%
max	113.8	1.2	99%	32.6	1.8	95%
mem_ctrl	108.1	36.3	66%	746.5	29.7	96%
multiplier	1949.1	26.3	99%	1304.7	35.7	97%
sin	1245.2	4.4	99%	84.7	4.5	95%
sqrt	3467.0	16.0	99%	1132.5	14.2	99%
square	872.0	9.6	99%	403.7	10.7	97%
voter	628.7	6.3	99%	133.4	5.4	96%

B. Step 2: Classification and Optimization

After partitioning, the tool determines which DAG representation best fits each sub-network. By default this is done using an *area-delay product* (ADP) oriented heuristic which considers DAG size and logic depth after optimization with a standardized optimization recipe for each DAG. The standard recipe consists of a combination of rewriting, balancing, and refactoring, similar to the *resyn2* script in ABC.

C. Step 3: Merging

When each partition has been optimized, they are reassembled into a single optimized network. In order to preserve the optimizations, this network type must be general enough to include each DAG used in step 2. For example, in the default AIG/MIG mixed synthesis flow, the final network is an MIG, as one majority node can represent a 2-input AND function by adding a constant input, but the reverse is not possible without expanding the network size.

After technology independent optimization, the network is mapped onto either FPGA look-up tables or ASIC standard cells. LSOracle is frequently used in conjunction with its supplied Yosys plugin, so when LSOracle is finished merging the network, it returns control to the Yosys script that invoked it [7].

III. LSORACLE VERSION 2.0

The recently released version 2.0 of LSOracle has brought many changes. Some are engineering changes which are not evident to the end user: all libraries have been updated and moved into submodules; unit testing is being integrated with the codebase; the *continuous integration/continuous delivery* (CI/CD) pipeline has been dramatically improved, etc. User-facing changes may broadly be divided into new features, improved integration with other EDA tools, and runtime and QoR improvements.

A. New Features

Because LSOracle strives to be automatic, the average user will not notice many of the new features in version 2.0. They are available to advanced users, however, and over the next several months many will be integrated with the automatic mixed synthesis mode, improving performance without changing the user experience. Notable new features include:

- Native standard cell mapping;
- Improved LUT mapping;
- Expanded support for XAG and XMG networks;
- Functional reduction;
- Exact synthesis support;
- Native equivalence checking;
- Structure aware partitioning [8].

B. Toolchain Integration

Three major changes have been made regarding LSOracle's integration with other EDA tools. Most significantly, LSOracle is now included with OpenROAD by default [9]. OpenROAD users may enable LSOracle by setting the USE_LSORACLE environment variable. This integration has not yet been updated to use LSOracle 2.0; the transition will be completed during Q4 2021. In addition, the LSOracle and the LSOracle Yosys plugin repositories have been merged, simplifying installation. The plugin has also been updated, fixing several bugs and modifying the default synthesis recipes. Finally, a Yosys frontend is available for the version of the commercial Verific parser which is licensed through the DARPA Toolbox Initiative. This integration provides DARPA performers a very robust Verilog frontend for OpenROAD or any other toolchain using Yosys.

C. Performance

Updates to the EPFL Logic Synthesis libraries [10], used in LSOracle, and adjustments to the default optimization recipes for each DAG have resulted in changes in both runtime and QoR. In general, runtime has been substantially reduced, while QoR shows overall improvement and a trend toward reduced delay at the expense of increased area compared to the previous version of LSOracle.

Table I shows the run time of the AIG and MIG modes of LSOracle version 2.0 compared to the most recent pre-refactor version over a subset of the EPFL benchmarks [11]. AIG and MIG homogeneous modes were used in this test to remove the effect of partitioning and focus on logic synthesis speed. The subset of benchmarks was chosen by, for convenience, excluding any benchmark which did not complete the pre-refactor MIG optimization in one hour. Of those that did not time out, all benchmarks which completed the pre-refactor MIG optimization in less than one second were also excluded due to poor reproducibility caused by other tasks running. Run

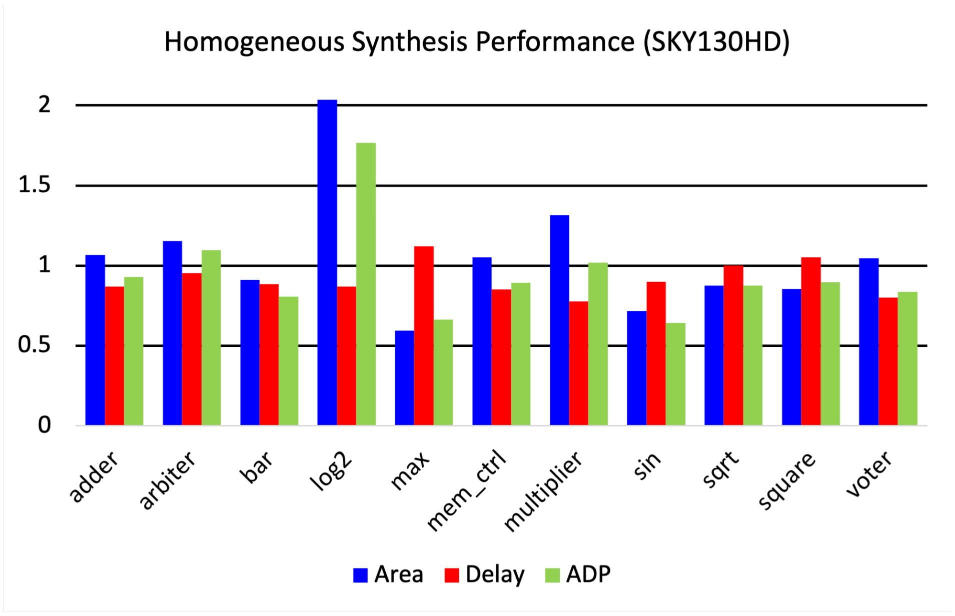

Fig. 2. Area, delay and ADP of LSOracle version 2.0 on a selection of EPFL benchmarks after technology mapping onto the Skywater 130HD library, normalized against the previous version of LSOracle. Arithmetic benchmarks optimized with MIG, control benchmarks with AIG.

time benchmarks were performed on an Intel i5-7360U with 8GB RAM running Mac OS 11.4. On average, version 2.0 reduces run time for AIG optimization by 94% and MIG run time by 87%. Averaging all run times, version 2.0 is an order of magnitude faster than the previous stable release.

Fig. 2 shows the performance of the new version compared to the previous stable release over a subset of the EPFL benchmarks after technology mapping with Yosys onto the SKY130HD library. The same subset of benchmarks used in the run time comparison was used for this analysis. Because the EPFL benchmarks are divided into arithmetic and control logic benchmarks, this set was used to compare the logic synthesis performance without any influence from partitioning. AIG optimization was used for control logic and MIG for arithmetic logic for both versions. In general, version 2.0 reduces delay more successfully than the previous version of LSOracle, but at the cost of increased area. On average, delay was reduced 8.5% and area was increased by 5.5%, leading to an average 5.2% reduction in *area-delay product* (ADP).

Finally, fig. 3 and fig. 4 show improvements in mixed AIG/MIG synthesis performance on two real world benchmarks: picoRV and the chip_bridge from the *Open Piton Design Benchmark* (OPDB) [12] [13]. LSOracle results were obtained using the Yosys plugin with default settings. Results in both figures are normalized against a Yosys flow which calls ABC's *resyn2rs* script rather than LSOracle. Technology mapping and all other aspects of the scripts are identical for all optimization methods. In both cases version 2.0 shows reduced delay and slightly increased area compared to the previous version, consistent with the trend seen in the homogeneous synthesis experiments. For picoRV, the delay reduction is 10.0%, with a corresponding 5.4% reduction in ADP. For chip_bridge the figures are 13.2% and 11.8%, respectively. ADP improvements over optimization with *resyn2rs* were 20.9% for picoRV and 16.8% for chip_bridge.

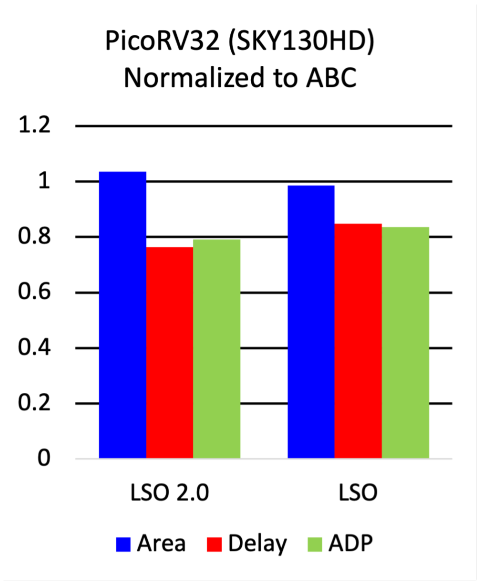

Fig. 3. Comparison of LSOracle 2.0 (LSO 2.0) and previous LSOracle (LSO) for the PicoRV32 benchmark after mixed synthesis optimization using the LSOracle Yosys plugin and technology mapping onto the SKY130HD library using Yosys. Results normalized against ABC performance for each value. As seen in the EPFL benchmark results above, the updated version of LSOracle improves delay and ADP compared to the previous version.

IV. Future Roadmap

LSOracle is in active development, and changes are published regularly on the Github repository [14].

Current development is focused on two related goals: timing driven synthesis and improving technology mapping. OpenSTA has been integrated with the application, as has a native standard cell mapper and an array of timing and area oriented

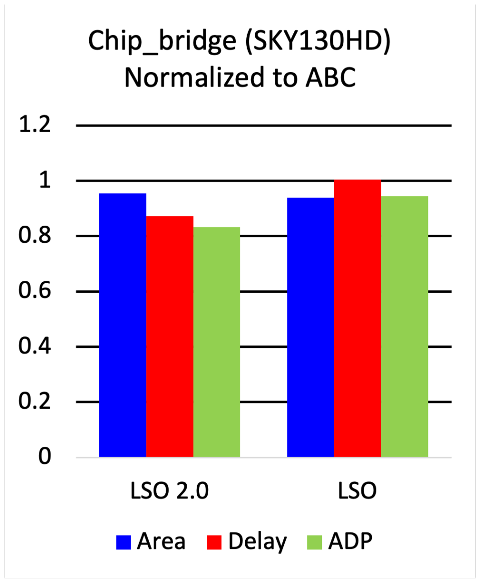

Fig. 4. Comparison of LSOracle 2.0 (LSO 2.0) and previous LSOracle (LSO) for the OPDB chip bridge benchmark after mixed synthesis optimization using the Yosys plugin and technology mapping onto the SKY130HD library using Yosys. Results normalized against ABC performance for each value.

optimization recipes, bringing timing driven synthesis closer to reality [15]. At the same time, an improved technology mapper is being developed with support for multi-output gates, addressing a long standing limitation of open-source EDA tools. Besides these major efforts, development is underway on the following:

- LUT synthesis tailored for OpenFPGA generated fabrics;
- Improved integration with Yosys through native RTLIL I/O;
- Post placement & routing physical resynthesis when used with OpenROAD;
- Improving technology mapping with machine learning;
- Integrating hardware security metrics into the logic synthesis process;
- Improving performance when used with high-level synthesis tools through custom optimization recipes.

Community contributions, feature requests, and bug reports are welcome.

V. Conclusion

For the average user, the newest version of LSOracle gives 10% lower delay and runs in 10% of the time. For researchers, developers, and certain niche users, version 2.0 brings a host of interesting features including novel partitioning techniques, a standard cell mapper which can operate natively on MIGs, a variety of optimization recipes for each DAG, and robust XAG and XMG support. In addition, long requested features like incremental timing and support for mapping to multi-output gates are now under active development and should be released by early 2022.

LSOracle is now distributed with OpenROAD; many users interested in ASIC synthesis will find that they already have a copy and can add mixed synthesis to their workflow simply by setting an environment variable. For other applications, besides source code, Docker images are available on Docker Hub [16] and Ubuntu binary packages are available on Launchpad [17].

References

[1] A. Mishchenko and R. Brayton, "Scalable Logic Synthesis using a Simple Circuit Structure," *IWLS*, 2006.
[2] Berkeley Logic Synthesis and Verification Group. "Abc: A system for sequential synthesis and verification." http://www.eecs.berkeley.edu/ãlanmi/abc/.
[3] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," IEEE TCAD, vol. 35, no. 5, pp. 806-819, 2016.
[4] E. Testa, M. Soeken, L. Amarú, and G. De Micheli, "Reducingthe multiplicative complexity in logic networks for cryptographyand security applications," *DAC*, 2019.
[5] M. Austin, S. Temple, W. L. Neto, L. Amarù, X. Tang, and P. Gaillardon, "A scalable mixed synthesis framework for heterogeneous networks," *DATE*, 2020.
[6] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz, "k-way hypergraph partitioning via *n*-level recursive bisection," *ALENEX*, 2016.
[7] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," *Austrochip*, 2013.
[8] A. Snelgrove, S. Temple, and P. Gaillardon, "Structure Aware Partitioning for Mixed Logic Synthesis," *IWLS*, 2021.
[9] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, and et al., "Toward an open-source digital flow: First learnings from the openroad project," *DAC*, 2019.
[10] M. Soeken, H. Riener, W. Haaswijk, and G. De Micheli, "The EPFL logic synthesis libraries," arXiv:1805.05121, 2018.
[11] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL Combinational Benchmark Suite," *IWLS*, 2015.
[12] C. Wolf, "PicoRV32," https://github.com/cliffordwolf/picorv32
[13] Princeton University, "Open Piton Design Benchmark," https://github.com/PrincetonUniversity/OPDB
[14] https://github.com/lnis-uofu/LSOracle
[15] J. Cherry, W. Scott OpenSTA https://github.com/lnis-uofu/LSOracle
[16] https://hub.docker.com/orgs/lnis/
[17] https://launchpad.net/l̃nis-uofu

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.