AFRL-RY-WP-TR-2023-0136

A LEARNING-BASED ORACLE FOR AUTOMATIC OPTIMIZATION

Pierre-Emmanuel Gaillardon
University of Utah

SEPTEMBER 2023
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with The Under Secretary of Defense memorandum dated 24 May 2010 and AFRL/DSO policy clarification email dated 13 January 2020. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2023-0136 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//

CHRISTOPHER A. BOZADA
Program Manager
Aerospace Components & Subsystems Division

//Signature//

GENE M. WILKINS, Lt Col, USAF
Deputy Chief
Aerospace Components & Subsystems Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
September 2023	Final	12 June 2018	31 August 2022

4. TITLE AND SUBTITLE

A LEARNING-BASED ORACLE FOR AUTOMATIC OPTIMIZATION

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8650-18-2-7849	N/A	62716E

5d. PROJECT NUMBER	5e. TASK NUMBER	5f. WORK UNIT NUMBER
N/A	N/A	Y1TF

6. AUTHOR(S)

Pierre-Emmanuel Gaillardon

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Utah
257 South 1400 East
Salt Lake City, Utah

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'SREPORT NUMBER(S)
Air Force Research Laboratory, Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command, United States Air Forces	Defense Advanced Research Projects Agency (DARPA/MTO) 675 North Randolph Street Arlington, VA 22203	AFRL/RYD	AFRL-RY-WP-TR-2023-0136

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

This material is based on research sponsored by the Air Force Research Laboratory (AFRL) and the 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 the Air Force Research Laboratory (AFRL), the Defense Advanced Research Projects Agency (DARPA), or the U.S. Government. Report contains color.

14. ABSTRACT

Standing at the upstream of IDEA TA-I design flow, logic synthesis is a crucial step and profoundly impacts the performance of downstream tools. Especially in very challenging and competitive applications, such as individual combat equipment, radar, sonar, etc., the superiority of a design depends on how it is architected and optimized at the logic level. To enable “no-human-in-the-loop” unified layout generator and advancements in PPA, a high-quality and fast logic synthesis platform is required. This project provides a public release of a learning-based logic synthesis platform exploiting logic optimization for combinational benchmarks, with an improved platform for runtime and memory usage.

15. SUBJECT TERMS

Electronics design flow, logic synthesis, computer architecture, logic level optimization, learning-based logic synthesis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
Unclassified	Unclassified	Unclassified	SAR	59

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Christopher Bozada	N/A

Table of Contents

Table of Contents

List of Figures

Figure **Page**

List of Tables

Table **Page**

1 SUMMARY

The LSOracle project has developed an open-source, extensible framework for logic synthesis using a variety of optimization strategies, including use of multiple strategies on a single design by partitioning the target circuit and applying the best-fit optimizer to each partition. Over the course of the project, the tool has evolved from a collection of scripts linking disparate tools to an integrated, cloud-ready, application, integrated with existing open-source and commercial electronic design automation tools, such as Yosys, Verific, ABC, PandA-Bambu, OpenROAD, and OpenFPGA. A standard mixed-logic synthesis flow was developed using a combination of and-inverter graphs, both natively and within ABC, majority-inverter graphs for improved arithmetic performance, xor-and graphs for cryptographic core and other applications, and xor-majority graphs.

2 INTRODUCTION

As the complexity of semiconductor chips increased rapidly during the last decades, Electronic Design Automation (EDA) has become one of the indispensable technology enablers for System-On-Chip (SoC) designs. Translating Register-Transfer-Level (RTL) design description into physical layouts of a full chip requires a complex flow of EDA tools. Standing at the upstream of such flows, logic synthesis aims at a compact translation of RTL schematics into gate-level implementationsmand is a crucial step that profoundly impacts the performance of downstream EDA tools. This is especially true in very challenging and competitive applications, e.g., for defense and military purpose, where the superiority of a design depends more and more on how it is architected and optimized at the logic level, rather than relying only the advancements of technology.

In this context, logic manipulation and optimization are the key design technologies that allow the Department of Defense to maintain large competitive advantages. However, the growing complexity of semiconductor chips has revealed two critical limitations of contemporary EDA techniques. First, while contemporary EDA methodologies are extremely well developed and highly optimized, they usually only target specific types of logic networks. Another contemporary issue of logic synthesis is the difficulty to parallelize the optimization steps. To overcome these two limitations and enable an open-source, extensible framework for logic synthesis, the project followed the tasks and milestones described below in Figure 1 and detailed in the following chapters.

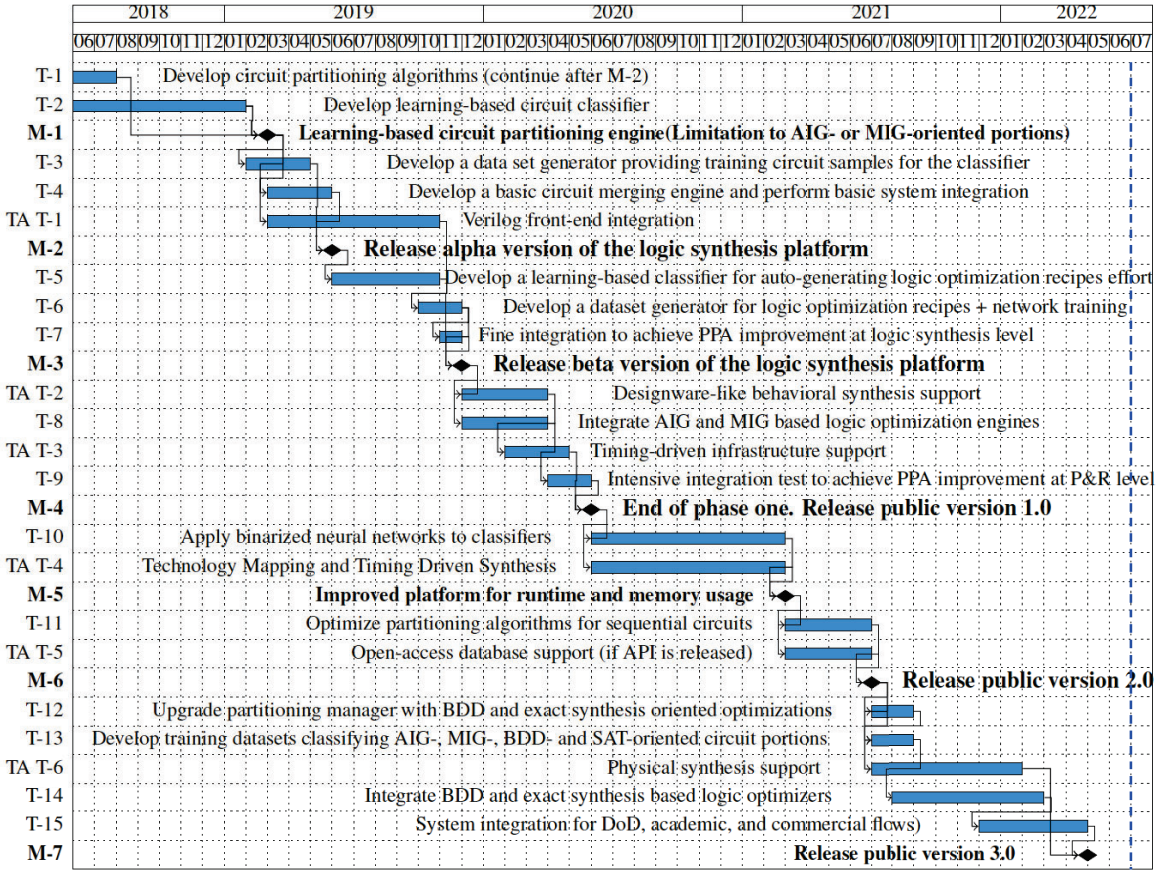

Figure 1: Tasks and Milestones Schedule

3 ACTIVITY REPORT

Listed below is a diary of the major milestones and accomplishments we have accomplished over the project:

- 6/15/18 Creation of github repository https://github.com/LNIS-Projects/LSOracle
- 6/29/18 Auto-generation of optimization recipes for ABC
- 7/13/18 Improvement on ABC optimization recipes to print out post-synthesis QoR
- 7/19/18 Deployment of the physical computer server with logic synthesis tools, i.e., ABC, Cirkit, LS-showcase, Mock-turtle, hMETIS, machine learning frameworks, i.e., Tensorflow, and P&R tools, i.e., Synopsys, Cadence and Mentor tool suites installed
- 7/23/18 Creation of logic synthesis framework based on open-source logic synthesis libraries
- 7/27/18 Auto-generation of optimization recipes for Cirkit
- 8/03/18 Improvement on Cirkit optimization recipes to print out post-synthesis QoR
- 8/5/2018 Addition of an interface to our logic synthesis platform which can output hypergraph files representing AIG networks as the input for hMETIS
- 8/10/2018 Addition of an interface to our logic synthesis platform that can parse circuit partitions generated by hMETIS
- 8/20/2018 Successful conversion from an AIG network to a MIG network
- 8/31/18 Built examples of potential CNN options for Tensorflow in Python and C++
- 9/4/2018 Completion of the truth table generator which extracts truth tables from circuit partitions
- 9/4/2018 Adaptation of the MIG network to support sequential elements
- 9/6/2018 Addition of a blif writer to our logic synthesis platform that can output the truth table representations
- 9/7/18 Completion of a CNN classifier for the circuit partitions using Tensorflow framework
- 9/11/2018 Completion of an alpha version of converting MIG networks back into AIG networks
- 9/13/2018 Addition of a AIG writer to our logic synthesis platform that can output MIG networks in the aiger format
- 9/19/18 Successful verification on the correctness of circuit partitions created by hMETIS

- 9/24/18 Extended training set to area-oriented/delay-oriented/area-delay-oriented
- 10/5/18 Completion of a functional sequential MIG optimizer
- 10/11/18 Completion of truth table representation for circuit partitions to interface CNN classifier
- 10/15/18 Integration of OpenTimer into the LSOracle framework
- 10/16/18 Completion of a truth table image writer for circuit partitions and successful interface to the TensorFlow framework
- 10/25/18 Validated improvements achieved by the sequential MIG optimizer against AIG counter- parts at pre-technology-mapping stage
- 11/12/18 Completion a neural network using TensorFlow ready for training
- 11/13/18 Validated improvements achieved by the sequential MIG optimizer against AIG counter- parts at post-technology-mapping stage (using Design Compiler)
- 11/14/18 Updated training set with circuit partitions generated by the EPFL benchmark set to en- hance neural network training
- 12/11/18 Achieved preliminary results on TensorFlow circuit classification
- 12/15/18 Successfully performed classification using our tensorflow model with external python scripts
- 12/20/18 Successfully performed circuit merging of partitions in LSOracle
- 12/22/18 Successfully created our full proposed flow using python scripts for each of the three steps: partitioning, classification, and optimization and merging
- 12/26/18 Successfully achieved noticeable optimization improvements with LSOracle over the state of the art with toy benchmarks after ASIC technology mapping
- 12/27/18 Static Timing Analysis of optimized benchmarks comparing the mapped net lists generated by design compiler for ABC optimized circuits and sequential MIG optimized circuits
- 01/11/19 Successful implementation of a generic hypergraph generation for Boolean networks
- 01/15/19 Successful implementation of the partition view, a new class to hold nodes belonging to a given partition
- 01/20/19 Successful implementation of the partition manager, a new class to manage and synchro- nize all partitions with the original network
- 02/01/19 Successfully recreated partitioning functionality from within LSOracle using the

open source library KaHyPar integrated with the partition manager

- 02/10/19 Successfully partitioned and optimized circuit partitions using either AIG or MIG based optimization
- 02/17/19 Successfully implemented partitions to deal with sequential circuits. AIG and MIG se- quential circuits may be optimized through partitions, with a restriction for mixing both optimizers
- 03/02/19 Successfully implemented darknet library
- 03/08/19 Tensorflow model imported
- 05/05/19 Comparison of critical paths between OpenTimer and OpenSTA complete; both tools give the same critical path
- 05/07/19 Public Alpha release of LSOracle
- 05/15/19 Comparison completed between the critical paths reported by a commercial tool after elaborating the design and after technology mapping. Reports show that critical paths change too much, and a timing driven technology independent optimization missleading
- 05/15/19 Implemented new code review/QA requirements for repository
- 05/29/19 Converted majority of OpenPiton Design Benchmark (OPDB) modules to AIGER format for further testing.
- 06/03/19 First ”Gold Standard” benchmark for partitioning engine completed
- 06/06/19 Implemented automatic end-to-end tests at build-time
- 06/07/19 Optimization technique which combines adjacent partitions that are classified to use the same optimization method is implemented; shows up to 18% improvement in high effort mode
- 06/12/19 Converted first benchmark from Titan23 to AIGER format for further testing
- 06/19/19 Began converting Titan23 benchmarks to AIG format; ran fpga bridge from OPDB through LSOracle; at 266K nodes, this is the largest design tested on LSOracle so far
- 07/01/19 Multithreading implemented in LSOracle; runtime reduced more that 50% on picoRV
- 07/15/19 - 07/19/19 Attended ERA summit and integration exercise in Detroit
- 07/22/19 Began development of pathological benchmarks, as discussed in closed session
- 08/05/19 Began integration of LSOracle into Yosys
- 08/06/19 Techmapped all NPN4 functions for use in in-house techmapper

- 08/08/19 Began writing LSOracle techmapper
- 08/09/19 First LSOracle function working inside Yosys
- 08/15/19 All LSOracle functionality integrated with Yosys
- 08/23/19 Proof of concept LSOracle techmapper completed
- 09/12/19 New pathological benchmark for control logic run through ABC and LSOracle MIG opti- mization: sparc IFU from OPDB.
- 09/18/19 Sequential support added to mixed optimization algorithm
- 09/27/19 LSOracle techmapper now populates internal storage with standard cells, rather than treat- ing each LUT's mapping as immutable
- 10/16/19 Obtained gMETIS source code from UT Austin team
- 10/20/19 Added sequential support to BLIF reader and writer, improved sequential support in ver- ilog writer
- 10/22/19 Galois running on our development server
- 10/24/19 Pull request for sequential API in mockturtle networks accepted by EPFL
- 10/28/19 Techmapper passes c432 and c499 from ISCAS85. Bugs remain in larger benchmarks
- 11/03/19 Updated AIG recipe. Performance equivalent to ABC, but runtime still slower
- 11/06/19 Techmapper passes all of ISCAS85. Shift focus to benchmarking and optimization
- 11/08/19 Begin work on simple optimization in techmapper, removing duplicated inverters from network
- 11/13/19 Finished implementing removal of duplicate inverters. Brings average network size in terms of number of standard cells from 1.85×ABC to 1.53×
- 11/14/19 Began openSTA integration
- 11/15/19 Updated scripts to build techmapping database to optimize the underlying DAG for each NPN class before mapping. Average network size reduced to 1.31×ABC, in terms of standard cell count
- 11/18/19 Updated techmapper to support complete ASAP7 library with AOI gates, built database using complete library
- 11/20/19 LSOracle Beta RC1 released

- 11/21/19 DARPA site visit to University of Utah
- 11/25/19 Ported techmapping branch to refactored codebase from Beta RC1; average network size, in terms of standard cell count, now 1.14×ABC
- 11/26/19 Built extended database for techmapping with support for different DAGs for each NPN class
- 12/03/19 Ran STA on ABC and LSOracle networks using a commercial tool. Found that average area over 26 circuits for LSOracle is 1.23×ABC, while delay is 1.37×ABC
- 12/05/19 Added interface to allow external partitioning using gMETIS
- 1/06/20 - 1/10/20: Integration exercise
- 1/09/20 Integrated LSOracle with PRGA flow
- 1/09/20 Obtained pickled Black Parrot core with fakeRAM for testing with LSOracle
- 1/13/20 Sequential LUT mapping working on small benchmarks
- 1/15/20 Received list of finalized phase I benchmarks: EPFL, OPDB, Black Parrot, LeWiz Ethernet core
- 1/23/20 Resolved cut rewriting bug identified in integration exercise
- 1/27/20 Built GF14 database for LSOracle techmapper
- 1/30/20 Resolved BLIF topological order bug identified in integration exercise
- 2/4/20 Assisted Sandia team with LSOracle to replace custom python script in experiments on majority based logic synthesis
- 2/5/20 Completed initial testing of phase I benchmark suite with LSOracle and commercial tools
- 2/7/20 Began generating technology independent results with commercial tools using a custom liberty file to force AIG and MIG output
- 2/14/20 Added support for 6 input NPN classes to LSOracle techmapper
- 2/25/20 Began integrating libABC to improve AIG performance and add support for LUT network resynthesis
- 3/3/20 Integrated libABC optimization for AIG networks
- 3/9/20 Integrated libABC LUT mapping for AIG networks
- 3/11/20 BLIF I/O pull request merged into EPFL master

- 3/12/20 Meeting with EPFL team to plan lossless MIG mapping
- 3/13/20 Began integrating liberty parser into LSOracle
- 3/12/20: Meeting with OpenROAD team to plan LSOracle integration into tool chain.
- 3/15/20 Liberty file view implemented, tested with ASAP7 library.
- 3/18/20 Implemented LUT mapper in libabc view, returning LSOracle LUT network from libabc tech mapper.
- 3/19/20 Begin testing liberty file view with GF12 library.
- 3/24/20 Modified libabc to dump network into pytorch format, allowing research into machine learning based techmapping
- 03/25/20 Created lossless synthesis API
- 4/2/20 Benchmarks of new LSOracle flow show improved delay vs. ABC's native LUT mapping at cost of increased area
- 4/6/20 Moved libABC LUT mapping to the miniLUT framework, simplifying integration.
- 04/09/20 Created initial cut enumeration algorithm for lossless MIG synthesis
- 4/13/20 Added sequential support to libABC integration
- 4/17/20 Began porting LSOracle Yosys integration to plugin architecture.
- 4/23/20 Initial benchmarks of refactored LSOracle show small improvement vs. ABC. Begin work on improving PPA
- 4/28/20 Initial XAG support added to LSOracle
- 5/5/20 LSOracle XAG benchmarking shows performance better than current state of the art
- 5/7/20 OpenROAD flow installed on server, allowing LSOracle integration to begin
- 6/4/20 Initial post P&R results with VPR show large improvement in delay with MIG preprocessing
- 6/10/20 MIG preprocessing showing average 12% improvement in ADP compared to ABC9 post P&R for FPGA flows across EPFL benchmark suite
- 06/14/20 Successfully integrated LSOracle plugin into the OpenROAD flow
- 6/15/20 Passed BlackParrot core through initial integration of OpenROAD and LSOracle. Find errors in output verilog due to bug in EPFL libraries
- 7/9/20 Received modified BlackParrot design with ports moved to allow functional

verification

- 7/15/20 Modified BlackParrot passed through LSOracle OpenROAD flow. Results comparable to unmodified design. Result passes functional verification
- 7/16/20 FPGA-tool-perf results show slight improvement compared to symbiflow/ABC9 in tech- nology independent results, no benefit after placement and routing for large benchmarks
- 7/24/20 API for lossless synthesis completed
- 8/4/20 Development begins on cloud based LSOracle
- 8/7/20 Discussion with OpenROAD team about BlackParrot results reveal that another SDC file exists, but is not publicly available. BlackParrot team working on getting permission to share more complex SDC.
- 8/14/20 Discussion with ASSURE team about optimizing obfuscated RTL and quantifying obfus- cation.
- 9/12/20 Initial discussion with University of Florida team about alternative obfuscation techniques and quantifying RTL security.
- 9/16/2020 Created reinforcement learning model for ML-based ASIC mapping
- 9/22/2020 Added functions to store required timing for each PO in the network data structure
- 9/24/2020 First CNN model finished for ASIC mapper
- 9/28/2020 Added obfuscation functionality to allow LSOracle to ignore selected nodes, preventing optimization from harming obfuscation
- 9/29/2020 Meeting with Prof. Swarup Bhunia at University of Florida on hardware security collab- oration
- 9/30/2020 Implemented RDD backend in Yosys for cloud-based LSOracle
- 10/1/2020 Began implementing native equivalence checking in LSOracle, improving verification flow
- 10/2/2020 Finished modifications to libABC to allow use of CNN model for ASIC mapping
- 10/6/2020 Added command to set delay target for POs
- 10/8/2020 Initial experiments with CNN for ASIC mapping show model is unstable, begin retrain- ing to address
- 10/13/2020 Retraining CNN complete. Obtain 10% delay improvement over ABC in arithmetic circuits by using CNN based technology mapper

- 10/16/2020 Follow up meeting with University of Florida hardware security team to discuss exist- ing tools and how to integrate them into LSOracle
- 10/19/2020 Second revision of CNN: Both area and delay improved compared to stock ASIC mapping, average 15% improvement in ADP over benchmark suite
- 10/22/2020 PNNL expresses interest in using LSOracle in high level synthesis flow
- 11/05/2020 LSOracle team participates in WOSET 2020, runner up for both best contribution and best video
- 11/09/2020 Began work on containerizing PandA/Bambu flow from PNNL to allow easier integra- tion with LSOracle and OpenROAD
- 11/12/2020 Received real SDC from BlackParrot team
- 11/17/2020 Finished containerizing PandA/Bambu
- 11/19/2020 Began collaboration with Alan Mishchenko at UC Berkeley to improve FPGA mapping algorithm
- 11/30/2020 Made roadmap for integration of LSOracle with OpenFPGA
- 12/01/2020 Restructured Github repository to enforce better project management
- 12/02/2020 Experiments on new area recovery algorithm for FPGA mapping gives improvement on several benchmarks
- 12/03/2020 PNNL team gives us benchmark suite for PandA/Bambu
- 12/06/2020 Migrated LSOracle CI/CD from Travis to GitHub Actions
- 12/09/2020 Improved FPGA mapper shows improvement on VexRISCV compared to Symbiflow
- 12/10/2020 Cloud based LSOracle flow shows comparable runtime to ABC when using mixed synthesis
- 01/21/2021 Cloud-based LSOracle tested with updated version of BiPart: cloud-based runtime reduced by 20%, and QoR improved by 70% compared to previous BiPart partitioner
- 01/28/2021 XMG support added to LSOracle
- 02/02/2021 LSOracle - OpenROAD integration tested using Ibex design on Skywater 130 technol- ogy; 9% reduction in delay achieved
- 02/04/2021 LSOracle - OpenFPGA integration tested by generating bitstreams for SOFA test chip, resulting in 6% reduction in CLB usage

- 02/09/2021 Binary packages added to continuous integration flow, allowing users to install an up to date version of LSOracle without needing to compile source code
- 02/10/2021 Video tutorial added to the LSOracle documentation
- 02/18/2021 1st version of MIG-based logic locking implemented
- 02/24/2021 Meeting with the OpenROAD team; begin benchmarking to show area-delay pareto frontier using LSOracle
- 03/04/2021 OpenSTA support added to native AIG datastructure
- 03/08/2021 First benchmarks with PandA/Bambu and OpenROAD generated, showing benefits of LSOracle in HLS - GDS flow
- 03/10/2021 Roadmap for physical synthesis support created
- 03/15/2021 First version of timing-driven LSOracle complete; 50% reduction in delay achieved over test benchmark with three iterations
- 03/18/2021 Received Verific library through DARPA toolbox program
- 03/22/2021 Agreement to bring LSOracle into OpenROAD master branch after benchmarks show benefits. Pull request made
- 04/01/2021 Updated OpenROAD pull request
- 04/02/2021 Began integrating LSOracle with Verific through DARPA toolbox
- 04/20/2021 Verific integration complete
- 04/22/2021 First results for structure aware partitioning. 5-15% improvement in technology inde- pendent metrics
- 05/05/2021 Final data gathered for structure aware partitioning; approximately 6% improvement over previous version of LSOracle after technology mapping
- 05/11/2021 Developing native technology mapping
- 05/12/2021 First documentation updates merged
- 05/18/2021 XAG and XMG mixed synthesis support added
- 05/20/2021 Updated OpenROAD pull request
- 05/23/2021 Second documentation updates merged
- 05/24/2021 CICD flow updated to use LSOracle Yosys plugin, include larger benchmarks
- 06/01/2021 Began refactoring LSOracle to use newest EPFL libraries

- 06/02/2021 Updated KaHyPar to latest version
- 06/03/2021 Patched mixed synthesis to improve runtime
- 06/08/2021 Updated OpenROAD CICD flow to use LSOracle plugin
- 06/15/2021 Merged LSOracle and Yosys plugin into one repository
- 06/18/2021 Began adding RTLIL support to LSOracle mapper
- 06/22/2021 Completed refactor of all core LSOracle functionality
- 06/28/2021 Added new FPGA mapper to LSOracle executable as experimental command
- 06/30/2021 LSOracle merged into OpenROAD repository as a submodule
- 07/01/2021 First benchmarks of refactored LSOracle complete, show 10-100× improvement in runtime
- 07/06/2021 Development begins on new optimization recipes for refactored LSOracle
- 7/14/2021 Refactor shows average 20-30% delay improvement over ABC post technology map- ping
- 07/20/2021 All libraries moved to submodules
- 07/29/2021 Added automatic visualization to CICD suite, expanded automatic benchmarks
- 08/02/2021 Meeting with EPFL team to collaborate on technology mapping;
- 08/04/2021 Added selfhosted runner to CICD flow to allow testing integration with commercial tools
- 08/11/2021 Development begins on integrated timing-driven synthesis
- 08/18/2021 Added XMG support to refactored codebase
- 08/23/2021 Added improved XAG, AIG recipes
- 09/08/2021 Added tunable area/delay optimization in support of timing-driven synthesis
- 09/13/2021 Timing driven synthesis flow in place
- 09/15/2021 Added new resubstitution and rewriting algorithms
- 09/16/2021 Integrated automatic recipe generation with multi-armed bandit
- 09/22/2021 Verified correctness of new LUT mapper on benchmark with 128 LUT4s
- 09/28/2021 Timing-driven synthesis flow demonstrated with placeholder optimizations, demon- strating that glue code is working

- 09/30/2021 Automatic recipe generation shows lower LUT count than area-oriented ABC mapping script
- 10/04/2021 Native LUT mapping begins testing within LSOracle, rather than as standalone code
- 10/07/2021 Name support added to all mockturtle optimizers used by LSOracle
- 10/13/2021 CICD flow upgraded to support multiple recipes for each DAG
- 10/18/2021 PicoRV passes formal verification after timing-driven synthesis flow
- 10/21/2021 Implemented area flow and exact area heuristics in native LUT mapper, set min delay and set dont touch support, and initial boolean division algorithm, showing 2-3% improvement in area and delay
- 11/30/2021 Merged native tech mapper into master branch
- 12/02/2021 Updated Panda-Bambu integration
- 12/04/2021 - 12/10/2021 Attended Design Automation Conference
- 12/14/2021 Added cut deduplication to native LUT mapper
- 12/16/2021 Generalized area LUT mapper for AIGs merged into LSOracle
- 01/10/2022 Automatic recipe generation using multi-armed bandit supervised learning approach implemented in LSOracle branch.
- 01/13/2022 Generalized area LUT mapper working on MIG networks
- 01/27/2022 Refactored partition view code to use window paradigm within EPFL libraries
- 02/10/2022 Completed pre-techmapping comparison with commercial tools, verifying that LSOr- acle matches or exceeds their performance
- 02/16/2022 Receive initial feedback from RapidSilicon on their LSOracle testing
- 02/22/2022 Patches to EPFL libraries complete, timing driven resynthesis functional
- 02/24/2022 Provided customized LSOracle scripts to RapidSilicon
- 02/25/2022 Began work on improving reintegration of results in Yosys plugin
- 03/01/2022 Completed advanced usage guide as requested by RapidSilicon
- 03/03/2022 Finished initial testing of generalized delay model in native LUT mapper
- 03/09/2022 Timing driven mode beats ABC by 85% on EPFL adder benchmark

- 03/11/2022 Began work on adding adder optimization to LSOracle in collaboration with Symbi- flow team
- 03/15/2022 Pull request for multi-armed bandit automatic recipe generation created in LSOracle

4 METHODS AND PROCEDURES, RESULTS AND DISCUSSION

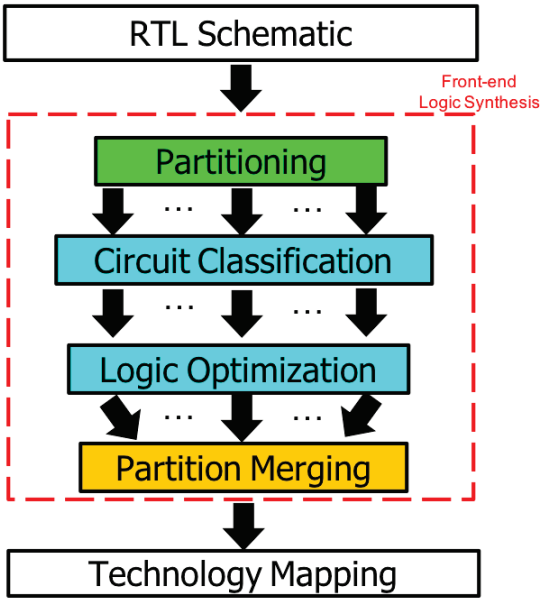

Figure 2: Flow Chart of the Alpha Version of LSOracle

4.1 M-1: Learning Based Circuit Partitioning

Milestone M-1 focused on the development of a circuit partitioning engine. Although the statement of work placed this milestone before task T-3, developing a data set generator, in practice, substantial work on T-3 was required for this task. In addition, work on task T-4, circuit merging, was closely related to circuit partitioning and was pursued in parallel, but completed for milestone M-2.

As shown in Figure 2, the framework (demonstrated during the first integration session) consists of four parts:

T-1: Circuit Partitioning, where we are using hMETIS to prove the feasibility of our logic synthesis concept. The partitioning algorithms will be revisited after a careful analysis of the framework performance.

T-2: Circuit Classification, where we have implemented a CNN-based circuit classifier capable of predicting the proper optimization methods (either AIG- or MIG-based) for a logic cone with an accuracy of 79.27%.

T-3: Training data sets, where we have collected 8,237 circuit samples, including circuit partitions from the EPFL, MCNC and ISCAS'85 benchmark sets. The data set has been used in the training of our circuit classifier in T-2.

T-4: Partition merging, where we are developing the merging engine to assembly optimized par-

titions guided by our circuit classifier.

4.1.1 T-1: Develop Circuit Partitioning Algorithms

We employ the graph partitioner hMETIS [1] as an external library. hMETIS can produce partitions with similar sizes, which helps sizing our neural network classifier. To interface hMETIS, we have developed:

- a hypergraph generator, which transforms an input logic network into a hypergraph file, we consider *And-Inverter Graph* (AIG) as the input logic network format since its parser is natively supported in lstools. More parsers will be built upon the needs from benchmark circuits. The hypergraph representation considers the grouping of nodes that share direct connections with each other as one hyperedge. For example, if a node's output is the input to two other nodes, those three nodes belong to the same hyperedge.

- a partitioner mapper, which parses the partitioning results and annotates the logic network. Note that hMETIS only outputs a file containing the index of the partition that each node in the network belongs to.

4.1.2 T-2: Develop Learning-based Circuit Classifier

The circuit classifier consists of two parts: (1) a truth-table generator, which translates the logic network to its equivalent truth table representation; and (2) a neural network classifier implemented with the Tensorflow C library.

Truth-table Generator In order for a neural network to classify circuits, efficient representation of a logic network that can reveal it logic features is critical. We select truth table representation as it is one of the most straightforward ways to describe the functionality of a logic network. Therefore, we have implemented a truth table generator which can create the truth tables for each circuit partition.

Algorithm The pseudo code for this algorithm is detailed in Algorithm 2, where the parameters are the AIG network (*aig*) and the current output of the partition (*root*). For each output node (*root* node), a truth table will be extracted. Our algorithm consists of following steps:

- First, we create a priority queue that stores logic nodes in a topological order, in order to ensure that the connections most critical to the output are added first.

- Then, we traverse the graph structure in the sequence of the priority queue, using a *Breadth First Search* (BFS) approach, and merge the truth table of each node visited.

The truth table of every node at a level is merged before moving to their children. During the merging, we consider all the truth tables coming from each fan-out, since high fan-out nets are common in practical benchmarks. As shown in Figure 3 for a logic node B with two fan-outs, two truth tables ($B, C|E$) and ($A, B, E|F$) are merged. Note that, some cases in the merged truth table are contradicted, e.g., ($B, C|E$) = ($0, 0|0$) and ($B, C|E$) = ($0, 1|0$) require B to be 0 while ($A, B, E|F$) = ($1, 1, 0, 1$) require B to be 1. Our algorithm will remove the lines that

will be always false, such as the deleted lines shown in Figure 3. Note that either the onset or offset of the current node in the traversal can be derived depending on the truth table of each node (represented by *wantedOut* in Algorithm 2). During the merging phase, the onset or offset of the current node is altered so only accurate data is merged with the output node's truth table.

Figure 3 shows a simple example of extracting the truth table of a 3-input logic network. In the Step 1 of this example, the truth table of *D* is merged with the truth table of *F*. Before Step 2, our algorithm will check if the truth table generated by Step 1 has to be altered in order to resolve contradicting values for the node *B*, as shown in *Step 2a*. In Step 2, truth table merging is applied as Step 1 does.

Extracting the truth tables may cause scalability issue and more efficient compressed representation can be developed. However, the truth table is the basis, from which other representations can be easily derived. Therefore, developing the truth table generator is a necessity from a technical perspective.

Neural Network Classifier

We have written the code for a basic convolutional network that should be able to classify the network partitions based on their logic attributes between two different classifiers. The neural network that we built uses a two convolutional layer network, using max pooling as our discretization process. The code is untested on our data, as we are trying to determine the best way to bring our data into the network and also how to limit our input data size.

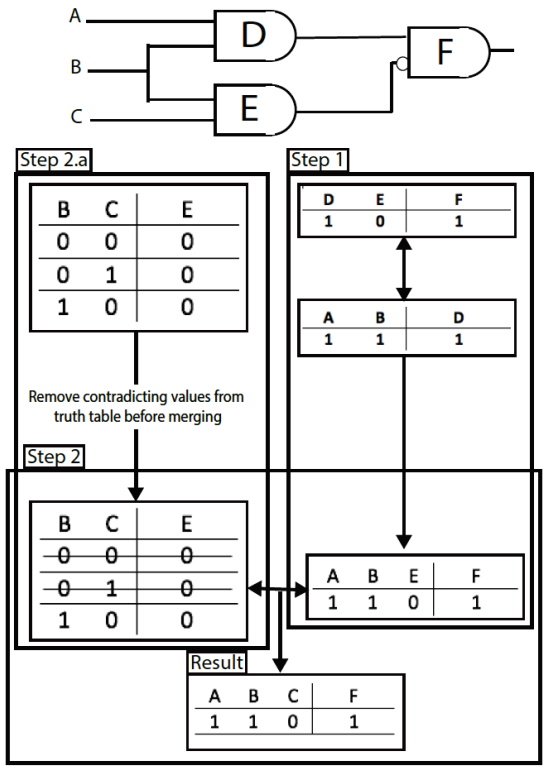

Figure 3: Example of Merging Truth Table with a Forked Connection

Algorithm 2 Detailed algorithm of the truth table generator

function *ADD TO TRUTH*(***aig***, ***root***)
Set all nodes to not visited ***q*** =
new Queue ***q****.enqueue(output*
node) **while** ***q*** *is not empty* **do**
n ← ***q****.dequeue()*
wantedOut ← *aig.wantedOut[****n****]*
if ***n*** *has not been visited* **then**
*visited[****n****]* ← **true**
for *every input (****n****,* ***child****)* **do**
if *child has not been visited* **then**
q*.enqueue(****child****)*
merge truth table of ***n*** *for* ***wantedOut*** *update* ***wantedOut***
for children of ***n***
else
merge truth table of ***n*** *for* ***wantedOut*** *with forked connection*
flag
update ***wantedOut*** *for children of* ***n***

We have implemented a circuit classifier based on a compact Convolutional Neural Network (CNN) using the TensorFlow library. Validated by our testing data sets (additional testing still running until completion of the task), the classifier achieves an accuracy of 79.27% in predicting if a circuit partition is best to be optimized with AIG- or MIG-based heuristics. To enable this,

our technical work includes three parts: circuit partition representation, circuit classifier topology, and training data set.

CNNs have shown best performances in classifying features in images of constant sizes. To use a neural network for recognizing the low-level logic attributes of a circuit, we partition them into logic cones, each of which representing a multiple-input and single-output logic network. We then extract the Karnaugh maps of the logic cones and represent them as images, where each "pixel" represents the Boolean output under a given input patterns, as exemplified in Figure 4. We call this representation KMI, which stands for Karnaugh Map Image.

ABC \ DE	000	001	011	010	110	111	101	100
000	1	1	1	1	1	1	1	1
001	1	1	0	0	0	0	1	1
011	1	1	1	1	0	0	0	0
010	1	0	0	1	1	0	0	1
110	1	0	0	1	1	0	0	1
111	1	1	1	1	0	0	0	0
101	1	1	0	0	0	0	1	1
100	0	0	0	0	0	0	0	0

Figure 4: Karnaugh Map Image Representation

Using the KMI as inputs, we have successfully implemented the classification of our circuit partitions using the Tensorflow framework. The classifier is able to predict whether an AIG-based opti- mizer is more efficient than an MIG-based optimizer or vice versa for a given logic cone. Our topology, shown in Figure 5 is based on the MNIST-handwriting model since our input KMI are also black and white and, at this stage, we also have a small number of classes (= 2, denoting either AIG- or MIG-based op- timizer) that we are classifying into. As CNNs require the input images to be formatted with the same dimensions, we set the input sizes of the circuit partitions (logic cones) to be no more than 16. Note that a 16-input logic cone leads to a 256×256 pixels KMI, which fits well within the resolution of modern image data sets. Our dataset used for training and testing consists of 8,327 KMIs whose best optimization method was determined by seeing which method improved the original circuit's area the most. This method resulted in about 64.7% of the images favoring the AIG-based optimizer and the remaining 35.7% favored the MIG-based optimizer. The overall accuracy achieved after training was 79.27% with the prediction accuracy for AIGs was 79.7% and the accuracy for MIGs was 78.3%.

4.1.3 T-3: Training Set Generation for AIG and MIG Based Optimizers

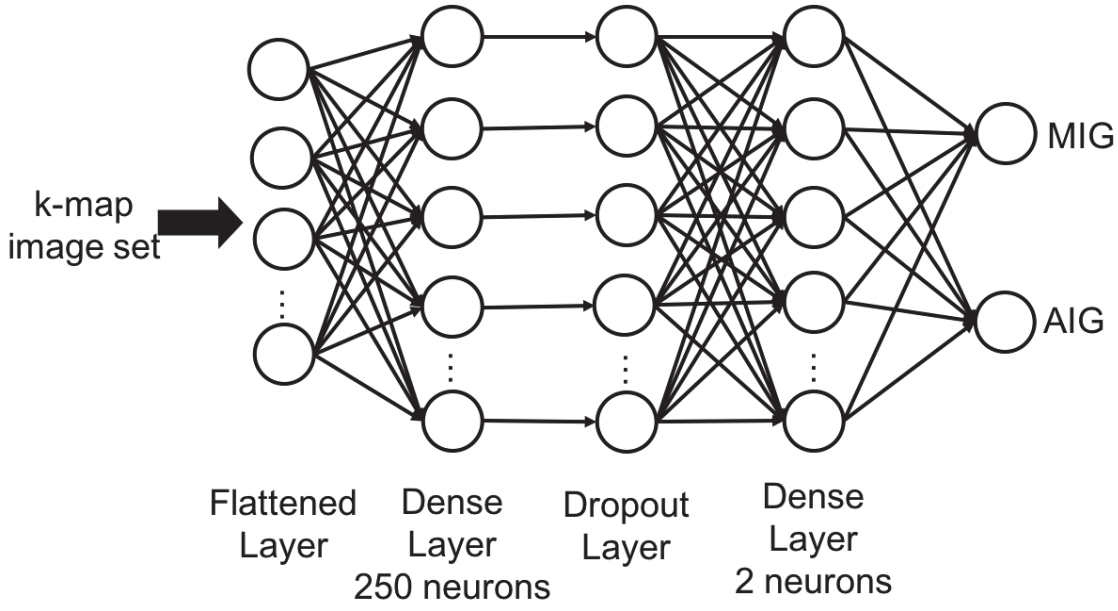

Figure 5: CNN Topology for LSOracle Classification

The training set is a necessary component of our circuit classifier. We have built a Python script which enables us to create a basic training set for the classification algorithm. Like all learning-based algorithms, the convolutional neural network requires a healthy amount of pre-labeled examples for the algorithm to learn. In our scenario, the pre-labeled examples are the ground truth that which one circuit optimization algorithm provide better performance, power and area (we consider area and performance from post-synthesis results). To automate the generation of the training set effectively, we have written a Python script that can process a large amount of benchmark examples through the optimization techniques of interest.

The Python script:

- generates optimization scripts to interact with ABC and Cirkit. Both scripts consider BLIF as input and output file formats. The ABC script optimizes the circuit with standard AIG methods (resyn; resyn2). The Cirkit script consists of three steps: (1) call ABC to convert them into AIG format which is the only acceptable file format for Cirkit; (2) read in the AIG file and perform the MIG algebraic optimization algorithm; (3) convert the AIG to BLIF format. Both scripts will print out statistics of optimized networks using command print-stats in ABC.
- executes the ABC and Cirkit scripts to apply logic optimizations.
- collects the statistics and evaluates the best logic optimization method for the provided network in terms of the number of logic nodes and logic levels.

We have tested the Python script with one-hundred circuit examples from MCNC and ISCAS'85

benchmark sets. The results are compared and stored in a spreadsheet (we are investigating how to in- tegrate it with an open-source database framework), showing that the AIG-based methods lead to better QoR in 70% of the circuits while the MIG-based methods perform better for the other 30%.

To provide similar circuit size for our circuit classifier, we have studied how to generate equal-size partitions from different benchmarks (by tuning the hMETIS parameters). Our statistical analysis show that using OpenCores benchmarks, on average a combinational logic cone consists of 15.73 inputs, which is in the scalable range of truth table images (16 inputs requires a 256×256 truth table image). We decide to use 16 inputs as the threshold in partitioning sizes, but further design space exploration will be performed.

We have organized the training sets (to fit different optimization targets) into three categories:

1. area-oriented, where area is the only factor of evaluation. We see that the AIG-based optimizer performs better in 64% of the ISCAS'89 benchmarks, the MIG-based optimizer is better in the rest 35%.
2. delay-oriented, where delay is the only factor in evaluation. We see that the AIG-based optimizer performs better in 55% of the ISCAS'89 benchmarks, the MIG-based optimizer is better in the rest 45%.
3. area-delay-oriented, where area-delay product is the only factor in evaluation. We see that the AIG- based optimizer performs better in 70% of the ISCAS'89 benchmarks, the MIG-based optimizer is better in the rest 30%.

To provide sufficient training samples for our circuit classifier, we have expanded our training sets to cover partitions generated from circuits (mainly from the EPFL benchmark set and ISCAS'89 - and worked on the more industrial relevant benchmarks with YOSYS making things harder from time to time with name changes, etc.): Our training set shows the following statistics over 11k training samples:

1. For area-oriented optimizations, we see that the AIG-based optimizer performs better in 49.7% of the partitions, the MIG-based optimizer is better in the rest 50.3%.
2. For delay-oriented optimizations, the AIG-based optimizer performs better in 55% of the partitions, the MIG-based optimizer is better in the rest 45%.
3. When both area and delay are targeted, the AIG-based optimizer performs better in 73.6% of the partitions, the MIG-based optimizer is better in the rest 26.4%.

4.2 M-2: Alpha Version of LSOracle

Due to working ahead of schedule on task T-3, milestone M-2 consisted of improving the code from milestone M-1 and completing circuit merging. No major technical challenges were encountered imple- menting the merging algorithm.

We have completed the alpha version of LSOracle, including optimization - after classification - and merge back, as shown in Figure 2. LSOracle has been released on Github and demonstrated during the integration session.

Our flow was tested using a small sequential toy example (s444) with 155 nodes, a logic depth of 12 levels, 24 inputs, and 27 outputs. This example was also optimized using AIG-based and MIG-based methods for optimization without partitioning for comparison. Immediate area and delay improvements in the Boolean network were observed along with area and delay improvements after performing FPGA tech mapping with ABC and ASIC tech mapping with Design Compiler. Table 1 shows strong improvements in delay optimization when using LSOracle over the other methods. While this example is small, it shows promising results for this methodology. We are planning on making improvements to our partitioning methods and our classification to expand our tool to larger real-world examples.

Table 1: Post-synthesis Results of LSOracles on a Toy Example s444

s444	ABC	LSTools	LSO 5 partitions	LSO 3 partitions
# Nodes	109	118	116	113
Logic Depth	9	7	11	11
# LUTs	31	37	30	30
LUT Depth	2	3	3	3
Arrival Time (ns)	54.3	54.37	51.45	50.95
Period (ns)	54.5	54.5	51.5	51

4.3 M-3: Beta Version of the Logic Synthesis Platform

Milestone M-3 consisted of releasing a beta version of LSOracle, featuring improved synthesis recipes and PPA, a Verilog frontend, and automatic, no-human-in-the-loop optimization. In the process of refactoring for improved PPA, we completed task T-8, integration of the AIG and MIG optimizers, ahead of schedule and included that task in M-3. In addition, later work on automatic recipe generation is included in task T-5, as it is a closely related effort, although this feature was not included in the beta release.

4.3.1 Single Command Interface, Beta Performance, Multithreading.

Creating the beta version of LSOracle involved extending LSOracle's features set to bring it from being purely a technology-independent logic optimization engine towards being a fully featured EDA tool with functionality along more of the EDA process. We integrated LSOracle into Yosys, giving LSOracle a Verilog parser and a native interface with ABC's Techmapper, implemented an automatic mode that runs with a single command, and added multithreading support to LSOracle to speed up the EDA flow.

No Human in the Loop Operation
One notable feature of the LSOracle beta is a single command called "oracle" that contains all of the steps in the flow from reading in a file, partitioning the network, performing optimization,

and writing out the resulting Verilog or BLIF files. If the number of partitions is not specified, LSOracle determines the number of partitions necessary to have about 300 nodes per partition. Along with being able to let the tool make all the decisions for the flow, we have also made the criteria for optimization customizable, in case the user requires node or depth specific optimization. The results for these strategies using the high effort mixed synthesis in LSOracle are shown in Figure 6and show that the results for node and depth product, node, and depth correlate for the strategy chosen, giving the user both a simplified and more flexible workflow.

Multithreading

The classification and optimization stages of our high effort flow have been multithreaded using OpenMP [2]. The partitioning and partition merging are still single threaded, taking the majority of execution time, however, our optimizations have led to an average 23% runtime improvement compared to our serial version in the benchmark suite. Figure 7 shows a selection of exploratory timing benchmarks, including some from EPFL, OPDB, and Titan23, demonstrating the runtime improvement when running multithreaded LSOracle on our server with 64 threads.

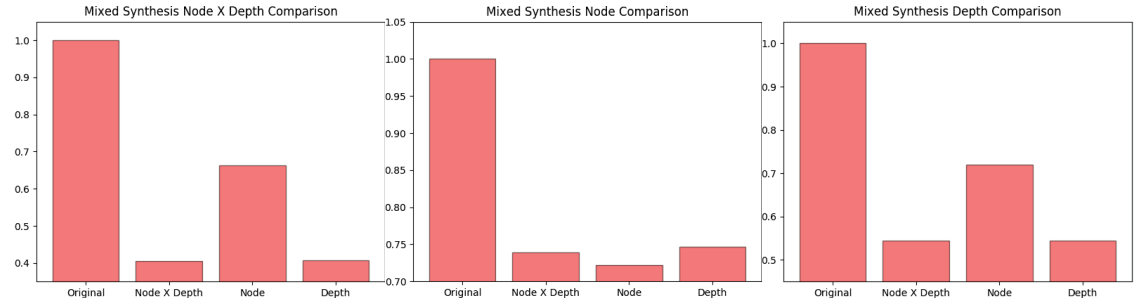

Figure 6: Comparison of User-selectable Optimization Approaches in Mixed-synthesis Flow

Users can now chose between a balanced approach (default, "Node × Depth"), or approaches prioritizing network size ("Node") or network depth ("Depth").

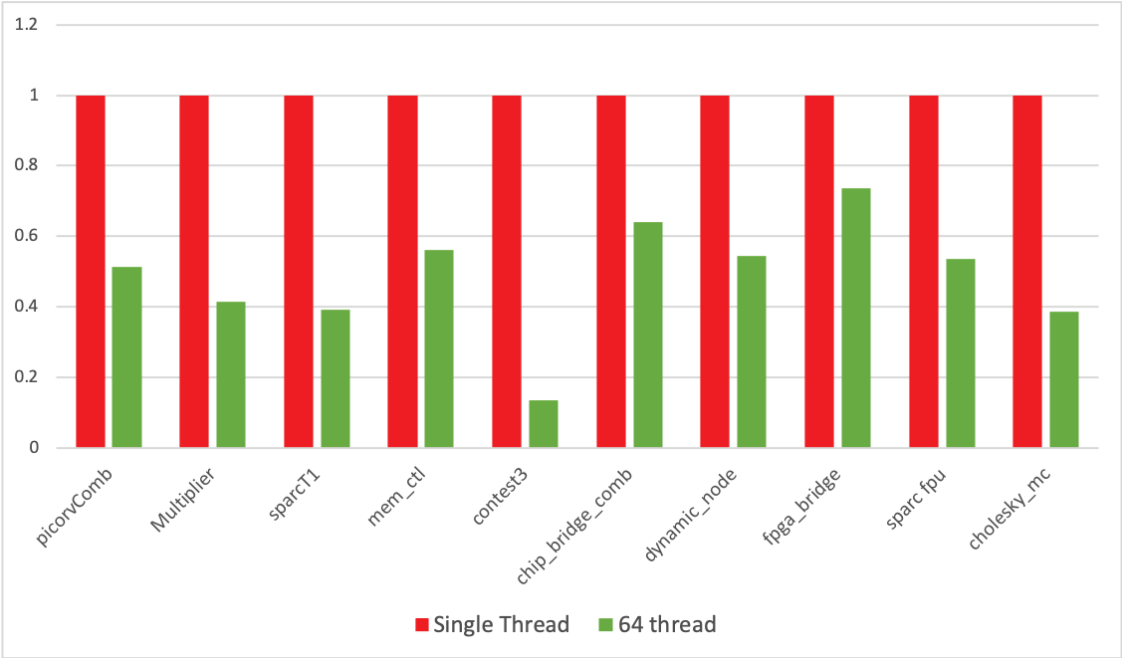

Figure 7: Single and Multithreaded Runtimes of LSOracle Across a Suite of Ten Circuits Selected from OPDB and Opencores

4.3.2 TA T-1: Verilog Frontend Integration

The Verilog parser in LSOracle is basic and limits the number of real-world benchmarks that can be optimized without pre-processing with Yosys and ABC. This requirement for external preprocessing was a hurdle to adoption at the integration exercise; users expect tools to read Verilog. To address this limitation, we decided to integrate the entirety of LSOracle into Yosys. We have completed the integration work and now all functionality from LSOracle is available within Yosys. This integration creates an end-to-end logic synthesis tool which utilizes the Verilog parser of Yosys, the logic optimization of LSOracle, and the Techmapping capabilities of

the Yosys-ABC integration.

In the first implementation, the LSOracle-Yosys integration required ABC as an intermediate step. Yosys uses BLIF as an internal format when it calls ABC, and LSOracle had poor BLIF support, so we used ABC to read the BLIF and generate an AIG file for LSOracle. To streamline the integration an AIG file for LSOracle. To streamline the integration and minimize computationally expensive calls to external tools, we updated LSOracle to use the newest version of the EPFL logic synthesis libraries, which add more robust BLIF support. Later in the project, we updated LSOracle to natively support Yosys' internal RTLIL data structure.

Although Yosys' Verilog parser is likely the most complete open-source tool, it still only supports a subset of the Verilog standard. To address this issue, we integrated Verific, a commercial parser, with LSOracle.

We finished integrating Verific from the DARPA toolbox with Yosys in May 2021, and have received permission from Verific to distribute the code we have written publicly. This will allow other DARPA performers participating in the DARPA toolbox program to easily integrate with Yosys by simply cloning a public repository, rather than having to contact us for code access. The Yosys Verific integration in Yosys' master branch is specific to the versions of Verific used in Symbiotic EDA, the commercial version of Yosys. We changed the integration to remove the parts unavailable through the DARPA toolbox, such as the VHDL frontend, and to remove anything specific to Symbiotic. For simplicity, we are integrating the new Verific Yosys frontend with the LSOracle Yosys plugin, so DARPA performers using LSOracle will have a full featured Verilog parser by default. To test the integration, we synthesized several OpenPiton Design Benchmark designs which previously could not be parsed due to multiple edge sensitivity.

4.3.3 T-5: Automatic Generation of Optimization Recipes

LSOracle aims to partition a design to optimize and synthesize different portions with specialized data structures for logic optimization and manipulation, i.e., AIG, MIG, XMG, XAIG, etc. To do so, we rely on pre-defined recipes that have as goal to optimize area or delay, depending upon the chosen data structure. Recent works have shown that recipes can be made more efficiently if they are specialized for a given design, i.e., there is not a single recipe that performs the best across different designs 1 2 3. To this end, Machine Learning techniques have been employed to define the right sequence of optimizations and build a design specialized recipe.

A multi-armed bandit approach for automatic flow generation (FlowTune) has been developed and fully integrated and validated in LSOracle. To compare our machine learning (ML) based approach, we used the current LSOracle optimizations recipes, and created a new one to be the new baseline, which outperforms previous LSOracle recipes. Our ML-based approach generates recipes with the same number of commands as the baseline recipe. Thus, we can ensure that we beat baseline because we generate a better flow, and ensure that is not because we simply run more rounds of optimization.

Results are extracted post-Placement and Routing (PnR), targeting a modern Intel Stratix IV-like

FPGA architecture, on top of OpenFPGA framework. We use six designs from VTR for evaluation.

Area Results: We improve total area (logic area + routing area) in all the cases - Figure 8. In the best case, we improve area by 10%. In the worst case, area is improved by 1%. On the average, we achieve 4.5% area reduction.

[1]C. Yu, "FlowTune: Practical Multi-armed Bandits in Boolean Optimization," 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD), 2020, pp. 1-9

[2]C.Yu,H.Xiao,andG.D.Micheli,"Developing synthesis flow without human knowledge," in Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018, 2018, pp. 50:1–50:6.

[3]C.Yu,H.Xiao,andG.D.Micheli,"Developing synthesis flow without human knowledge," in Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018, 2018, pp. 50:1–50:6

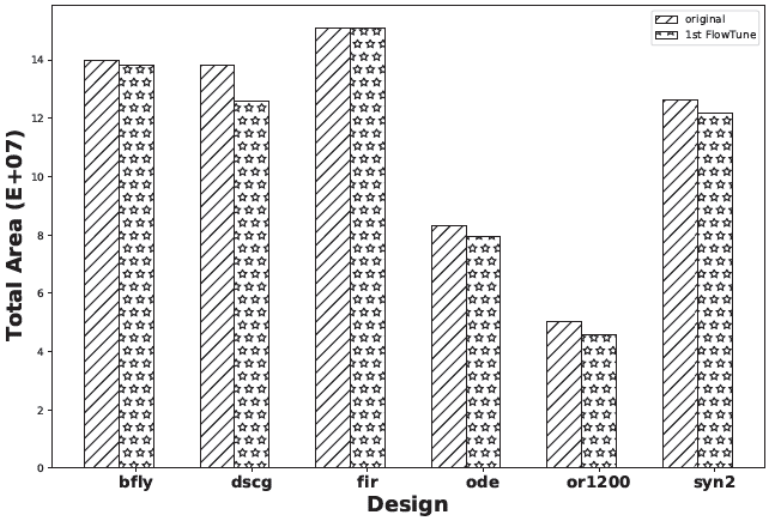

Figure 8: Area results

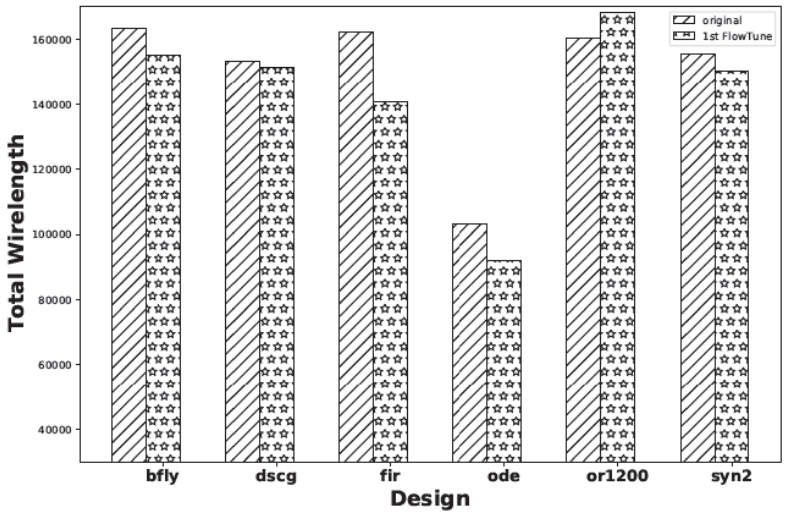

Figure 9: Routing wirelength

Routing Results: Total routing wirelength is presented in Figure 9. We improve five cases, with up to 14% reduction. On average, the reduction is about 5%, with small overhead in one design.
Timing Results: Worst Negative Slack (WNS) refers to the critical path delay of the design, and our new ML-based flow improves WNS in all the cases, as observed in Figure 10. Our gains are up to 37% for the or1200 design, with an average of 13% (5 cases have over than 7% improvement in WNS). As for

With Total Negative Slack (TNS), which refers to the timing of the whole design instead of only looking to the critical path, we improve five out of six cases, as observed in Figure 11 In the case we do not improve, our overhead is of only 1%. For the remaining designs, we achieve up to 14% TNS reduction, with an average of 5% TNS reduction.

4.3.4 T-6: Dataset Generator for Optimization Recipes and Network Training

To help guide partitioner and optimizer development, we have created gold standard benchmarks that contain arithmetic and control logic portions that we have manually partitioned. This allows us to tune parameters in the partitioning algorithm to recover known-good results on a small scale, speeding development, and functions as a sanity test for our partitioning algorithms. Figure 12 shows the first of these benchmarks, a 4-bit ALU with controller.

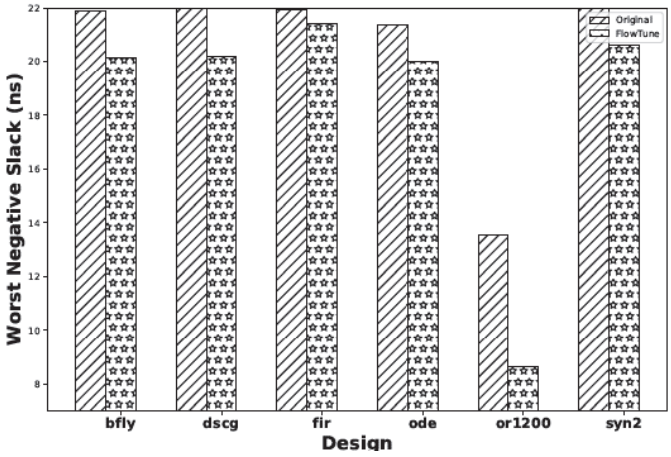

Figure 10: Worst Negative Slack

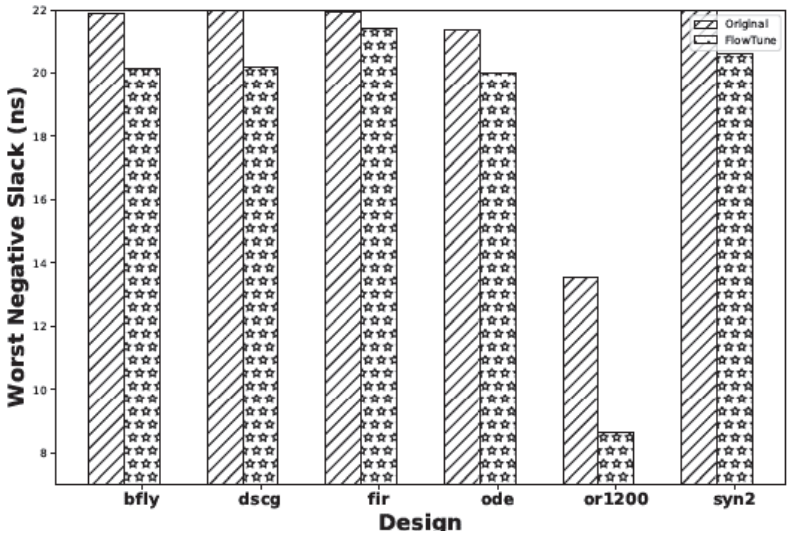

Figure 11: Total Negative Slack

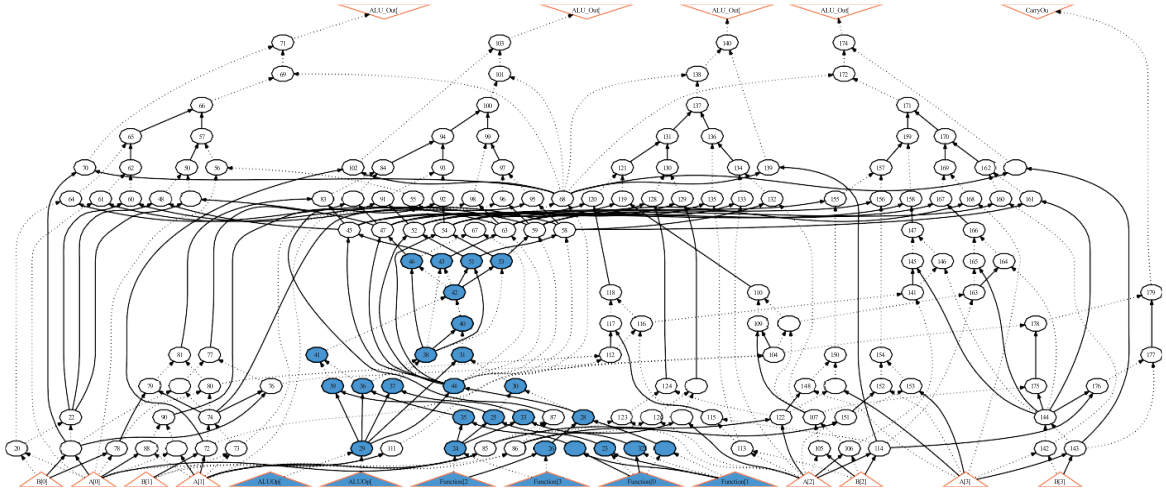

Figure 12: Simple Gold Standard Benchmark
A 4-bit ALU and controller (shaded blue), manually partitioned into arithmetic and control portions.

In addition, we created two large edge case benchmarks, a 2048-bit ripple carry adder, and a 4096 state FSM to test arithmetic and control logic, respectively. Table 2 presents the detailed results of opti- mizing those circuits: the performance of LSOracle in AIG, MIG, mixed synthesis high effort, and mixed synthesis high effort with partition merging compared to ABC resyn2 on two edge case benchmarks, a 2048-bit full adder, expected to perform well under MIG optimization, and a 4096 state FSM, which should perform better under AIG optimization. Columns are network size in terms of nodes (AND nodes for original network, ABC, and LSO AIG, MAJ3 for all others), network depth, and the product of nodes times depth, normalized to the original network.

For the large adder, LSOracle mixed synthesis with merged partitions duplicated the performance of MIG optimization on the unpartitioned network, reducing logic levels from 6142 in the original network to 17 after optimization, with a corresponding 99.8% reduction in nodes × depth. The large FSM had less dramatic results, with most optimization methods yielding a reduced network size at the cost of network depth leading to a node × depth product higher than the original network. An additional problem is that the large finite state machine showed a relatively small improvement even under the best available optimization approach, ABC resyn2. This small available room for improvement makes comparison between different optimization methods difficult, so we chose the SPARC IFU from OPDB as a large, real-world example where we expect AIG optimization to outperform MIG optimization. As shown in Table 3, a slightly modified version of resyn2, using drf and drw for stability reasons, reduced the product of nodes and depth by approximately 65% compared to the original network, outperforming MIG optimization, as expected, and providing a much larger potential signal to compare optimization ap-proaches. Unfortunately, the network, at more than 2 million nodes, is too large to run in LSOracle mixed synthesis with the current partitioner. We explored ways to reduce this size without limiting the file's utility as a benchmark, or else to improve memory management in the LSOracle partitioner to allow larger files.

To improve LSOracle's performance on control logic, a first short term step is to implement an

AIG op- timization recipe equivalent to ABC's resyn2, which outperformed LSOracle's AIG optimization recipe, which is based on ABC's rw command. This is addressed in more detail later. A longer-term issue to address is the apparent poor performance of our classification method on control circuits, since the mixed synthesis optimization was not equivalent to the AIG optimization, indicating that individual partitions were being classified for MIG optimization. This could be as simple as the less advanced AIG optimization recipe occasionally being outperformed by the MIG optimizer, or it could be a more fundamental limitation of the nodes × depth criterion used in high effort mode.

Table 2: Performance of LSOracle

Method	2048 Bit Adder			4096 state FSM		
	nodes	depth	norm. product	nodes	depth	norm. product
Original Network	22521	6142	1	27779	18	1
ABC resyn2	16379	4095	0.485	25746	18	0.927
LSO ABC	16380	4096	0.485	26888	21	1.129
LSO MIG	17104	17	0.002	27842	19	1.058
LSO High Effort	14422	526	0.055	27655	29	1.604
LSO Merged	17104	17	0.002	27803	21	1.168

Table 3: Performance of AIG and MIG Optimization on Two Control Logic Benchmarks, where AIG is Expected to Outperform MIG

On the left, for comparison is the 4096 state FSM we presented.

4096 state FSM				SPARC IFU		
	nodes	depth	norm. product	nodes	depth	norm. product
Original Network	27779	18	1	2009904	531	1
ABC resyn2	25746	18	0.927	1957839	193	0.354
LSO MIG	27842	19	1.058	1784630	341	0.570

4.3.5 T-7: Fine Integration to Achieve PPA Improvements at Logic Synthesis Level Improved AIG Optimization Recipe

In our discussion of the pathological benchmark suite, we noted that the AIG optimization recipe imple- mented in the EPFL logic synthesis framework is relatively primitive compared to the resyn2 and resyn2rs recipes available in ABC. We implemented an equivalent to resyn2 in LSOracle. We have updated our AIG optimization recipe to include AIG refactoring alongside the existing rewriting and implemented an AIG balancing algorithm. Once this was complete, we interleaved refactoring, rewriting, and balancing in our optimization recipe to build an equivalent

to resyn2.

Following up on the addition of refactoring to our AIG optimization recipe, we added balancing and updated the recipe to be equivalent to ABC's resyn2 in terms of number and order of steps. Compared to the original AIG recipe, the updated version had substantial improvements; however, compared to the version which did not include balancing, the effect was primarily to trade node count for reduced depth.

We then explored ways to close this gap between LSOracle's AIG optimization recipe and ABC's resyn2. When comparing each step of the optimization, we saw that balancing, refactoring, and rewriting in LSOracle were 0.448% worse, 24.9% worse, and 9.20% respectfully compared to ABC on a set of 10 benchmarks from the OpenPiton, OpenCores, and EPFL suites. To close the overall gap, we focused on improving refactoring and rewriting. We updated the rewriting command to perform repeated rewriting until there are no more gains in network size, at which point we found performance comparable to ABC's rewriting command. We were able to improve refactoring slightly by using NPN resynthesis, but it still remains about 20% worse than ABC's refactoring method in isolation. When all three are interleaved, however, we are able to obtain results that are within 0.5% of resyn2 with respect to the node and depth product as shown in Figure 13. We improved the refactoring implementation in LSOracle to exceed resyn2's performance on AIG networks, and explore ways to reduce the number of rewriting steps needed in order to improve run time.

Merging Adjacent Partitions to Recover Global Optimization

We have successfully implemented an optimization technique that combines adjacent partitions of the same type. By splitting a network, it is possible to lose some global optimization. In theory, by merging partitions that share connections that are classified to be optimized using the same method (AIG or MIG), we mitigate this loss. For homogeneous synthesis modes in LSOracle, i.e., either AIG or MIG alone, merging the partitions recovers the original network, removing all partitioner overhead. This simplifies the workflow for the user because now for all syntheses, the control flow is to read the AIGER file, partition, and optimize. The main purpose of partition merging, however, is to improve performance in mixed synthesis mode. So far, in comparison to our earlier approach of optimizing smaller partitions independently, we see further improvement with our high effort method of classification, i.e., trying both techniques and choosing the one that produces the best results. Because of the 80% accuracy of the classifier, low effort mode does not see an improvement, since even a relative few misclassified partitions can break up the larger partitions and remove the benefit. We expected to see gains in low effort mode as the classifier improved. Figure 14 shows the performance gains realized by partition merging. The metric is number of LUTs ×Depth, and the test suite is the selection of circuits shown in Table 1, normalized to the original network size. High effort mixed synthesis mode results in an approximately 18% improvement compared to low effort mode, and an approximately 22 % improvement compared to the original circuit. Because of the composition of the benchmark, MIG with partition merging performs best overall, but for individual circuits in the benchmark with heterogeneous control/arithmetic logic, mixed synthesis remains the top performer.

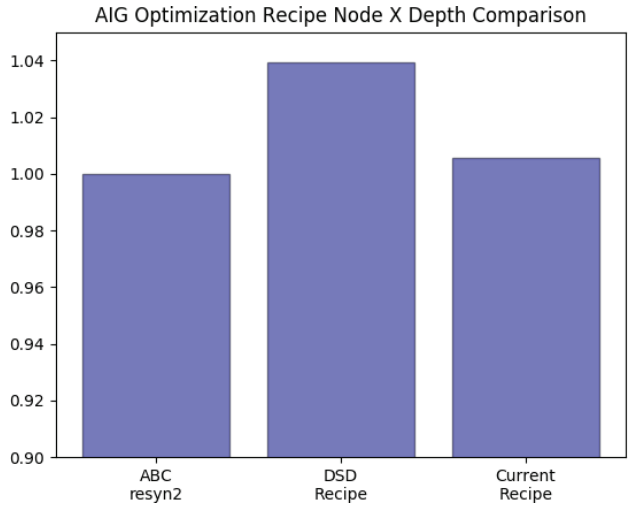

Figure 13: Comparison of Current AIG op- timization Recipe vs the Recipe and ABC's *Resyn2*

The current optimization recipe is now within 0.5% of resyn2's performance in terms of nodes x depth.

Partition Optimization Grader

Our circuit classification techniques are applied to logic cones, where each logic cone is represented through a *Karnaugh Map Image* (KMI). To do so, first, we need to retrieve each logic cone truth table. Given the exponential complexity for truth table generation, large logic cones are not converted into KMI, and therefore not assigned to either AIG- or MIG-based optimizers. Note that optimizing large logic cones is of paramount importance since they are more likely to lead to further global optimization. In this sense, to avoid discarding cones that have a considerable influence in the circuit, we developed a heuristic. This heuristic works as a grader for logic cones that cannot be converted into a KMI, and works as follows:

$$score = \sum_{i=1}^{m} (W_{ni} * N_i) + (W_{di} * D_i) \quad (1)$$

Where m is the number of cones in the partition, W_{ni} is the weight given for nodes, N is the number of nodes, W_{di} is the weight given for logic depth, and D is the logic depth in the i^{th} cone. The node's weight is given as follows: in case the number of nodes in the considered cone is greater than the average number of nodes in the cones of the same partition, the weight is assigned to be 1.5. For the depth, we have an incremental grading scheme: if the depth of the i_{th} cone is greater than average, we assign the weight to 1.3. If it is greater than the average + 1, it is assigned a weight of 2. If it is greater than the average + 2, its weight is set to 3. Such incremental grading is designed to give weight to circuit performance rather than to area. Through this grader, LSOracle is capable of optimizing large partitions that cannot be classified, but still, play an important role for the final circuit optimization.

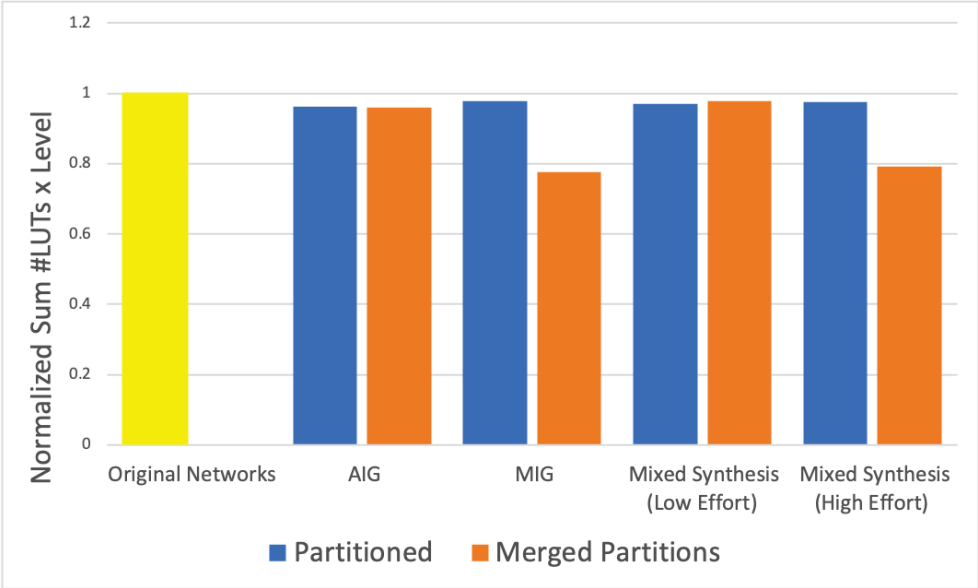

Figure 14: Partitioner Performance: Separate Partitions vs Merging Adjacent Partitions of the Same Type

4.4 T-8: Integrate AIG and MIG Optimization Engines

AIG and MIG optimizers were integrated into LSOracle and regression tests were launched to cover both benchmarks which were working in the pre-refactored version as well as new benchmarks.

We have refactored the LSOracle code base (following the LStools modularity standards) with the purpose of (1) easily integrating various partitioning algorithms and (2) supporting generic hybrid logic networks between MIG/AIG/*etc.* In the alpha version, the partitioning engine had two major limitations: First, partitioning was done externally by calling the HMETIS binary, where hypergraph files were out-putted and read back in the tool (hitting some I/O speed limitations). Second, hypergraph generation partitioning supported only AIG network, where other types of networks had to be converted. The integrated framework can reduce the runtime and I/O access, leading to a stable modulized code base for future development.

Integration of Circuit Partitioner We have developed new C++ class called partition_view and partition_manager:

1. The partition_view is an individual partition and can be thought of as a window that focuses on a specific section of a network. The input, output, and gate nodes of a partition are stored in this class so it is able to be treated as its own network type and therefore can be run through other algorithms in the tool the same way as a normal AIG or MIG network. The nodes in this class, however, are still mapped to their respective nodes in the original network. This way, if a partition is optimized, the structure in the original network can be changed as the same time as the partition_view.

2. The partition_manager performs the partitioning. The network to be partitioned along with a number of partitions desired is given to the partition manager and it determines what partition each node belongs to and maps them accordingly. This is done with an open-source hypergraph partitioner that has been implemented in our framework. The manager can then create partition_view from this mapping. By handling partitions this way, nodes of the original network are only duplicated when their respective partitions are being viewed or altered. This brings significant runtime and memory efficiency when dealing with logic networks with millions of nodes.

Integration of Circuit Classifier After testing and investigation, it was determined that without code addition to the Darknet library, we would not be able to get access to the same CNN framework capabilities than before. Instead an open source C++ library [**3**] was used to convert the trained Tensorflow model into a json file and be able to perform the same classification commands as Tensorflow from within the LSOracle tool. A reminder that this trained model has an accuracy of about 79% with an 80% accuracy for AIGs and 78% accuracy for MIGs. Modifications were made to the partition manager to create compatible input KMImages so no files need to written or read out by the tool in order to perform this classification. The full framework flow has now all been implemented internally in the tool: partitioning, classification, optimization, and merging so no other programs are needed to use our proposed logic synthesis flow. Large benchmarks from the EPFL benchmark suite as well as the pico-rv benchmark have been tested using this flow. Testing was also conducted to test the performance and efficiency of the LSOracle tool in order to determine areas in need of improvement.

4.4.1 Testing: Q2 2019

In April of 2019, we ran an initial set of regression tests on LSOracle comprising ten benchmarks from MCNC, EPFL, IWLS'05 and practical projects. In general, we found performance for the mixed synthesis mode fell in between the partitioned AIG and the partitioned MIG results. Table 1 shows a selection of results from this test demonstrating the performance of our mixed synthesis approach before technology mapping. Table 4 shows the post-tech mapping results for the same data set for both ASIC and LUT tech mapping, and both high and low effort mixed synthesis; i.e., with and without the learning based classifier.

In addition, during the lead-up to the integration exercise, we ran LSOracle on six selections from the OpenPiton Design Benchmark (OPDB) [**4**], with sizes up to approximately 265,000 nodes, on several benchmarks from Titan23 [**5**] with sizes up to 350,000 nodes, and on the leon2 processor, with approx- imately 800,000 nodes. Figure 15 shows a suite of ten circuits taken from OPDB and OpenCores. Figure 16 shows the average ADP, PDP, and EDP of the benchmark suite above, minus the largest and smallest in the figure, techmapped with the ASAP 7nm standard cell library.

Table 4: Technology Independent Results

	Original network		ABC Optimized		Mighty		Partitioned AIG optimizations		Partitioned MIG optimizations		Mixed Synthesis	
Circuit	Nodes	Depth	AND Nodes	Depth	MAJ Nodes	Depth	Nodes	Depth	MAJ Nodes	Depth	MAJ Nodes	Depth
c3540	1,038	41	941	31	1,041	28	964	37	1,040	35	936	31
c5315	1,773	38	1,309	29	1,313	23	1,441	38	1,327	28	1,299	26
Pico-RV	17,010	36	16,483	36	15,522	26	16,909	60	16,610	38	16,566	40
b14	6,069	60	4,439	57	5,626	38	5,397	77	5,835	52	5,515	69
multiplier	27,062	274	24,556	262	34,278	107	25,791	274	31,932	229	28,674	263
sin	5,416	225	5,039	177	7,985	106	5,251	223	5,732	152	5,299	210
bar	3,336	12	3,141	12	3,097	13	3,165	12	3,188	15	3,106	15
arbiter	11,839	87	11,839	87	6,386	11	11,839	87	6,714	55	11,041	87
mem ctrl	46,836	114	45,614	110	44,155	91	46,653	114	48,661	96	47,156	111
i2c	1,342	20	1,162	15	1,254	12	1,294	20	1,273	9	1,303	12

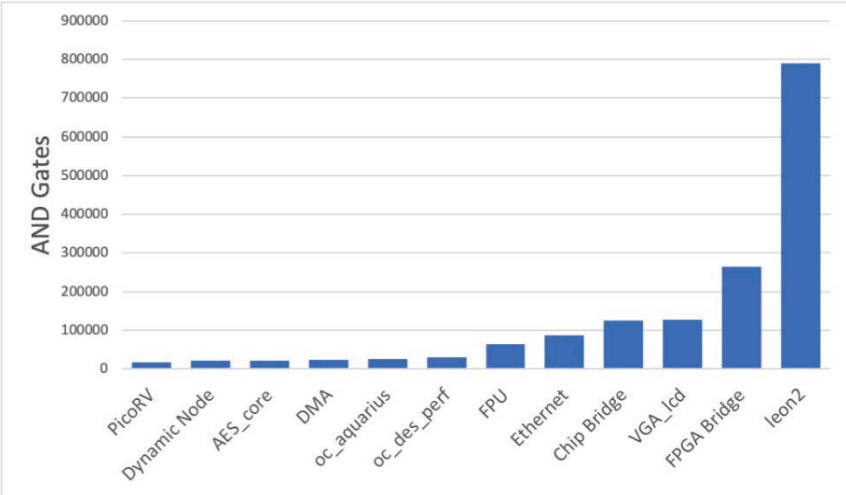

Figure 15: Size, in Number of and Gates, of 12 Circuits

4.5 TA T-2: Designware like Behavioral Synthesis

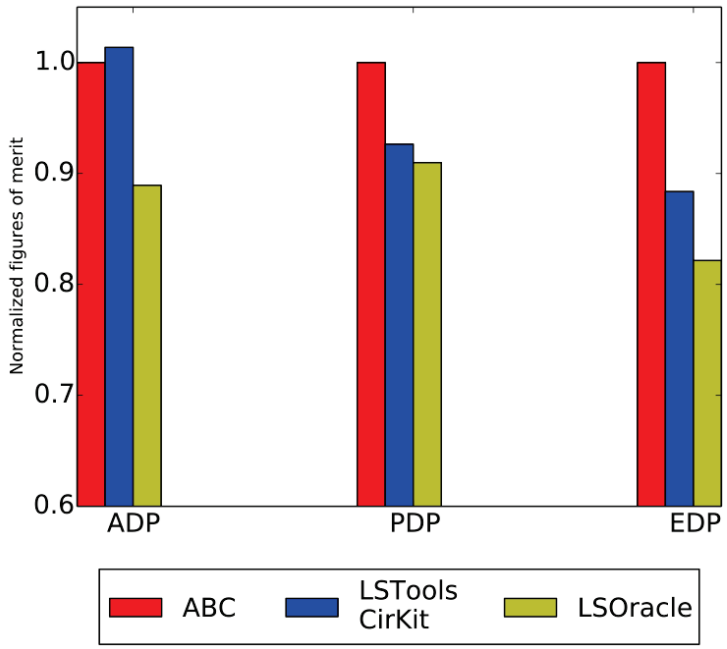

Figure 16: Average Performance Post tech-mapping on ASAP 7nm of a Suite of Ten Circuits Selected from OPDB and Opencores

We have developed a prefix tree synthesis plugin for Yosys in collaboration with Google and the University of Oklahoma. This tool allows users to use Verilog attributes to mark performance-critical adders in the HDL and specify an adder structure that should be used, such as ripple carry, sklansky, etc. It also allows a set of transforms to be specified which modify the base adder structure to improve performance. The Yosys plugin identifies adders in the design that have attributes set, then generates a custom adder of appropriate structure and size in which is reimported into Yosys to replace the generic adder used by default. Adders may be generated either in HDL which is then techmapped by Yosys, or directly into a SKY130 netlist. Support for additional technology libraries can be readily added by writing a mapping file. Figure 17 shows a set of transforms performed by the plugin which reduces an example adder from seven to three logic levels.

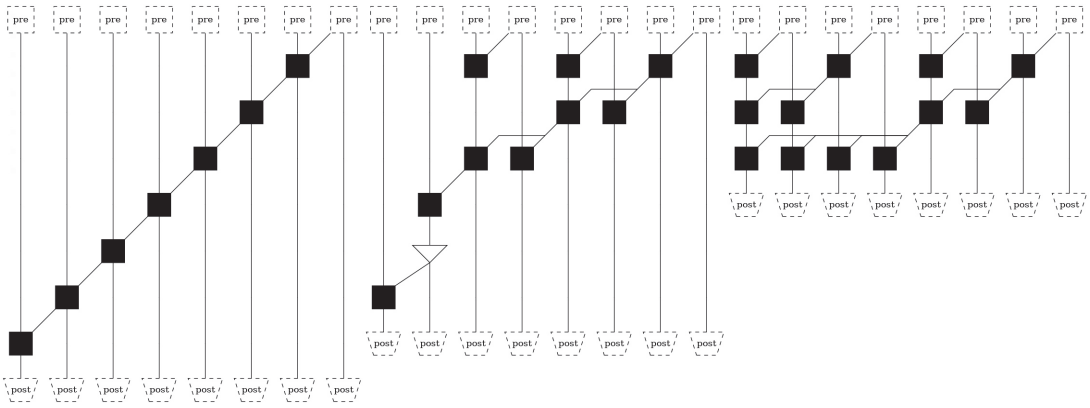

Figure 17: Example Transform of an Adder from Seven Levels to Three Levels, including One Intermediate Step to Show the Incremental Nature of the Transformation

4.5.1 TA T-3: Timing Driven Synthesis

Infrastructure Timer Evaluation
We have evaluated the available options for integrating static timing analysis into LSOracle. We found that both OpenTimer and OpenSTA produce the same critical paths on a test suite of 12 circuits that we prepared. However, we have decided to implement a lighter weight, in house timer, based on the timer from ABC, but with an expanded feature set to support SDC file input to define constraints. Our evaluation with a commercial tool found that timing analysis at the technology-independent level does not translate well to the technology-dependent level, so we evaluate the timers using the ABC standard cell mapper. Furthermore, we intend to tune the developed timer to quickly predict possible gains that different techniques for critical path restructuration might bring.

Liberty Parser and Timing Driven Infrastructure
We integrated libABC's liberty parser into LSOracle. This enables LSOracle to read liberty files into libABC's timer, as well as access standard cell attributes in LSOracle. Now that a timer and liberty parser are integrated, we can begin implementing timing driven synthesis, and can eliminate the manual work required to use the ASIC tech-mapper with new standard cell libraries.

4.5.2 T-9: Intensive Integration Testing to Achieve PPA Improvements at P&R Level

In order to see how LSOracle compares with a commercial tool at the logic synthesis level, we created two custom liberty files to force the commercial tool to output the network as an And-Inverter Graph and as a Majority-Inverter Graph. Surprisingly, given equal area, delay, and power for both and and majority standard cells, the commercial tool never uses majority gates, even when run with maximum effort. This results in comparatively inefficient representations of arithmetic circuits. Figure 18 shows the number of gates used by a commercial tool generating an MIG and by LSOracle in high effort mode over a subset of seventeen of the twenty EPFL benchmarks. The three that were excluded were because the commercial tool's runtime at

maximum effort was impractically long; in one case it ran for five days before the run was aborted. Performance between the tools is comparable at the logic synthesis level; the commercial tool outperforms LSOracle on six of the seventeen benchmarks, and on average LSOracle produces networks that are 0.5% smaller than the commercial tool.

4.6 TA T-4: Technology Mapping

While investigating methods to improve ASIC techmapping performance, we discovered that the current state-of-the-art open-source technology mapper, ABC, leaves much room for improvement in its implementation of boolean matching. By modifying the heuristics that affect the standard-cell (cover) selection during technology mapping, we found that performance could be improved substantially.

Figure 18 shows the distribution of quality of results in terms of performance (X axis), and area (Y axis) for an Advanced Encryption Standard (AES) core, using a large random selection of possible

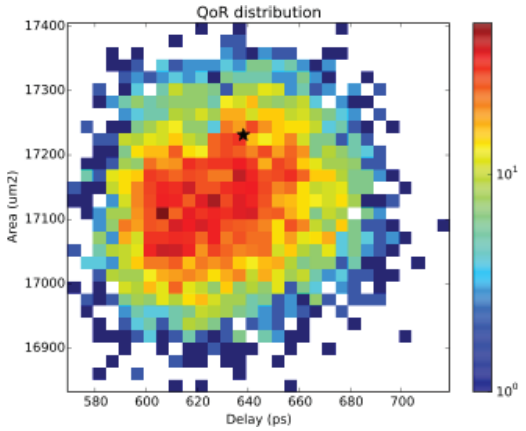

Figure 18: Distribution of Standard Cell Mappings by Area and delay for an AES Core
The best possible mapping would be in the bottom left quadrant.

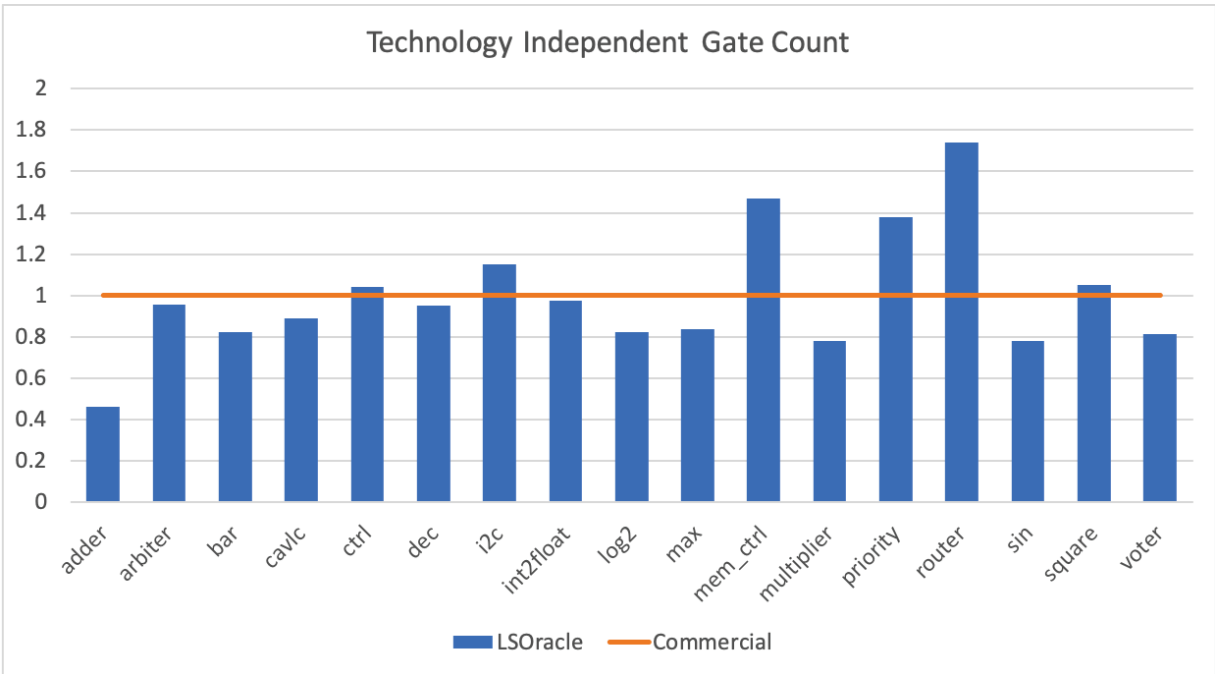

Figure 19: Technology Independent Network Size for EPFL Benchmarks, LSOracle vs Commercial Tool

Mappings within ABC. The black star shows the quality of results delivered by ABC in its unmodified configuration. These data show a gap in both metrics when comparing ABC default mapping choices with possible mappings that could be obtained by changing the mapping selection heuristic. In the case of the benchmark shown, 10% improvement in delay, and a slight improvement in area should be easily achievable with a better selection method. A similar gap was observed in many other benchmarks.

This performance loss is mainly due to a large search space forcing ABC to make assumptions about the appropriate mapping. This makes machine learning an attractive option to improve performance with- out expanding resource use beyond a feasible limit.

We have developed and implemented an intelligent flow to improve the standard-cell mapping in ABC. Our approach replaces the ABC method of selecting the covering candidates, and instead uses a Convolutional Neural Network (CNN) to improve the choice of candidate.

Figure 20 depicts the architecture used. We modified libABC and use an external ML implementian in python to apply the model; libABC is used in order to use its boolean matching engine. However, because libABC is integrated with LSOracle, these changes give immediate benefit to LSOracle's ASIC view. The blue boxes are commands created inside libABC to prepare for tech-mapping. These commands generate a low-dimensional representation of node and cuts. These feature maps go through our model, and the output is a list of cuts to be used during the match, for each node. This list is read-back into libABC with a custom command, and the usual matching algorithm takes place. The system output is a final mapped netlist.

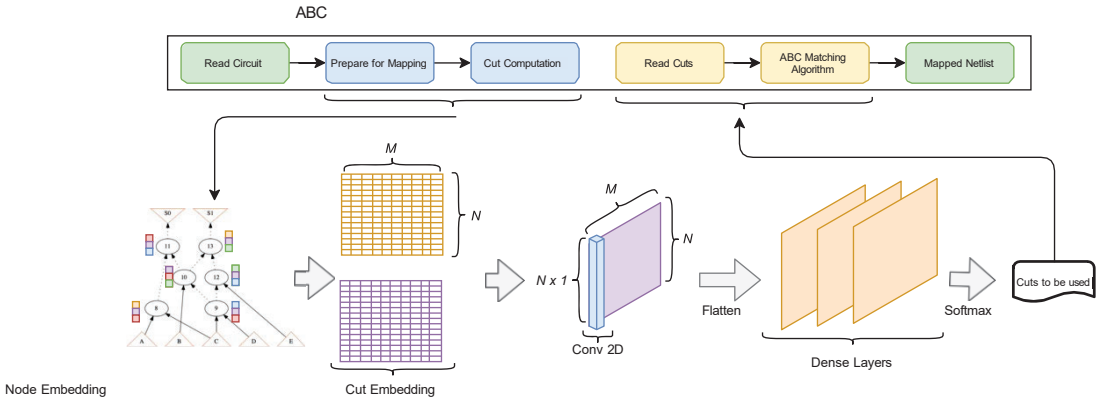

Figure 20: Architecture of ML Based ASIC Mapping Tool

An external toolchain is used for the machine learning component, and is integrated with libabc.

We went through several iterations of improvement to this model, using increasingly large training sets, and yielding better results. The current model is trained over two different adder architectures: a ripple-carry and a carry look-ahead. With this model, we achieve improvements across all 14 arithmetic blocks used for evaluation. Furthermore, we increase our memory efficiency compared to ABC. Overall, the proposed ML-based mapper reduces the number of considered cuts by 24% compared to the state- of-the-art, while improving the circuit delay, and Area-Delay-Product (ADP), by average about 10%, and 7%, respectively, with a 2% area penalty. Compared to an exhaustive approach that considers all cuts, we achieve similar or better results while saving over two times the number of considered cuts on average, yielding runtime improvements. Table 5 presents the results achieved: Results comparing our approach against standard ABC and Unlimited ABC. We highlight our method can beats the standard ABC algorithm in delay for all the test-cases. Compared to the Unlimited ABC, we improve 10 out of 14 cases. We also present results for area (µm2), and number of cuts considered.

Table 5: Results Comparing our Approach Against Standard ABC and Unlimited ABC

We highlight our method can beats the standard ABC algorithm in delay for all the test-cases

Circuit	ABC Original			ABC *Unlimited*			**SLAP**			Δ **SLAP**/ABC			Δ **SLAP**/*ABC Unlimited*		
	Area (μm^2)	Delay (*ps*)	Cuts Used	Area (μm^2)	Delay (*ps*)	Cuts Used	Area (μm^2)	Delay (*ps*)	Cuts Used	Area (μm^2)	Delay (*ps*)	Cuts Used	Area (μm^2)	Delay (*ps*)	Cuts Used
adder	898.13	3,770.65	10,118	1,009.40	3,404.62	18,228	1,031.33	**3,268.67**	1,3522	1.15	0.87	1.34	1.02	0.96	0.74
bar	2,680.39	1,114.90	42,760	2,680.39	1,114.90	43,784	3,083.23	**923.82**	10,705	1.15	0.83	0.25	1.15	0.83	0.24
c6288	3,000.45	1,265.88	95,519	3,014.68	1,228.32	197,022	3,023.54	**1,236.59**	111,609	1.01	0.98	1.17	1.00	1.01	0.57
max	2,312.27	3,809.03	39,727	2,049.60	3,980.21	41,478	2,292.44	**3,710.54**	22,521	0.99	0.97	0.57	1.12	0.93	0.54
rc256b	1,794.39	7,388.03	22,387	2,482.33	6,861.76	40,978	2,428.21	**6,807.02**	30,476	1.35	0.92	1.36	0.98	0.99	0.74
rc64b	450.70	1,844.59	5,491	601.16	1,688.38	10,066	602.33	**1,565.70**	6,397	1.34	0.85	1.16	1.00	0.93	0.64
sin	5,207.04	3,955.57	191,131	5,073.14	3,737.30	290,664	5,087.6	**3,584.79**	141,788	0.98	0.91	0.74	1.00	0.96	0.49
c7552	2,045.40	817.46	52,150	1,905.90	828.46	93,745	2,002.01	**800.09**	30,933	0.98	0.98	0.59	1.05	0.97	0.33
mul32-booth	6,802.44	1,837.68	185,406	5,623.21	1,743.56	355,106	5,597.55	**1,773.47**	147,043	0.82	0.97	0.79	1.00	1.02	0.41
mul64-booth	25,717.95	3,583.35	733,156	20,797.85	3,743.95	1,394,922	21,984.07	**3,280.47**	601,889	0.85	0.92	0.82	1.06	0.88	0.43
square	15,744.07	3,680.87	541,321	14,107.84	2,970.49	919,522	15,789.09	**3,023.01**	380,787	1.00	0.82	0.70	1.12	1.02	0.41
AES	17,321.27	638.09	264,380	16,994.21	632.07	317,827	16,489.63	**594.64**	145,532	0.95	0.93	0.55	0.97	0.94	0.46
64b mult	25,458.31	4,649.10	833,565	24,168.27	4,144.31	1,484,104	24,021.07	**4,278.48**	892,650	0.94	0.92	1.07	0.99	1.03	0.60
Pico RISCV	12,190.05	1,782.08	181,410	12,010.19	2,126.47	197,069	12,148.52	**1,601.21**	103,174	1.00	0.90	0.57	1.01	0.75	0.52
Geomean	4,637.80	2,297.91	96,321.81	4,610.87	2,222.85	150,657.67	4,760.47	**2,090.01**	73,739.02	1.03	0.90	0.77	1.03	0.94	0.49
Improvements	1.0	1.0	1.0	0.99	0.96	1.56	1.02	**0.90**	0.76	-	-	-	-	-	-

We investigated how to implement the learned model as a heuristic inside ABC. Furthermore, we started a collaboration with Alan Mishchenko from UC Berkeley, who developed the ABC tool, to create better heuristics for cut pruning targeting FPGA technology-mapping.

LUT-Based Optimization for ASICs
In our paper at the Design Automation Conference (DAC 2022) we describe LUT- based optimization in the context of ASIC designs. We show how trading off functionality and structure enhance quality of results. In particular, using LUT-based techniques allow for more powerful Boolean optimization, whereas traditional AIG optimization provides more cut-points for logic restructuring. We present how to tweak cost functions to make LUT-based optimizations tailored to ASICs, and improve nine results over the best known area results in the EPFL logic synthesis competition. This work was done in collaboration with researchers from Synopsys Inc and the University of California at Berkeley.

4.7 T-11: Optimize Partitioning Algorithm for Sequential Circuits

The partitioning tools used, both KaHyPar and BiPart, did not require modification to work on se- quential circuits. The underlying data structures, however, including the EPFL libraries, our libABC implementation, and all parsers and writers, did require modification to support sequential elements. In addition, new optimization algorithms were developed for MIGs to enhance performance when working with sequential circuits. Whenever possible, modifications were upstreamed to the parent repositories.

Sequential support for MIG-based logic optimization
Majority-Inverter Graphs (MIGs) were proposed and show a reduction in area, power and delay for arithmetic logic while comparing with AND-inverter graphs [**6**]. Current MIG methods support only combinational benchmarks, while most practical circuits are se- quential. This causes two strong limitations in advancing on the IDEA project:

1. the circuit partitions could be sequential, leading to difficulties in applying MIG optimizations.
2. the training set should include both combinational and sequential examples. If sequential MIG optimization is not available, the training set will be biased for combinational examples, leading to inaccurate classification.

Besides the inherent requirement of having sequential MIG networks, given its application in practical designs, it is also expected that new opportunities in sequential synthesis and optimization will show up with this new data structure. Furthermore, this could also benefit our POSH project, OpenFPGA, as MIG optimization has demonstrated potential in mapping carry chains which are ubiquitous in modern FPGA architectures [7]. Therefore, there is a strong interest to develop sequential MIG-based logic synthesis.

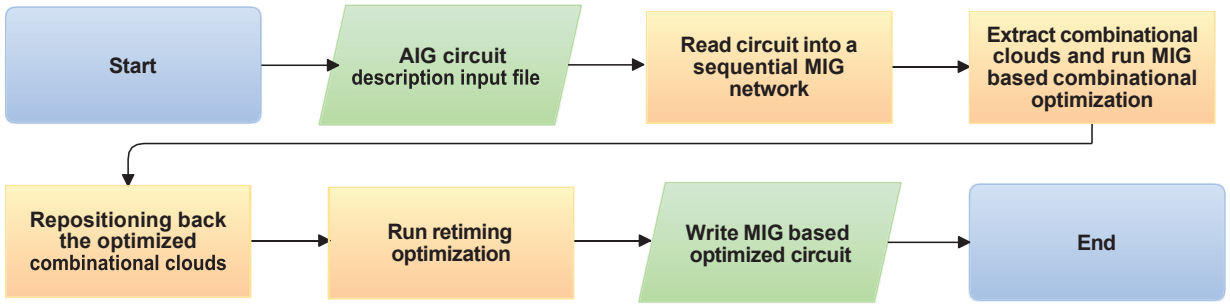

Figure 21: Flow of Sequential MIG-based Optimization

We have developed and intensively evaluated the performance of the sequential MIG optimizer against AIG and commercial counterparts. Figure 22 illustrates the two designs flows considered in comparison, inputs of which include 16 opencores benchmark circuits (more representative benchmarks to be added). The sequential MIG optimization evaluation was extended to take into account post-technology mapping results. It is also worth pointing out that our approach performs better than Synopsys DC in compile ultra mode.

We first compared our sequential MIG optimization against its sequential AIG counterpart after technology independent optimization steps. Both networks are then used as inputs to a commercial tool to proceed with technology mapping (technology mapping is not available natively on MIGs and will be implemented later targeting the predictive standard cell library ASAP 7nm. The same flow is employed in both optimized networks. Furthermore, we also compare post-technology mapping results taking into account the MIG optimized network against the original circuit feeding directly the commercial tool. Raw results are presented in Table 6, while the figure-of-merit results considering Area- Delay Product (ADP), Power-Delay Product (PDP) and Energy-Delay Product (EDP) are presented in Table 7 and Table 8 respectively. When comparing the optimized MIG network with an AIG optimized network, our results show an improvement of 12% in EDP and 13% in ADP. While comparing the MIG optimized network with the original circuit description plugged into the commercial flow, EDP is im- proved by 4%, while ADP is improved by 7% on average. The results have been presented to the Design Automation Conference (DAC 2019).

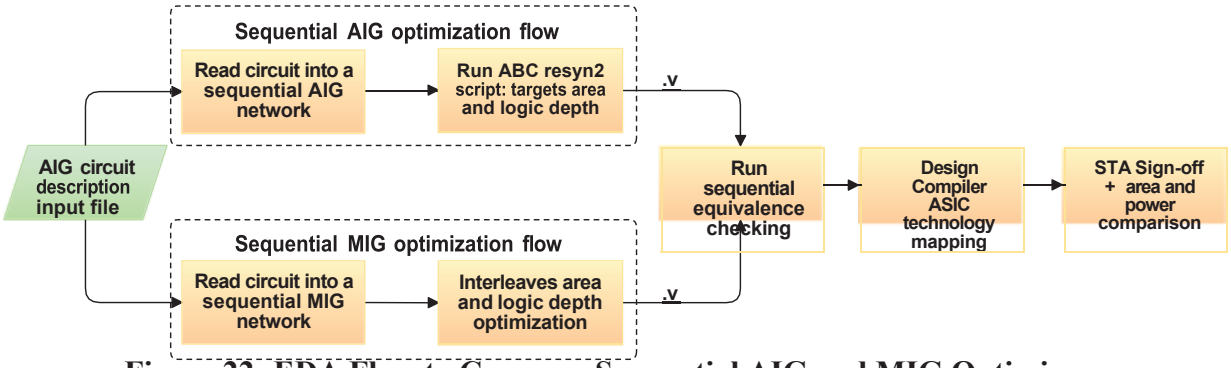

Figure 22: EDA Flow to Compare Sequential AIG and MIG Optimizers

Table 6: MIG Versus AIG After Technology Mapping

	MIG Optimized				AIG Optimized			
Circuit	Arrival (ns)	Area (um^2)	Power (mw)	Period (ns)	Arrival (ns)	Area (um^2)	Power (mw)	Period (ns)
oc_aquarius	638.82	22,609.49	2.29	650	991.21	19,899.25	1.43	1,000
oc_cfft_1024	259.27	12,550.69	7.31	270	362.89	12,190.97	5.29	370
oc_cordic_p2r	209.42	11,276.52	4.73	220	232.01	11,092.46	4.26	240
oc_cordic_r2p	202.66	15,324.86	6.58	220	262.55	16,124.54	5.44	280
oc_des_perf	169.99	27,203.48	23.20	180	179.88	27,672.13	23.19	190
oc_ethernet	153.42	3,524.39	2.34	165	140.39	3,553.08	2.47	150
oc_mem_ctrl	97.75	5,234.33	5.66	100	89.70	4,797.86	5.59	100
oc_video_dct	509.56	49,551.00	10.27	520	509.84	54,934.87	10.75	520
oc_video_jpeg	479.39	55,385.57	9.69	490	460.12	69,099.69	10.72	470
Sum:	2,720.28	202,660.33	72.07	2,815.00	3,228.59	210,364.85	69.14	3,320.00
Avg. Gain:	15.74%	3.66%	-2.61%	5.82%	-15.74%	-3.66%	2.61%	-5.82%

Sequential Support in LibABC

We added complete sequential support to our libABC integration, allowing synthesis on large blocks, without having to break a network into combinational clouds.

Sequential Support in EPFL Libraries and Yosys Integration

One of the bugs that we found during the integration exercise was that not all BLIF files written by Yosys were accepted by our tool's parser. This was because Yosys does not guarantee that the BLIF file be written in topological order, and the parser required that circuits with sequential elements be read in topological order. This bug has been fixed and a pull request has been opened with the EPFL library repositories. This fix also allowed us to improve our integration of LSOracle within Yosys. Previously, we had to use ABC as an intermediate step between Yosys and LSOracle, however since ABC uses only AIGs as its underlying data structure, MIG optimization was lost. We have updated our Yosys integration to address this and to use the latest versions of both Yosys and LSOracle. We extended the functionality of this integration to more closely mimic Yosys added ABC9 integration, and to add LSOracle techmapping to Yosys.

Table 7: MIG versus the Commercial Tool Results in the Original File

	MIG Optimized				Original		Circuit	
Circuit	Arrival (ns)	Area (um^2)	Power (mw)	Period (ns)	Arrival (ns)	Area (um^2)	Power (mw)	Period (ns)
oc_aquarius	638.82	22,609.49	2.29	650	688.65	21,222.64	2.09	700
oc_cfft_1024	259.27	12,550.69	7.31	270	280.33	12,467.41	6.62	290
_ oc cordic _p2r	209.42	11,276.52	4.73	220	209.75	11,438.41	4.69	230
oc cordic r2p	202.66	15,324.86	6.58	220	215.49	15,086.91	6.41	230
oc des perf	169.99	27,203.48	23.2	180	179.18	26,571.52	22.04	190
oc ethernet	153.42	3,524.39	2.34	165	136.95	3,536.52	2.57	250
oc mem ctrl	97.75	5,234.33	5.66	100	99.39	5,236.20	5.20	109
oc video dct	509.56	49,551.00	10.27	520	519.92	55,763.95	10.62	530
- oc-video - jpeg	479.39	55,385.57	9.69	490	469.85	59,775.89	9.99	460
Sum: _ _	2,720.28	202,660.33	72.07	2,815.00	2,799.51	211,099.45	70.23	2,989.00
Avg. Gain:	2.83%	3.99%	-2.61%	5.82%	-2.83%	-3.99%	2.61%	-5.82%

Table 8: Normalized EDP, ADP and PDP for MIG, AIG and the Commercial Tool Running Over the Original File

	Optimized			mized		O iginal	
Circuit	ADP	PDP	EDP	DP			EDP
c aquarius	1	1	1	,50		0,97	1,06
c cfft 1024	1	1	1	,42		0,94	1,06
cordic p2r	1	1	1	,11		0,95	0,99
cordic r2p	1	1	1	,39		0,95	1,10
c des perf	1	1	1	1 ,12		0,95	1,06
c ethernet	1	1	1	,88		0,91	0,88
c mem ctrl	1	1	1	,83		0,91	0,95
c video dct	1	1	1	,05		1,03	1,08
video jpeg	1	1	1	,02		0,99	0,99
Avg. Ratio:	1.000	1.000	1.000	9 0	136	3 0.978	1.042

4.7.1 TA T-5: Open-access Database Support (if API released)

OpenDB, the database used by OpenROAD referred to in this task, was never updated to store pre- technology mapping HDL, of the sort most useful to LSOracle. As such, LSOracle has not been explicitly integrated with OpenDB. However, the OpenSTA integration, detailed in TA T-4 and T-15 does link to OpenDB, allowing LSOracle to from OpenDB through OpenSTA, and for the OpenDB database to be updated by OpenSTA when called by LSOracle.

5 MANAGEMENT SUMMARY

5.1 Personnel

Over the course of the project, four researchers, one post doc, two PhD Students, one master student, and two interns worked on LSOracle at different times, for periods ranging from one or two semesters to several years. Walter Lau Neto's PhD thesis, which he defended in June 2022, was almost exclusively related to work performed under the project on the use of artificial intelligence in logic synthesis.

5.2 Collaboration with EPFL

We have worked closely with EPFL throughout the duration of the LSOracle project. LSOracle is built in part on the EPFL logic synthesis libraries, and there has been a constant exchange of ideas, bug fixes, and new features between the two teams. We have incorporated the EPFL libraries as submodules, in order to benefit from upstream improvements, and we have contributed several substantial new features, including support for sequential circuits, support for named nets, and improved parsers for a variety of formats.

5.3 Collaboration with BiPart Team at UT Austin

We have integrated the BiPart partitioner, developed by another IDEA team at UT Austin, into LSOra- cle to provide multithreaded partitioning. We also evaluated partitioning performance for several versions and contributed patches to accommodate our format requirements.

5.4 Collaboration with OpenROAD

We have integrated LSOracle into OpenROAD as a submodule and added an environment variable to enable or disable LSOracle. In addition, we are continuing to collaborate with the OpenROAD team on using LSOracle as part of their resynthesis flow. Using an OpenROAD pass, failing logic cones are extracted by OpenDB and passed to LSOracle for resynthesis using a variety of recipes and techmapping approaches. If timing is improved, the new results are integrated into the design.

5.5 ASSURE Collaboration

We collaborated with the ASSURE team at NYU on the use of LSOracle for synthesis of obfuscated logic. Using a small example, we see a substantial reduction in logic depth using LSOracle compared to ABC. In particular, the obfuscated HDL was reduced from 25 levels to 16 using LSOracle, while ABC was unable to reduce the depth. This is comparable to the original HDL's 15 logic levels. Following this work, the collaboration focused on using OpenFPGA to generate small embedded FPGAs for obfuscation; that work led to several publications.

5.6 PNNL Collaboration and PandA Framework

We have collaborated closely with the RTML team at Pacific Northwest National Lab,

integrating LSOracle in their backend flow. This work has led to several talks and publications using PandA-Bambu for HLS, then LSOracle as a Yosys plugin, then OpenROAD or OpenFPGA for placement and routing.

5.7 Verific

As part of the DARPA toolbox program, we acquired a license for Verific's verilog pareer. We de- veloped a Yosys integration with the DARPA toolbox version of Verific which is available through our Yosys plugin directory. Any user with a Verific license through the DARPA toolbox can use Verific with LSOracle without further configuration.

5.8 Sandia Collaboration

The Advanced CMOS Products/Design group at Sandia National Lab employed LSOracle as part of a toolchain for Majority based logic-optimization to replace a custom tool. We provided support getting started and integrating with other tools.

5.9 Google Collaboration on Resynthesis

We have implemented, in collaboration with Google and the University of Oklahoma, a tool for post- P&R optimization of adders and other prefix tree structures within a Yosys plugin. The tool shows sub- stantial improvements over the default adders implemented in Yosys/ABC in delay, and allows adders on non-critical paths to be area-optimized.

5.10 Commercial Adoption

RapidSilicon is exploring the use of LSOracle in their commercial FPGA synthesis flow. As part of that effort, they are developing custom LSOracle recipes which perform well in concert with their propri- etary optimizations. To assist that effort, we provided an expanded user's guide intended for developers and advanced users which includes both in depth usage for various commands and rules of thumb for tunable parameters and when to apply various strategies for best results.

6 CHALLENGES AND ISSUES

Although we were able to meet all project goals as described in the statement of work, we encountered two main challenges throughout the program.

FPGA Performance: When using more advanced FPGA synthesis methods, such as the *synth xilinx* command in Yosys/Symbiflow, much of the benefit of MIG optimization is lost due to efficient mapping of arithmetic logic onto IPs in the FPGA fabric. Although MIG synthesis reduces the number of LUTs required to implement an adder, for example, often the adder is mapped to a carry chain, leaving only AIG heavy portions of the logic to be mapped to LUTs. Based on that observation, it is clear that FPGA performance comes from using the resources on an FPGA fabric efficiently, not simply high quality logic synthesis. To address that problem, we developed an in-house, DAG agnostic, FPGA technology mapper that has support for black boxes, white boxes, and multi-output gates.

Timing Driven Synthesis and Physical Resynthesis: Implementing high quality timing driven synthesis was a major challenge during this project. We implemented timing driven passes in LSOracle, using OpenSTA for timing analysis and a library of recipes with varying degrees of area vs timing focus. However, even with aggressive recipes there may still be failing paths, particularly after placement and routing. For that reason we developed, in collaboration with OpenROAD, a pass to extract failing timing arcs after placement and routing and pass them back to LSOracle for additional optimization. We have also developed automatic recipe generation in order to avoid the limitations of a fixed library of standard recipes. This collaboration with OpenROAD is ongoing.

7 PUBLICATIONS

The following publications were produced and accepted during the period of the project:

- E. Testa, L. Amaru, M. Soeken, A. Mishchenko, P. Vuillod, J. Luo, C. Casares, P.-E. Gaillardon, G. De Micheli, ” Scalable Boolean Methods In A Modern Synthesis Flow,” Proceedings of the Design, Automation and Test in Europe (DATE), 25-29 March 2019, Florence, Italy.

- W. Lau Neto, X. Tang, M. Austin, L. Amaru and P. -E. Gaillardon, ”Improving Logic Optimiza- tion in Sequential Circuits using Majority-inverter Graphs,” 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2019, pp. 224-229, doi: 10.1109/ISVLSI.2019.00049.

- W. L. Neto, M. Trevisan Moreira, L. Amaru, C. Yu and P. -E. Gaillardon, ”Read your Circuit: Leveraging Word Embedding to Guide Logic Optimization,” 2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC), 2021, pp. 530-535.

- S. Temple, W. Lau Neto, M. Austin, X. Tang, P.-E. Gaillardon, ”LSOracle: Using Mixed Logic Synthesis in an Open Source ASIC Design Flow”, WOSET 2020

- Scott Temple, Walter Lau Neto, Max Austin, Xifan Tang, Pierre-Emmanuel Gaillardon, ”LSOra- cle: Open-source Mixed Logic Synthesis”, Invited Paper, Government Microcircuit Applications Critical Technology Conference (GOMACTech), March 29-April 1, 2021, Virtual

- W. L. Neto, M. T. Moreira, Y. Li, L. Amarù, C. Yu and P. -E. Gaillardon, ”SLAP: A Supervised Learning Approach for Priority Cuts Technology Mapping,” 2021 58th ACM/IEEE Design Au- tomation Conference (DAC), 2021, pp. 859-864, doi: 10.1109/DAC18074.2021.9586230.

- A. Snelgrove, S. Temple, P.E. Gaillardon, ”Structure Aware Partitioning for Mixed Logic Synthe- sis”, International Workshop on Logic Synthesis (IWLS) 2021

- J. Bhandari, A. K. Thalakkattu Moosa, B. Tan, C. Pilato, G. Gore, X. Tang, S. Temple, P.-E. Gaillardon, R. Karri ”Exploring eFPGA-based Redaction for IP Protection”, 2021 IEEE/ACM In- ternational Conference On Computer Aided Design (ICCAD), 2021, pp. 1-9, doi: 10.1109/IC- CAD51958.2021.9643548.

- G. Ammes, W. Lau Neto, P. Butzen, P.-E. Gaillardon, R. Ribas, ”Approximate Digital Circuit De- sign by Combining Two- and Multi-Level ALS Methods”, IEEE International Symposium on Cir- cuits and Systems (ISCAS) 2022

- R. Gauchi, A. Snelgrove, P.-E. Gaillardon, ”An Open-source Three-Independent-Gate FET Stan- dard Cell Library for Mixed Logic Synthesis”, IEEE International Symposium on Circuits and Systems (ISCAS) 2022

• W. Lau Neto, L. Amaru, V. Possani, P. Vuillod, J. Luo, A. Mishchenko and P.-E. Gaillardon, “Improvements to LUT-Based Optimization for ASIC”, DAC2022

8 CONCLUSION

During this four-year, the IDEA project developed an open-source, extensible framework for logic synthesis using a variety of optimization strategies, including use of multiple strategies on a single design by partitioning the target circuit and applying the best-fit optimizer to each partition. By partitioning SoC designs in terms of logic attributes and applying ad-hoc logic optimization technique, the tool offers an opportunity for logic synthesis to achieve significant runtime reduction and performance improvement simultaneously, and the functional integration with other IDEA groups was done. LSOracle is also integrated as a synthesis front-end on OpenRoad, allowing to extend the user base. All the open-source code was made available as initially planned and can be found on GitHub (https://github.com/lnis-uofu/LSOracle) where to this date the project received 74 stars and 33 forks, demonstrating a wide and successful response and community engagement.

9 REFERENCES

[1] Karypis Lab, University of Minnesota (2007) hMETIS - Hypergraph & Circuit Partitioning. http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download

[2] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," in IEEE Computational Science and Engineering, vol. 5, no. 1, pp. 46-55, doi: 10.1109/99.660313, 1998

[3] Frugally-deep, Header-only library for using Keras models in C++, https://github.com/Dobiasd/frugally-deep, 2019

[4] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, D. Wentzlaff," OpenPiton: An Open Source Harware Platform for your research", Communications of the ACM, 2019

[5] K. Murray, S. Whitty, S. Liu, J. Luu, V Betz, 'Titan: Enabling large and complex benchmarks in academic CAD" 1-8. 10.1109/FPL.2013.6645503, 2013

[6] L. Amarú, P. -E. Gaillardon and G. De Micheli, "Majority-Inverter Graph: A novel data-structure and algorithms for efficient logic optimization," in 51st ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 2014, pp. 1-6, doi: 10.1145/2593069.2593158, 2014

[7] Z. Chu, X. Tang, Mathias Soeken, Ana Petkovska, Grace Zgheib, Luca Amarù, Yinshui Xia, Paolo Ienne, Giovanni De Micheli, and P.-E. Gaillardon, "Improving Circuit Mapping Performance Through MIG-based Synthesis for Carry Chains", in Proceedings of the on Great Lakes Symposium on VLSI 2017 (GLSVLSI '17). doi.org/10.1145/3060403.3060432, 2017.

LIST OF ABBREVIATIONS, ACRONYMS, AND SYMBOLS

ACRONYM	DESCRIPTION
ADP	Area- Delay Product
AIG	And-Inverter Graph
BFS	Breadth First Search
CNN	Convolutional Neural Network
DAC	Design Automation Conference
EDA	Electronic Design Automation
EDP	Energy-Delay Product
EPFL	Swiss Federal Institute of Technology Lausanne
FPGA	Field Programmable Gate Array
KMI	Karnaugh Map Image
MIG	Majority-Inverter Graphs
PDP	Power-Delay Product
PPA	Power, Performance Area
RTL	Register-Transfer-Level
TNS	Total Negative Slack
WNS	Worst Negative Slack