E-Syn: E-Graph Rewriting with Technology-Aware Cost Functions for Logic Synthesis

Chen Chen*
HKUST(GZ)
cchen099@connect.hkust-gz.edu.cn

Guangyu Hu*
HKUST
ghuae@connect.ust.hk

Dongsheng Zuo
HKUST(GZ)
dzuo721@connect.hkust-gz.edu.cn

Cunxi Yu
University of Maryland, College Park
cunxiyu@umd.edu

Yuzhe Ma
HKUST(GZ)
yuzhema@hkust-gz.edu.cn

Hongce Zhang
HKUST(GZ)
hongcezh@hkust-gz.edu.cn

ABSTRACT

Logic synthesis plays a crucial role in the digital design flow. It has a decisive influence on the final Quality of Results (QoR) of the circuit implementations. However, existing multi-level logic optimization algorithms often employ greedy approaches with a series of local optimization steps. Each step breaks the circuit into small pieces (e.g.,k-feasible cuts) and applies incremental changes to individual pieces separately. These local optimization steps could limit the exploration space and may miss opportunities for significant improvements. To address the limitation, this paper proposes using e-graph in logic synthesis. The new workflow, named E-Syn, makes use of the well-established e-graph infrastructure to efficiently perform logic rewriting. It explores a diverse set of equivalent Boolean representations while allowing technology-aware cost functions to better support delay-oriented and area-oriented logic synthesis. Experiments over a wide range of benchmark designs show our proposed logic optimization approach reaches a wider design space compared to the commonly used AIG-based logic synthesis flow. It achieves on average 15.29% delay saving in delay-oriented synthesis and 6.42% area saving for area-oriented synthesis.

CCS CONCEPTS

• **Hardware → Combinational synthesis**; **Circuit optimization**.

KEYWORDS

E-graph, technology-aware, logic synthesis

ACM Reference Format:
Chen Chen, Guangyu Hu, Dongsheng Zuo, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. 2024. E-Syn: E-Graph Rewriting with Technology-Aware Cost Functions for Logic Synthesis. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27,2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3656246

1 INTRODUCTION

Logic synthesis is a common starting point of the digital design automation flow and significantly affects the Quality of Results (QoR) such as area, timing, and power consumption. The modern logic synthesis flow typically contains a technology-independent optimization phase followed by technology mapping.

Technology-independent optimization applies various transformations to minimize design costs represented by technology-independent metrics such as graph node count and logic level. This step focuses on optimizing the logic structure of the circuit without considering specific gate-level implementations.

One of the commonly used modern logic synthesis frameworks is ABC [3], which is based on the And-Inverter Graph (AIG) representation of logic circuits. ABC is equipped with various algorithms (for example, `rewrite`, `refactor`, and `resubstitute` [13, 15, 16]) to reduce the node count or logic level in an AIG. These algorithms follow a common design concept: they attempt incremental changes to a fraction of the graph (e.g., a k-feasible cut) and each chooses the logic form with the most cost saving. This idea has achieved great success in multi-level logic synthesis. However, there are two major challenges that limit the optimality of the synthesized circuit. First, the local decisions of logic rewriting are greedy in nature as they do not account for the influence of other optimization steps. Consequently, a sequence of local optimizations may lead to a local minima and result in a tremendous loss of optimization opportunities. Second, the technology-independent cost metrics, such as AIG node count, may not always reflect post-mapping QoR. Figure 1 gives such an example. It compares the gate-level netlists after applying different optimizations. The original logic form contains 20 AIG nodes. While `rewrite` reduces the node count to 17, the area after technology mapping actually goes up.

Motivated by the limitations of existing works, we introduce E-Syn, a novel logic optimization method that utilizes equivalence graphs (e-graphs) in Boolean logic rewriting. E-graph is a data structure that preserves equivalence during rewriting-based optimizations. Thanks to the efficient and concise representation, it is possible to keep a large set of equivalent forms of the same logic. We can defer candidate selection until the completion of rewriting and therefore, can take a more global view when choosing from equivalent candidates. This unique feature of e-graph makes it easier to explore more logic forms in the search for an optimal gate-level implementation. Meanwhile, E-Syn allows the use of customizable cost functions such as a machine learning model that directs optimization towards a specific technology-dependent target. The last two netlists in Figure 1 show the results of delay-oriented and area-oriented E-Syn optimizations. In this example, E-Syn does not blindly reduce the corresponding AIG nodes which may not always lead to actual area saving. Instead, it targets post-mapping QoR and obtains logic forms with a much lower delay and a comparable area consumption.

Specifically, this paper makes the following contributions:

- It presents a novel logic optimization framework E-Syn that leverages e-graph for combinational logic rewriting. E-Syn maintains equivalent classes of the logic specification and can

*Both authors contributed equally to this research.

This work is supported in part by Guangzhou Municipal Science and Technology Project (Municipal Key Laboratory Construction Project, Grant No.2023A03J0013) and by National Natural Science Foundation of China (Grant No. 62304194).

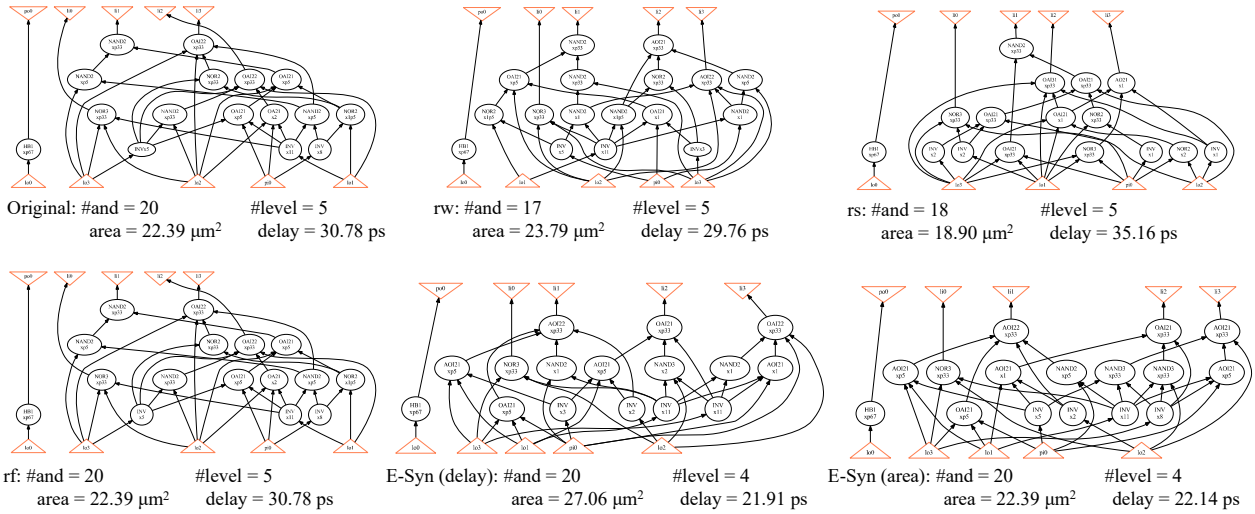

Figure 1: Netlists after technology mapping and gate sizing, using the original logic form (`original`) or after applying one of AIG rewriting (`rw`), resubstitution (`rs`), refactoring (`rf`), delay-oriented e-graph rewriting, or area-oriented e-graph rewriting.

explore a wider range of equivalent logic forms to search for an optimal design.

- This is the first work that extends the application of e-graph to Boolean logic optimization at the gate level, while previous studies concentrated on applying e-graph optimizations at higher levels (such as RTL or high-level synthesis).
- We propose a technology-aware logic optimization method using cost models obtained from XGBoost regression. The integration of the machine learning bridges the gap between technology-independent logic cost and post-mapping QoR.
- For e-graph processing, we bring up a *pool extraction* method to accommodate customizable cost functions in e-graph extraction, which scales better than the time-consuming integer linear programming method in the prior work and lifts the linear and monotone restrictions on the cost function. It also outperforms the default extraction heuristics by 21% and 10% in delay and area, respectively.
- We conduct a comprehensive evaluation based on post-mapping QoR on benchmark circuits from EPFL [1], LGSynth [22, 23], ITC [6], ISCAS [4] etc. Experiments show `E-Syn` improves 15.29% in delay for delay-oriented optimization compared to the state-of-the-art AIG-based logic optimizations in `ABC`, and achieves 6.42% area saving in area-oriented synthesis.

The rest of the paper is structured as follows: in the next section, we discuss the related work of logic synthesis and the applications of e-graph in optimization. Section 3 presents the workflow of `E-Syn`, followed by experiment results in Section 4. Finally, the paper concludes with Section 5.

2 RELATED WORK

2.1 Multi-level logic synthesis

Modern multi-level logic optimization techniques work on common technology-independent logic representations, such as And-Inverter-Graphs (AIGs) [15], Majority-Inverter-Graphs (MIGs) [2], and XOR-based representations [10, 11]. Optimizations are centered around reducing certain metrics defined upon these representations, such as the graph node count or the longest path. For example, in the commonly-used logic synthesis tool `ABC` [3], the `rewrite` operation traverses the graph to find opportunities to replace a k-feasible cut with a logic-equivalent form given that the replacement brings about the most node count decrease [15]. Other AIG rewriting techniques [13–15], such as `resubstitute`, `refactor` and `balance`, follow a similar fashion that they work on the subgraphs and apply local changes each time. However, it is generally a hard question of how to precisely evaluate the different choices of local rewriting at each step. E-graph-based optimization in `E-Syn` differs from the prior works as it keeps the equivalent classes in the graph. There is a separate *extraction* step after rewriting to pick the best logic form for the whole graph. At this step, the selection decision may refer to a more general cost model that predicts the actual delay or area cost with features from the whole graph.

2.2 Optimization using e-graphs

An e-graph, or equivalence graph, is a data structure that compactly represents a large number of equivalence relations. It has been used inside automated theorem provers, for example, Z3 [9]. Recent works have demonstrated the power of e-graph in rewriting-driven optimizations. Coward *et al.* proposed a datapath optimization approach that represents designs as data-flow graphs and leverages e-graphs and equality saturation techniques for efficient design space exploration at the register-transfer-level [7]. The IMpress framework [19] employed e-graphs to tackle the implementation problem of large integer multiplication in high-level synthesis (HLS), where e-graph helps to explore the possible ways to to decompose multipliers corresponding to different hardware implementations [20]. Applying e-graph in optimization solves the phase ordering problem as the ordering of transformations in e-graph is less of a concern. Besides optimization, in the work [8], the authors proposed an equivalence checking method that utilizes the e-graph to rewrite RTL for complex datapaths. E-graph in verification transcends the traditional bit-level logic reasoning and helps to prove equivalence at a higher level.

Table 1: Rewriting Rules in E-Syn

Class	Boolean Rewriting Rules
Complements	$a * 1 \Rightarrow a$ $a * 0 \Rightarrow 0$ $a + 1 \Rightarrow 1$ $(\neg a) * a \Rightarrow 0$ $(\neg a) + a \Rightarrow 1$
Covering	$a * (a + b) \Rightarrow a$ $a + (a * b) \Rightarrow a$
Combining	$((a * b) + (a * \neg b)) \Rightarrow a$ $((a + b) * (a + \neg b)) \Rightarrow a$
Idempotency	$a * a \Rightarrow a$ $a + a \Rightarrow a$
Commutativity	$a * b \Leftrightarrow b * a$ $a + b \Leftrightarrow b + a$
Associativity	$(a * b) * c \Leftrightarrow a * (b * c)$ $(a + b) + c \Leftrightarrow a + (b + c)$
Distributivity	$a * (b + c) \Rightarrow a * b + a * c$ $(a + b) * (a + c) \Rightarrow a + (b * c)$ $(a * b) + (a * c) \Rightarrow a * (b + c)$
Consensus	$(a * b) + ((\neg a) * c) + b * c \Rightarrow (a * b) + (\neg a) * c$ $((a+b)*((\neg a)+c))*(b+c) \Rightarrow (a+b)*((\neg a)+c)$
De-Morgan	$\neg(a * b) \Rightarrow \neg a + \neg b$ $\neg(a + b) \Rightarrow (\neg a) * (\neg b)$

These aforementioned prior works were based by egg [21], a fast and flexible open-source library, which provides built-in functionality for efficient e-graph manipulation and an extendable interface to incorporate domain-specific rewriting rules.

Compared to the prior works, this paper extends e-graph to bit-level optimization with Boolean algebra rules for rewriting. To effectively process the large-scale logic formulas in digital circuits, we devise efficient format converters to integrate the egg library into the logic synthesis flow and put forward a fast and objective-aware pool extraction method to accommodate nonlinear technology-aware cost models in logic optimization.

3 E-GRAPH REWRITING FOR LOGIC SYNTHESIS

In this section, we introduce how e-graph rewriting is used to optimize Boolean logic. Figure 2 shows the overall workflow of our proposed method. E-Syn is built upon the classic Yosys/ABC logic synthesis flow with additional e-graph optimization steps to allow a wider exploration for optimal design forms.

3.1 Rewriting using E-graph

In E-Syn, we take advantage of e-graph to efficiently represent the Boolean logic formulas that are equivalent to the given function specification. In an e-graph, equivalent Boolean functions are clustered into equivalent classes (e-classes). For example, Figure 3 demonstrates an e-graph for the logic function $xy + xz$, where each dotted box is an e-class. Nodes are maximally reused in the parent functions to avoid the repetition of equivalent nodes. Thanks to the compact representation of nested e-classes, a number of e-graph nodes can potentially represent exponentially many equivalent forms. E-graph does not restrict the set of operators to use. Though it is possible to use only AND-gates and inverters to mimic AIG rewriting, we decide to loosen the requirement on the operators and allow free use of AND, OR and NOT to match the input logic function specification.

The construction of an e-graph utilizes the equality saturation technique [18], which has been well-established in the formal methods and compiler research communities. Equality saturation applies a series of rule-based transformations to generate equivalent representations of the given function. In E-Syn, we use the laws of Boolean algebra shown in Table 1 as the rewriting rules. "⇔" in the table means the rewriting rule is applied in both directions, while for a few rules that serve as purely a simplification, we only apply it from left to right, as indicated by "⇒". Unlike typical logic rewriting techniques (e.g., DAG-aware AIG rewriting) that heuristically select among equivalent representations locally at the rewriting step, E-Syn keeps those equivalent representations in the graph and takes a separate extraction step to select the best candidate after the completion of rewriting.

3.2 Extraction

The extraction step traverses the e-graph to select an optimal logic form for each node. For each e-class with more than one element, it needs to choose based on a certain cost model. The selection finally returns an abstract syntax tree (AST) from the graph. Prior works have used the depth or size of the AST as the cost function. However, these cost functions may not well reflect the actual area or delay cost of the circuit netlist after technology mapping. Therefore, to better account for the actual QoR in the extraction step, we make use of regression to obtain a more technology-aware cost model.

3.2.1 Regression. Inspired by the regression model used in RTL-stage QoR prediction [17], we can employ an XGBoost model to fit the area and delay cost from the AST of a Boolean expression. The features used in the regression are listed as follows:

- **Boolean operator count.** For each type of Boolean operator, we count their occurrence in the AST.
- **AST node count.** The number of nodes in an AST could reflect the overall scale of the logic circuit and therefore could be one feature in the regression.
- **AST depth.** AST depth serves a similar purpose as the logic level in logic synthesis, which could correlate with the length of paths and therefore, the delay of the circuit.
- **Graph density and edge sum.** AST can also be regarded as a graph and we utilize the commonly-used graph features. In our regression model, we consider the graph density and edge sum as two features. Roughly speaking, these two features indicate the abundance of edges in a graph.

To generate the training data for regression, we use `aigfuzz` in the AIG library (https://github.com/arminbiere/aiger) to create a dataset of 50000 random-sized combinational logic circuits with an average logic level of 234 and an average size of 6305 AIG nodes. These circuits are then converted into the equation format and transformed into e-graphs for feature extraction. We further run technology mapping and technology-dependent optimization on these circuits using ABC. The reported delay and area at this step are used as the training labels. We use two separate XGBoost regression models to predict area and delay, respectively. Each contains 200 estimators and has a maximum depth of 5. The delay prediction achieves an R-value of 0.78, and the R-value of area prediction is 0.76. and an MAE of 19.6ps.

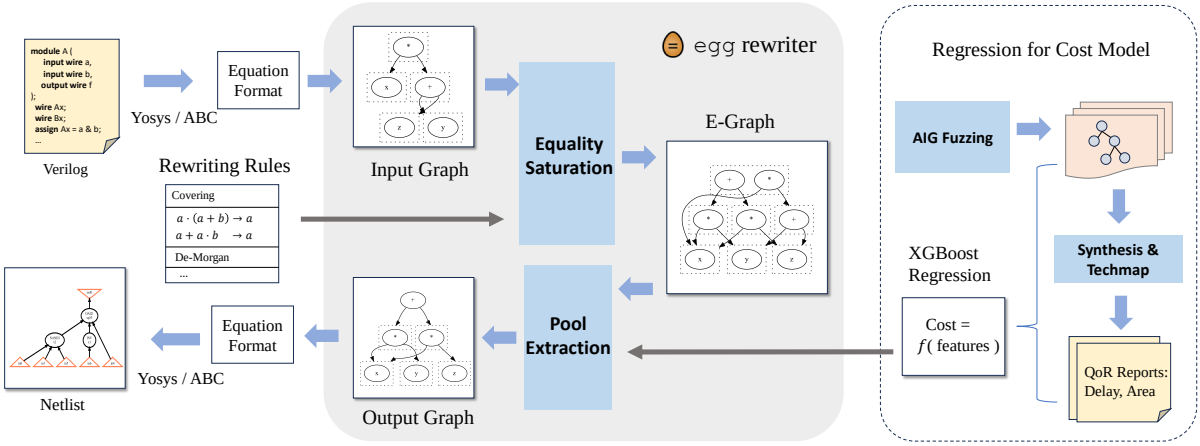

Figure 2: The framework of E-Syn

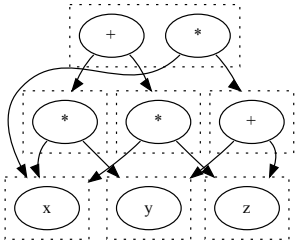

Figure 3: An example of e-graph for logic function $xy + xz$ where we use "$*$" for AND and "+" for OR.

To integrate the XGBoost model with egg library for e-graph extraction, we use the Rust binding of the XGBoost library.

3.2.2 Extraction with Technology-Aware Cost. The existing extraction methods in the prior works either (1) depend on local heuristic decisions per e-node or (2) require solving an integer linear programming (ILP) problem. For extractor (1), the local heuristics primarily rely on local cost functions such as the size or depth of a graph, which may significantly diverge from the actual cost. While for existing extractor (2), though it is global as it formulates extraction into an ILP problem to solve, it limits the form of the cost function to be linear and monotonically increasing from leaf nodes to parent nodes. In order to best fit the technology-dependent cost, the cost model might not be linear and monotone. Besides, it is generally hard to scale ILP up for large-scale Boolean logic functions in practical digital circuits. Therefore, in this application of e-graph, we are looking for a fast and flexible extraction method.

With the consideration of the pros and cons of the two existing methods, we bring up the novel *pool extraction method* in E-Syn to achieve efficient logic function extraction using technology-aware cost models. As its name suggests, pool extraction first collects a pool of candidates from an e-graph using a combination of heuristics, and then it evaluates each candidate using the given cost model to pick the best one in the pool. Pool extraction combines the above two existing extraction methods and serves as a trade-off between efficiency and performance. It also removes the limitations on the cost functions.

Specifically, the candidate pool consists of one AST with the fewest number of nodes, one with the least tree depth, and candidates from sampling in the e-graph. The sampling process traverses the e-classes in the e-graph with two strategies: (a) randomly select an e-node only from those with the same least local cost, or (b) select an e-node that has a sub-optimal local cost with a probability. The first strategy is different from the default extractor (1) as we introduce randomness, whereas the default extractor will always choose the first candidate among those with the same least cost. The second strategy occasionally explores a choice with sub-optimal local cost and therefore, can potentially find a form with better global cost. Here, the local cost is one of the AST depth, AST size, or the weighted sum of operators (we assign a lower weight to NOT than AND, OR). We empirically set the probability of sub-optimal exploration to 0.2 and the ratio of candidates sampled from these two strategies as 1:3.

After we obtain a pool of ASTs following the above strategies, we will evaluate each candidate using the technology-aware cost model obtained in Section 3.2.1. The one with the lowest cost will be selected as the optimal logic form for the given function specification.

3.3 Integration with the Existing Synthesis Flow

E-graph rewriting is not exclusive to the existing logic synthesis flow. Instead, it is designed as an enhancement to be inserted at the beginning of a logic optimization flow to improve the final QoR. With the command write_eqn from ABC, a combinational logic circuit in the AIG synthesis flow can be written into the equation format, which contains Boolean AND, OR, NOT operators, intermediate variables and nested parentheses. The equation format specification is then transformed into nested S-expressions in Common Lisp, which is the input format for egg, the library for e-graph rewriting and equality saturation. The output AST from egg is then converted back to the equation format and can be later processed by the traditional logic optimization flow. We also check the result using combinational equivalence checking to ensure correct implementation of logic rewriting in e-graph. As for the implementation, we design high-performance parsers in Rust for the above format conversion, which supports parallel execution to minimize the influence on the synthesis time.

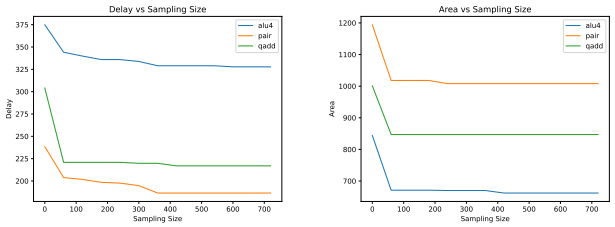

Figure 4: Sampling size vs. minimum delay and area

4 EXPERIMENT RESULT AND DISCUSSION

4.1 Experiment Setup

E-graph rewriting and extraction are implemented in `Rust` to interface with the egg library. All experiments are performed on a Ubuntu 20.04.4 LTS server equipped with Intel Xeon Platinum 8375C processors, an NVIDIA 3090 GPU, and 128GB memory. The test circuits are primarily arithmetic blocks, including those from LGSynth [22, 23], ITC [6], EPFL [1] and ISCAS85 [4] benchmark suites. We also use `genmul` [12] to generate two multipliers (3x3 and 5x5). Additionally, we take an open-source divider from OpenCores [1]. The arithmetic circuits cover a wide range of complexity levels, ranging from simple adders to complex multipliers, and therefore, provide a diverse set of benchmarks that can test different aspects of the logic optimization algorithm. The setup for equality saturation runtime limit is 300 seconds and the e-node limit is 2500000 nodes. Throughout these experiments, we use the ASAP 7nm technology library [5].

4.2 Effectiveness of the Pool Extraction Method

For fast and efficient extraction using a technology-aware cost model, we design the pool extraction method in Section 3.2. To pick a proper sampling size for our pool extraction method, we conduct experiments to measure the influence of sampling size to the QoR. We incrementally sample more candidates on the e-graphs of test circuits while measuring the technology-dependent costs using the following command: ***strash; dch -f; map; topo; upsize; dnsize; stime***. We take record of the best area and delay among all candidates in the pool under different sampling pool sizes, and plot their relation with the sampling size in Figure 4. For QoR under a small sampling size (< 100), the best in the pool may vary from run to run due to the randomness in sampling, and it becomes more deterministic for larger sampling sizes. It can be seen that there is a diminishing return from expanding the sampling pool. A pool size of over 100 would suffice in most cases.

We also compare our proposed pool extraction method to the vanilla extraction methods in the egg library. Because we use the XGBoost regression model as the cost function, which is not linear, it is not feasible to compare with the ILP method. The other built-in extractor for comparison is the default greedy extractor that takes either AST size or AST depth as the cost function. For delay comparison, we use AST depth as the cost function for the vanilla egg extractor and the delay regression model for the pool extraction method. For area comparison, we use AST size in vanilla extractor and the area regression model for pool extraction. We assess the final delay and area cost by the same `ABC` flow above. The results are plotted in Figure 5, where delay and area metrics are normalized by the QoR when no e-graph rewriting is used, as indicated by the legend ABC in the figure. Our findings suggest that e-graph rewriting with the vanilla extractor may not always improve delay or area. It is necessary to use pool extraction with technology-aware cost models. Compared to the vanilla extractor, the pool extraction method achieves up to 25% area saving (avg. 10%), and up to 34% of reduction on delay (avg. 21%). Overall, it improves from the baseline `ABC` flow by 6% and 18% in area and delay, respectively.

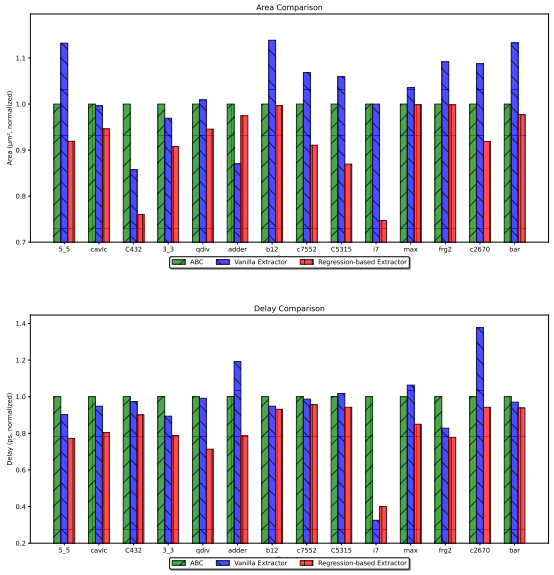

Figure 5: Comparison of e-graph optimization using the vanilla extractor vs. our pool extraction. Results are normalized by QoR from the baseline `ABC` flow.

4.3 Comparison on the Design Space

To further validate our claim that e-graph logic rewriting can reach a wider design space, we conduct experiments to compare circuits produced by `E-Syn` to those resulted from a commonly-used logic synthesis flow in `ABC`. The default ABC synthesis flow available as the ***abc*** command in Yosys is as follows: ***strash; ifraig; scorr; dc2; dretime; retime -o -D {delay}; strash; &get -n; &dch -f; &nf -D {delay}; &put; buffer; upsize -D {delay}; dnsize -D {delay}; stime***. It contains a delay constraint parameter that can be set to control the delay-area trade-off in synthesis. In addition to the various AIG rewriting

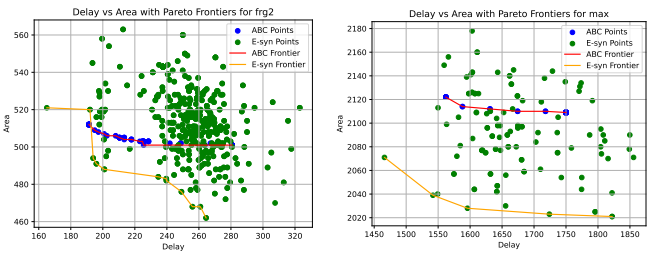

Figure 6: Comparison of design space and Pareto-frontier of `E-Syn` and AIG rewriting

[1] https://opencores.org/projects/verilog_fixed_point_math_library

Table 2: QoR of E-Syn and ABC synthesis flow under different constraints

Circuit	ABC delay-oriented		E-Syn delay-oriented		ABC area-oriented		E-Syn area-oriented		ABC balanced		E-Syn balanced	
	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)
adder (EPFL)	988	2172.78	988	**2168.92**	981	2182.02	981	**2178.23**	983	2173.38	988	**2169.75**
bar (EPFL)	2262	197.82	2266	198.21	2235	234.2	**2141**	307.91	2238	218.63	**2201**	**206.8**
max (EPFL)	2122	1562.07	**2071**	**1466.06**	2109	1750.01	**2021**	1822.13	2112	1631.15	**2105**	**1581.91**
cavlc (EPFL)	452	150.2	466	**129.12**	434	186.19	**415**	211.37	441	151.49	442	**149.67**
3_3 (genmul)	41	146.44	44	**113.28**	37	165.24	**33**	181.14	40	146.95	**37**	**117.17**
5_5 (genmul)	144	424.64	**132**	**329.64**	130	466.33	**116**	**402.48**	135	437.86	**120**	**422.85**
qdiv (opencore)	1123	747.75	1280	**465.18**	1101	812.67	**1089**	**709.38**	1103	755.68	**1102**	**648.15**
C5315 (LGSynth91)	1075	351.03	**1043**	**314.19**	1058	384.14	**1012**	401.84	1062	367.51	**1050**	**347.12**
i7 (LGSynth91)	477	96.69	**347**	**93.98**	468	162.32	**321**	180.81	473	103.39	**345**	**99.85**
c7552 (ISCAS85)	1191	465.85	1298	**299.55**	1176	587.9	**1175**	**470.56**	1185	482.12	**1182**	**459.39**
c2670 (ISCAS85)	536	240.84	537	**200.65**	494	299.45	**481**	304.84	522	256.67	**516**	**219.55**
frg2 (LGSynth89)	512	191.34	521	**165.04**	501	280.64	**470**	307.57	505	210.4	**488**	**200.95**
C432 (LGSynth89)	98	372.69	112	**335.19**	95	451.72	**91**	**368.79**	96	396.51	**94**	**363.48**
b12 (ITC99)	770	244.33	776	**219.74**	736	303.24	**734**	**299.32**	750	257.07	**747**	**249.71**
GEOMEAN	540.10	342.93	540.90	**290.51**	520.49	414.37	**487.10**	416.50	529.97	358.34	**507.39**	**334.28**
Improvements				**15.29%**			**6.42%**				**4.26%**	**6.71%**

operations performed by the ***dc2*** command, this flow also makes use of techniques like structural hashing, correlated signal reduction to further optimize the given logic. Moreover, the ***&dch*** command combines different networks seen during technology-independent synthesis into a single network with choices. This is to help the subsequent technology mapping to better choose the logic structure with the optimal costs. We argue that this synthesis flow in comparison is already a relatively powerful one. We tune the target delay in this script to generate designs with different area-delay trade-offs and plot the QoR on the delay-area plane. These make up the ABC design points for comparison. As for the E-Syn flow, we plot the QoR of all candidates in the pool for comparison.

Figure 6 shows the design points for a medium-size and a large-size test circuit. In general, the design points from E-Syn span a wider range in the delay-area plane. In both designs, the frontier of E-Syn completely dominates. We also compare QoR for the test circuits under different constraints (delay-oriented, area-oriented, and balanced). The experiment results are shown in Table 2. Though AIG-based rewriting has been extensively optimized towards area, our method can further reduce the area cost by a margin of 6.42%, if the users are willing to sacrifice more delay. On the side of delay-oriented optimization, E-Syn achieves a delay reduction on almost all designs, averaging 15.29%. For the delay-area-balanced optimization target, our method outperforms in both delay and area. Overall, the comparison over three regions in the design space indicates the Pareto-frontier of E-Syn dominates the above ABC synthesis flow.

Regarding runtime, E-Syn flow takes 80 seconds on average to run on a test circuit, in addition to the 300-seconds time-limit for equality saturation, which may be lowered to trade quality for time.

5 CONCLUSION

This paper proposes using e-graph rewriting in logic synthesis. It extends e-graph optimization to the bit level. E-graph-based optimization explores a wider range of logic forms than local logic rewriting. It can also factor in technology-aware costs to better target delay or area optimization in logic synthesis.

REFERENCES

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL combinational benchmark suite. In *IWLS*.

[2] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2016. Majority-Inverter Graph: A New Paradigm for Logic Optimization. *TCAD* 35, 5 (2016), 806–819.

[3] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*. Springer, 24–40.

[4] Franc Brglez. 1985. A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran. In *Proc. Intl. Symp. Circuits and Systems, 1985.*

[5] Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016), 105–115.

[6] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. 2000. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design & Test* 17, 3 (2000), 44–53.

[7] Samuel Coward, George A Constantinides, and Theo Drane. 2022. Automatic Datapath Optimization using E-graphs. In *ARITH*. IEEE, 43–50.

[8] Samuel Coward, Emiliano Morini, Bryan Tan, Theo Drane, and George Constantinides. 2023. Datapath Verification via Word-Level E-Graph Rewriting. *arXivL preprint arXiv:2308.00431* (2023).

[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[10] Winston Haaswijk, Mathias Soeken, Luca Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2017. A novel basis for logic rewriting. In *ASP-DAC*. 151–156.

[11] Ivo Hálecek, Petr Fiser, and Jan Schmidt. 2017. On XAIG Rewriting. In *IWLS*. 89–96.

[12] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. 2021. GenMul: Generating architecturally complex multipliers to challenge formal verification tools. In *Recent Findings in Boolean Techniques: Selected Papers from the 14th International Workshop on Boolean Problems*. Springer, 177–191.

[13] Alan Mishchenko and Robert Brayton. 2006. Scalable Logic Synthesis using a Simple Circuit Structure. In *IWLS*, Vol. 6. 15–22.

[14] Alan Mishchenko, Robert Brayton, and Stephen Jang. 2010. Global Delay Optimization using Structural Choices. In *FPGA*. 181–184.

[15] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. DAG-Aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis. In *DAC*. ACM, 532–535.

[16] Heinz Riener, Siang-Yun Lee, Alan Mishchenko, and Giovanni De Micheli. 2022. Boolean Rewriting Strikes Back: Reconvergence-Driven Windowing Meets Resynthesis. In *ASP-DAC*. 395–402.

[17] Prianka Sengupta, Aakash Tyagi, Yiran Chen, and Jiang Hu. 2022. How Good Is Your Verilog RTL Code? A Quick Answer from Machine Learning. In *ICCAD*. 1–9.

[18] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *POPL*. ACM, New York, NY, USA, 264–276.

[19] Ecenur Ustun, Ismail San, Jiaqi Yin, Cunxi Yu, and Zhiru Zhang. 2022. Impress: Large Integer Multiplication Expression Rewriting for FPGA HLS. In *FCCM*. IEEE, 1–10.

[20] Ecenur Ustun, Cunxi Yu, and Zhiru Zhang. 2023. Equality Saturation for Datapath Synthesis: A Pathway to Pareto Optimality. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–2.

[21] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. In *POPL*, Vol. 5. ACM, New York, NY, USA, 1–29.

[22] S Yang. 1989. Logic synthesis and optimization benchmarks. In *IWLS*. 14.

[23] Saeyang Yang. 1991. *Logic synthesis and optimization benchmarks user guide: version 3.0.* Citeseer.