DAG-aware Synthesis Orchestration

Yingjie Li, Mingju Liu, Haoxing Ren, Alan Mishchenko, Cunxi Yu

***Abstract*—Modern logic synthesis techniques use multi-level technology-independent representations like And-Inverter-Graphs (AIGs) for digital logic. This involves structural rewriting, resubstitution, and refactoring based on directed-acyclic-graph (DAGs) traversal. Existing DAG-aware logic synthesis algorithms are designed to perform one specific optimization during a single DAG traversal. However, we empirically identify and demonstrate that these algorithms are limited in quality-of-results due to the solely considered optimization operation in the design concept. This work proposes Synthesis *Orchestration*, which is a fine-grained node-level optimization implying multiple optimizations during the single traversal of the graph. Our experimental results are comprehensively conducted on all 104 designs collected from ISCAS'85/89/99, VTR, and EPFL benchmark suites. The orchestration algorithms consistently outperform existing optimizations, *rewriting*, *resubstitution*, *refactoring*, leading to an average of 4% more node reduction with reasonable runtime cost for the single optimization. Moreover, we evaluate the orchestration algorithm in the sequential optimization, and as a plug-in algorithm in *resyn* and *resyn3* flows in ABC, which demonstrate consistent logic minimization improvements (1%, 4.7% and 11.5% more node reduction on average). Finally, we integrate the orchestration into OpenROAD for end-to-end performance evaluations. Our results demonstrate the advantages of the orchestration optimization techniques, even after technology mapping and post-routing in the design flow.**

I. Introduction

Logic optimization plays a critical role in design automation flows for digital systems, significantly impacting area, timing closure, and power optimizations [1], [2], [3], [4], [5], [6], [7], as well as influencing new trends in neural network optimizations [8], [9]. The goal of logic optimization is to achieve higher performance, reduced area, and lower power consumption, all while maintaining the original functionality of the circuit.

Y. Li, M. Liu and C. Yu are with the Department of Electrical and Computer Engineering, University of Maryland, College Park, US (e-mails: yingjieli@umd.edu, mliu9867@umd.edu cunxiyu@umd.edu). A. Mishchenko is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, US (e-mail: alanmi@berkeley.edu). M. Ren is with Nvidia Research, Austin, Texas, US (e-mail: haoxingr@nvidia.com).

This work is funded by National Science Foundation (NSF) NSF-2008144 and NSF CAREER award NSF-2047176.

Digital Object Identifier 10.1109/TCAD.XXXXXXX

Modern digital designs are complex and feature with millions of logic gates, coupled with an extensive exploration space. This complexity underscores the importance of efficient, technology-independent optimizations for design area and delay at the logic level. Key methodologies in modern logic optimization techniques are conducted on multi-level, technology-independent representations, such as And-Inverter-Graphs (AIGs) [10], [11], [12] and Majority-Inverter-Graphs (MIGs) [4], [13], for digital logic. Additionally, XOR-rich representations are crucial for emerging technologies, as seen in XOR-And-Graphs [14] and XOR-Majority-Graphs [15].

A framework for logic synthesis, ABC [5], introduces multiple state-of-the-art (SOTA) Directed-Acyclic-Graphs (DAGs) aware Boolean optimization algorithms. These include structural rewriting (command *rewrite* in ABC) [10], [16], [17], resubstitution (command *resub* in ABC) [18], and refactoring (command *refactor* in ABC) [10], all of which are based on the AIG data structure. During the existing logic optimization process, the algorithm considers a single specific optimization method and applies the optimization based on a single criterion [10]. Our empirical studies further reveal critical limitations inherent in the mainstream stand-alone concept of logic optimization, particularly in missing significant optimization opportunities. These opportunities are often overlooked due to a consistent tendency of becoming stuck in "bad" local minima. In other words, the optimization of a node, when various applicable optimization opportunities are present, is constrained by the limitations inherent in the current stand-alone optimization concept. For instance, as depicted in Figure 1, although node g is suitable for both *refactoring* and *resubstitution*, it misses potential optimization opportunities when subjected solely to *rewriting*.

In this work, we propose a novel logic synthesis development concept, **DAG-aware Synthesis *Orchestration***, that maximizes optimizations through Boolean transformations by orchestrating multiple optimization operations in the single traversal of the logic graph. Specifically, we implement the synthesis orchestration approach based on AIGs by orchestrating *rewrite*, *refactor*, and *resub* implemented in ABC [5] in the single optimization command *orchestration*. The orchestration algorithm is orthogonal to other DAG-aware synthesis

algorithms, which can be applied to Boolean networks independently and/or iteratively. Our results demonstrate that applying orchestration in DAG-aware synthesis can significantly improve logic optimization compared to the existing optimization methods. We anticipate that the concept of logic synthesis orchestration can be effectively extended to other data structures, such as Majority-Inverter Graphs (MIGs) [4].

The main contributions of the work are summarized as follows:

- Our comprehensive analysis and examples (Figures 1 and 2) highlight significant optimization losses in current logic optimization implementations.
- We propose two DAG-aware synthesis orchestration algorithms, *Priority-ordered orchestration* and *Local-greedy orchestration* to define the criteria for orchestrating *rewrite*, *refactor*, and *resub* in AIG optimizations (Section III).
- We provide the performance evaluations and runtime analysis on 104 designs from five benchmark suites (ISCAS'85/89 [19], ITC/ISCAS'99 [20], VTR [21], and EPFL benchmarks [22]), which shows our orchestration technique achieves an average of 4.2% more AIG reductions compared to existing logic optimization algorithms in ABC (Section IV-A and IV-B).
- We provide the evaluations of sequential optimizations with orchestration algorithms, where the orchestration techniques show its performance advantage of 4.7% for `resyn` and 11.5% for `resyn3` (Section IV-C).
- We further integrate orchestrated logic optimizations into OpenROAD [23] for end-to-end design evaluations, demonstrating consistent AIG minimization and area improvements for post-technology mapping and routing (Section IV-D).
- Our approach is available in ABC [5] through a new command, *orchestration*.

II. Preliminary

A. Boolean Networks and AIGs

A Boolean network is a directed acyclic graph (DAG) denoted as $G = (V, E)$ with nodes V representing logic gates (Boolean functions) and edges E representing the wire connection between gates. The input of a node is called its *fanin*, and the output of the node is called its *fanout*. The node $v \in V$ without incoming edges, i.e., no *fanins*, is the *primary input* (PI) to the graph, and the nodes without outgoing edges, i.e., no *fanouts*, are *primary outputs* (POs) to the graph. The nodes with incoming edges implement Boolean functions. The level of a node v is defined by the number of nodes on the longest structural path from any PI to the node inclusively, and the level of a node v is noted as $level(v)$.

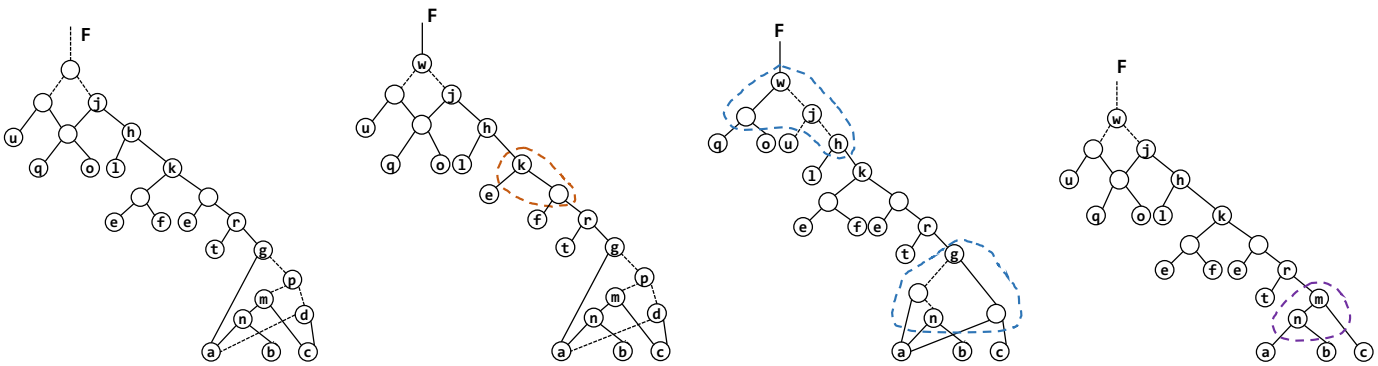
And-Inverter Graph (AIG) is one of the typical types of DAGs used for logic manipulations, where the nodes in AIGs are all two-inputs AND gates, and the edges represent whether the inverters are implemented. An arbitrary Boolean network can be transformed into an AIG by factoring the SOPs of the nodes, and the AND gates and OR gates in SOPs are converted to two-inputs AND gates and inverters with DeMorgan's rule. There are two primary metrics for evaluation of an AIG, i.e., *size*, which is the number of nodes (AND gates) in the graph, and *depth*, which is the number of nodes on the longest path from PI to PO (the largest level) in the graph. A *cut* C of node v includes a set of nodes of the network. The leaf nodes included in the *cut* of node v are called *leaves*, such that each path from a PI to node v passes through at least one leaf. The node v is called the *root* node of the *cut* C. The cut size is the number of its leaves. A cut is K-feasible if the number of leaves in the cut does not exceed K. The logic optimization of Boolean networks can be conducted with the AIGs efficiently [24], [7] based on the Boolean algebra enabled logic transformations.

B. DAG-Aware Logic Synthesis

To minimize logic complexity and size, subsequently leading to enhanced performance, DAG-aware logic optimization approaches leverage Boolean algebra at direct-acyclic-graph (DAG) logic representations, aiming to minimize area, power, delay, etc., while preserving the original functionality of the circuit.

This is achieved through the application of various technology-independent optimization techniques and algorithms, such as node rewriting, structural hashing, and refactoring. In this work, we focus specifically on exploring DAG-aware logic synthesis using And-Inverter Graphs (AIGs) representations. The AIG-based optimization process, during a single traversal of the logic graph, typically involves two steps: **(1)** *transformability check* – checking the transformability of the optimization operation for the logic cut in relation to the current node; **(2)** *graph updates* – if the optimization is applicable, the optimization operation is applied at the node to realize the transformation of the logic cut and subsequently update the graph.

Rewriting [10], denoted as `rw`, is a fast greedy algorithm for logic optimization. It iteratively selects an AIG logic cut with the current node as the root node and replaces the selected subgraph with the same functional pre-computed subgraph (NPN-equivalent) of a smaller size



(a) original AIG (b) AIG with `rw` (c) AIG with `rf` (d) AIG with `rs`

Fig. 1: The optimized graph produced by stand-alone optimization operations: (a) original AIG, graph size is 25; (b) optimized AIG with stand-alone `rw`, graph size is 23; (c) optimized AIG with stand-alone `rf`, graph size is 23; (d) optimized AIG with stand-alone `rs`, graph size is 22.

to realize the graph size optimization. In the default settings in ABC, the target logic cuts for each node are 4-feasible cuts. For AIG rewriting, all 4-feasible cuts of the nodes are pre-computed using the fast cut enumeration procedure. In each iteration, the Boolean function for the current logic cut is computed and its NPN-class is determined by hash-table lookup. After trying all available subgraphs, the one that leads to the largest improvement at a node is used. For instance, Figure 1b illustrates the optimization of the original graph shown in Figure 1a using `rw`. The algorithm traverses each node in topological order, checking the transformability of its cut with *rewriting*. In Figure 1b, node $k = efr$ is optimized using `rw`, resulting in a reduction of 2 nodes for the logic optimization.

Refactoring [10], denoted as `rf`, is a variation of the AIG *rewriting* using a heuristic algorithm to produce a larger cut for each AIG node. Refactoring optimizes AIGs by replacing the current AIG structure with a factored form of the cut function. For example, Figure 1c illustrates the optimization of the original AIG with `rf`. Node g is optimized to the factored form of $g = ac(\overline{n}+a)$ and the node w is optimized to $w = qo(u+h)$. As a result, the optimized graph with `rf` has a graph size of 23 with 2 nodes reduction.

Resubstitution [18], denoted as `rs`, optimizes the AIG by replacing the function of a node with functions of other existing nodes, referred as divisors, within the graph. This approach aims to eliminate redundant nodes unnecessary for expressing the function of the current node. In *resubstitution*, cuts containing no more than 12-16 leaves are considered, and the optimization is performed using explicitly computed truth tables and exhaustive simulation. During *resubstitution*, the introduction of new nodes may occur to complete the functionality in the AIG, which is a process known as k-resubstitution (where k represents the number of newly introduced nodes) and k should not exceed the number of nodes saved by the optimization. In the default settings of ABC, k-resubstitution is checked for $k = \{0, 1, 2, 3\}$, and the number of divisors in each cut is limited to 150. For example, in Figure 1a, the node $g = a\overline{p}$, with $p = \overline{m}\overline{d}$, $d = \overline{a}c$, and $m = abc$, implies $g = abc$. This condition allows for resubstitution with node m, leading to the removal of nodes $g, p,$ and d from the graph, as depicted in Figure 1d. Consequently, `rs` optimizes the original AIG by reducing the graph size through the removal of 3 nodes.

Definition 1: *Stand-alone* Logic Optimization: Stand-alone logic optimization refers to the process of optimizing the logic graph using a single pre-set optimization criterion during the single traversal of the entire graph. **Example 1:** The existing optimizations, such as structural *rewriting*, *refactoring*, and *resubstitution*, are stand-alone optimizations as they only assess the transformability with respect to a single pre-set operation and update the graph based on the corresponding optimization criterion.

III. Approach

In existing logic optimization algorithms that follow a stand-alone optimization approach as shown in Figure 1, certain nodes may miss optimization opportunities. For instance, node g, which is suitable for both *refactoring* and *resubstitution*, may be overlooked for optimizations in *rewriting*. To further enhance the logic optimization process in DAG-aware logic synthesis, we introduce "*Orchestration*" for logic optimization in this work. This approach is in contrast to *Stand-alone Logic Optimiza-*

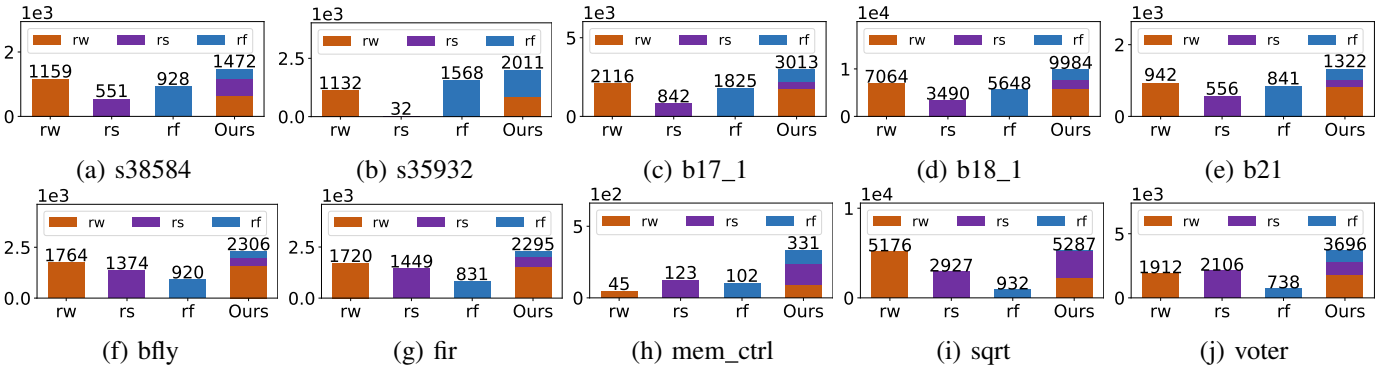


Fig. 2: The optimization opportunities with different optimization operations. The $X-$axis denotes the optimization operations. The $Y-$axis denotes the number of valid iterations with the corresponding operation.

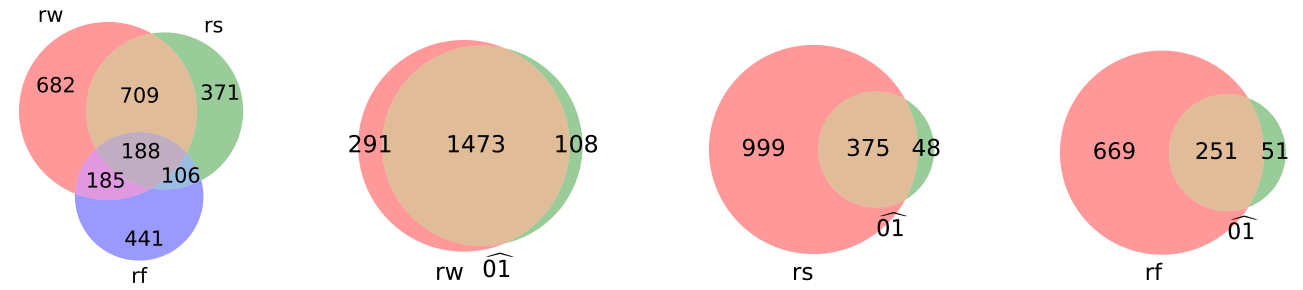


Fig. 3: The Venn diagram for the design *bfly*—a detailed analysis of Figure 2f—illustrates the relationships as follows: (a) among standalone optimizations; and (b) to (d) between orchestration optimization $\widehat{\texttt{O1}}$ (as defined in Section III-C) and each of `rw`, `rs`, `rf`, respectively. In each diagram, we consider only the nodes that participate in valid iterations within the respective optimizations. Furthermore, for $\widehat{\texttt{O1}}$ in diagrams (b) to (d), the number of valid iterations is aligned with that of the corresponding optimization for comparison.

tion defined in Definition 1. We provide the details of *Orchestrated Logic Optimization* in Definition 2.

Definition 2: *Orchestrated* Logic Optimization: Orchestrated logic optimization involves multiple optimization operations being considered during a single traversal of the logic graph. In each optimization iteration, multiple operations can be evaluated and applied based on the predefined orchestration criteria.

In the orchestration optimization, multiple optimizations are made available for each node, thereby maximizing its optimization opportunities. Specifically, we orchestrate optimization operations including *rewriting* (`rw`), *resubstitution* (`rs`), and *refactoring* (`rf`), in a single traversal of the AIG for the logic optimization. The orchestration technique can be iteratively applied to the AIG multiple times, in combination with other optimization operations such as *balance*, *redundancy removal* to achieve iterative DAG optimization. Moreover, the optimized AIG resulting from our orchestration method can be verified for equivalence to the original AIG using Combinatorial Equivalence Checking (CEC).

In this section, we first explore optimization opportunities in the single traversal of AIG for both standalone optimizations and orchestrated optimization. We then introduce two orchestration policies: *Local-greedy orchestration*, which selects the operation yielding the highest local gain (i.e., the number of nodes saved by applying the optimization operation) at each node for AIG optimization, and *Priority-ordered orchestration*, which prioritizes operations in a predefined order for AIG optimization at each node.

A. Optimization Opportunities Studies

First, we analyze the optimization opportunities in a single traversal of the AIG for various optimization methods. We record the number of iterations where optimization leads to graph updates, termed as "valid iterations," in this analysis. The results for logic optimizations using `rw`, `rs`, `rf`, and the orchestration method are illustrated in Figure 2. The orange bar represents the number of valid iterations with `rw`, purple for `rs`, and blue for `rf`. The bar labeled "Ours" shows the number

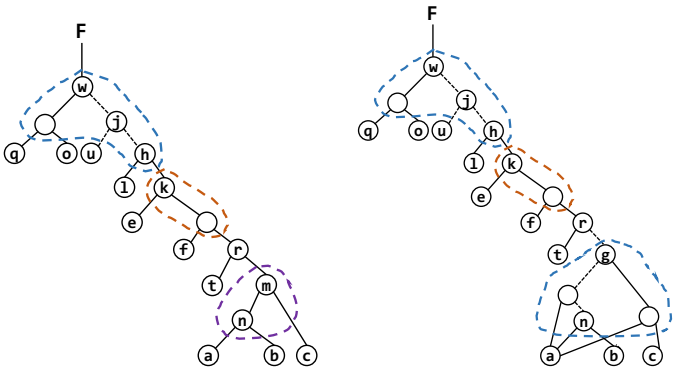
of valid iterations with the orchestration optimization, incorporating valid iterations from different optimizations (`rw`, `rs`, `rf`), indicated by the corresponding colors within the bar. For instance, for the design *voter*, stand-alone optimization methods (`rw`/`rs`/`rf`) result in 1917/2106/738 valid iterations respectively (Figure 2j). In contrast, the orchestration method yields 3696 valid iterations, presenting 93%, 75%, and 400% more valid iterations than the `rw`, `rs`, and `rf` methods, respectively, in a single traversal of the AIG.

For a better illustration, we present a Venn diagram using design *bfly* in Figure 3 as a detailed analysis of Figure 2f. Here, the orchestration algorithm employed is the *priority-ordered* algorithm with $\widehat{\texttt{01}}$, which prioritizes `rw` most, then `rs` and `rf` least, and follows the definition in Section III-C. The diagram in Figure 3a demonstrates that while there are overlaps between different stand-alone optimizations, most root nodes found are distinct for each method. Additionally, the diagrams in Figure 3b to 3d for orchestration optimization and its corresponding stand-alone optimizations highlight unique root nodes in both approaches. It is noteworthy that the ratio of overlap to uniqueness varies with different orchestration algorithms and across designs.

Our observations from this study highlight two key points: **(1)** Stand-alone optimization algorithms can miss significant optimization opportunities; **(2)** Orchestrating multiple optimizations in a single DAG traversal can introduce more optimization opportunities and more efficient logic optimization.

Given the context of orchestration, we can define the theoretical solution space and its optimal solution as follows: Consider a combinational And-Inverter Graph (AIG), denoted as $G(V, E)$. It is postulated that within the entire solution space, which encompasses $3^{|V|}$ possibilities, there exists at least one orchestration decision ensuring that $G(V, E)$ can be minimized to its smallest possible form utilizing a single traversal algorithm. Consequently, the theoretical upper limit for the complexity associated with pinpointing the optimal orchestration solution scales exponentially with the size of the graph, represented by $|V|$. Nevertheless, within the scope of Boolean minimization, it has been empirically observed that the expansive solution space of $3^{|V|}$ may actually equate to a considerably reduced space of quality-of-results, specifically concerning the dimensions of the optimized AIGs. Note the solution space will increase if orchestration elaborates more than three synthesis techniques (i.e., increasing the base of the exponential complexity). This space of results tends to be notably constricted for smaller Boolean networks.

Thus, to orchestrate multiple optimizations in a single AIG traversal, an effective orchestration policy (heuristic) is essential. In this work, we propose two policies: **(1) The *Local-greedy orchestration***, which selects the optimization operation resulting in the highest local gain (node reductions from the logic transformation of the operation) at the node for AIG optimization; and **(2) The *Priority-ordered orchestration***, which follows a pre-defined priority order for orchestrating multiple operations, i.e., applying optimizations according to the order. These policies are detailed in Algorithms 1 and 2, respectively.



(a) AIG with `Local-greedy` orchestration (b) AIG with `Priority-ordered` orchestration

Fig. 4: The optimized graph produced by orchestration optimization operations: (a) optimized AIG with *Local-greedy orchestration*, graph size is 19; (b) optimized AIG with *Priority-ordered orchestration*, graph size is 21.

B. Algorithm 1: Local-greedy Orchestration

The *Local-greedy orchestration* algorithm takes a graph $G(V, E)$ as input, where V represents the set of nodes in the AIG, and E denotes the edges between nodes. Following the topological order, we initially check the transformability of each node with respect to all orchestrated optimization operations, namely `rw`, `rs`, and `rf`. This process yields the corresponding local optimization gains, G_{rw}, G_{rs}, and G_{rf} (line 2). When none of the operations are applicable to a node, the local gain G is set to -1. Once the local gains G_{rw}, G_{rs}, and G_{rf} at the node are determined, the algorithm identifies the optimization operation with the highest non-negative local gain (lines 3, 6, and 9). The operation with the highest gain is then applied, and the graph is updated accordingly (lines 4, 7, and 10). If no operation is applicable (all gains are negative), the node is bypassed for optimization (line 12), and the algorithm proceeds to the next node in the iteration (line 13).

Compared to *stand-alone optimizations*, the *Local-greedy* orchestration algorithm incurs additional runtime overhead due to the necessity of pre-computing transformability checks and local gains for all available optimization operations (line 2).

Algorithm 1: Local-greedy Orchestration

```
   Input : G(V,E) ← Boolean Networks/Circuits in AIG
   Output: Post-optimized AIG G(V,E)
1  for v ∈ V in topological order do
2      check transformability of v w.r.t orchestrated operations:
         rw, rs, rf, and get the corresponding optimization
         gain: G^v_rw, G^v_rs, G^v_rf. // if operation is not
         applicable, G is −1; otherwise, G is
         a non-negative number.
3      if G^v_rw ≥ 0 and G^v_rw ≥ G^v_rs and G^v_rw ≥ G^v_rf then
4          Apply rw to v and update G(V,E)
5          continue   // rw with the highest gain
6      else if G^v_rs ≥ 0 and G^v_rs ≥ G^v_rw and G^v_rs ≥ G^v_rf then
7          Apply rs to v and update G(V,E)
8          continue   // rs with the highest gain
9      else if G^v_rf ≥ 0 and G^v_rf ≥ G^v_rw and G^v_rf ≥ G^v_rs then
10         Apply rf to v and update G(V,E)
11         continue   // rf with the highest gain
12     else
13         continue
```

Example: An illustrative example is shown in Figure 4a, based on the original AIG from Figure 1a. Following the topological order of the AIG, the Primary Inputs (PIs) are bypassed for optimization. Nodes n, m, d, and p are also skipped as none of the optimizations are applicable to them. The algorithm then evaluates node g, checking its transformability with `rw`, `rs`, and `rf`, and determining the local gains as $G_{rw} = -1$, $G_{rs} = 3$, and $G_{rf} = 1$. The *Local-greedy orchestration* algorithm selects the operation with the highest local gain for optimization, in this case, `rs`. By iteratively traversing the entire logic graph, the *Local-greedy orchestration* algorithm optimizes the AIG with a node reduction of 6, as depicted in Figure 4a.

C. Algorithm 2: Priority-ordered Orchestration

In this algorithm, the selection of the optimization operation to be applied at each node depends on a pre-defined priority order with respect to the available optimizations. For the three operations `rw`, `rs`, and `rf`, there are six possible permutations of the priority order, namely: $\widehat{\texttt{O1}} \mapsto$ `rw>rs>rf`, $\widehat{\texttt{O2}} \mapsto$ `rw>rf>rs`, $\widehat{\texttt{O3}} \mapsto$ `rs>rw>rf`, $\widehat{\texttt{O4}} \mapsto$ `rs>rf>rw`, $\widehat{\texttt{O5}} \mapsto$ `rf>rs>rw`, and $\widehat{\texttt{O6}} \mapsto$ `rf>rw>rs`. For instance, the priority order $\widehat{\texttt{O1}}$ implies that `rw` has the highest priority during optimization, meaning it is checked first for transformability; `rs` is evaluated next if `rw` is not applicable to the node; and `rf` is considered last when the higher priority operations are not applicable.

Algorithm 2 outlines the implementation of the priority-ordered orchestration for logic graphs. This algorithm takes an AIG $G(V, E)$ and a priority orchestration policy $P_>$ as inputs. The policy $P_>$ is defined as a precedence-ordered set of operations, wherein the operation positioned first has the highest priority. As delineated in Algorithm 2, for each node, the algorithm initially examines the transformability of the highest-priority operation, $P_>[0]$ (line 2). If $P_>[0]$ is applicable, it is applied, and the graph is updated accordingly (line 3). Following this, the algorithm proceeds to the next node without evaluating other lower-priority operations (line 4). If $P_>[0]$ is not applicable, the algorithm assesses the next highest-priority operation, $P_>[1]$ (lines 5 – 7). This process is repeated, methodically evaluating operations in descending order of priority (lines 8 – 10). In cases where none of the operations in the policy $P_>$ are applicable, the node is bypassed in the current iteration, resulting in no modifications to the graph (lines 11 – 12).

The selection of the most effective priority order depends heavily on the specific design domain. The initial transformation chosen can significantly impact the optimization process. Operations with higher priority tend to play a more critical role. Furthermore, incorporating domain knowledge into the optimization process can improve performance. Machine learning techniques can be helpful in this regard and exploring their potential leads to more further work.

Algorithm 2: Priority-ordered orchestration

```
   Input : G(V,E) ← Boolean Networks/Circuits in AIG
   Input : Orchestration rule: P_>(rw, rf, rs)
   // P_> is a list as the permutation of the
      available Boolean transformations.
   Output: Post-optimized AIG G(V,E)
1  for v ∈ V in topological order do
2      if v is transformable w.r.t P_>[0] then
3          Apply P_>[0] to v and update G(V,E)
4          continue        // check first priority
5      else if v is transformable w.r.t P_>[1] then
6          Apply P_>[1] to v and update G(V,E)
7          continue       // check second priority
8      else if v is transformable w.r.t P_>[2] then
9          Apply P_>[2] to v and update G(V,E)
10         continue        // check third priority
11     else
12         continue
```

Example: An illustrative example is shown in Figure 4b, using the original AIG from Figure 1a. The priority sorting is set as $P_>(\texttt{rw}, \texttt{rf}, \texttt{rs})$, corresponding to $\widehat{O2}$.

TABLE I: Detailed results of selected large size designs. Comparison of single-traversal *orchestration* with stand-alone optimizations from ABC.

Design		AIG										
		Baseline	`rw`	`rs`	`rf`	$\widehat{\texttt{O1}}$	$\widehat{\texttt{O2}}$	$\widehat{\texttt{O3}}$	$\widehat{\texttt{O4}}$	$\widehat{\texttt{O5}}$	$\widehat{\texttt{O6}}$	`LocalGreedy`
		#Node	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)
ISCAS	s38584	12400	10697 (13.7%)	11505 (7.2%)	10932 (11.8%)	10366 (16.4%)	10379 (16.3%)	**10336 (16.7%)**	10655 (14.1%)	10932 (11.8%)	10932 (11.8%)	10449 (15.7%)
	s35932	11948	9110 (23.8%)	11916 (0.3%)	9836 (17.7%)	**8561 (28.4%)**	**8561 (28.4%)**	**8561 (28.4%)**	9836 (17.7%)	9836 (17.7%)	9836 (17.7%)	**8561 (28.4%)**
	b17_1	27647	24178 (12.6%)	26305 (4.9%)	24533 (11.3%)	22951 (17.0%)	23125 (16.4%)	**22935 (17.0%)**	23393 (15.4%)	24532 (11.3%)	24532 (11.3%)	23008 (16.7%)
	b18_1	79054	66807 (15.5%)	73076 (7.6%)	69606 (12.0%)	63431 (19.8%)	64135 (18.9%)	**63167 (20.1%)**	65956 (16.6%)	69586 (12.0%)	69587 (12.0%)	63726 (19.4%)
	b20	12219	10659 (12.8%)	11197 (8.4%)	10593 (13.3%)	10017 (18.02%)	10110 (17.3%)	**10011 (18.1%)**	10228 (16.3%)	10590 (13.3%)	10590 (13.3%)	10129 (17.1%)
	b21	12782	10863 (15.1%)	11449 (10.4%)	10961 (14.3%)	10146 (20.6%)	10237 (19.9%)	**10133 (20.7%)**	10458 (18.2%)	10958 (14.3%)	10958 (14.3%)	10261 (19.7%)
	b22	18488	15983 (13.6%)	16891 (8.6%)	15983 (13.6%)	14977 (19.0%)	15115 (18.2%)	**14953 (19.1%)**	15275 (17.4%)	15965 (13.6%)	15965 (13.6%)	15127 (18.2%)
VTR	bfly	28910	26827 (7.2%)	27060 (6.4%)	27487 (4.9%)	**25996 (10.1%)**	26183 (9.4%)	26027 (10.0%)	26353 (8.8%)	27487 (4.9%)	27487 (4.9%)	26181 (9.4%)
	dscg	28252	26132 (7.5%)	26352 (6.73%)	26972 (4.5%)	**25339 (10.3%)**	25552 (9.5%)	25345 (10.3%)	25768 (8.8%)	26970 (4.5%)	26970 (4.5%)	25496 (9.7%)
	fir	27704	25641 (7.5%)	25768 (7.0%)	26437 (4.6%)	**24778 (10.6%)**	25061 (9.5%)	24831 (10.4%)	25189 (9.1%)	26437 (4.6%)	26437 (4.6%)	24987 (9.8%)
	syn2	30003	27787 (7.4%)	28031 (6.6%)	28617 (4.6%)	**27013 (10.0%)**	27266 (9.1%)	27048 (9.9%)	27444 (8.5%)	28617 (4.6%)	28617 (4.6%)	27198 (9.3%)
EPFL	div	57247	41153 (28.1%)	52621 (8.1%)	56745 (0.9%)	**41123 (28.2%)**	41143 (28.1%)	41124 (28.2%)	52098 (9.0%)	56738 (0.9%)	56738 (0.9%)	41147 (28.1%)
	hyp	214335	214274 (0.0%)	209164 (2.4%)	212341 (1.0%)	207335 (3.3%)	212327 (0.9%)	**207315 (3.3%)**	**207315 (3.3%)**	212338 (1.0%)	212338 (1.0%)	207319 (3.3%)
	mem_ctrl	46836	46732 (0.2%)	46554 (0.6%)	46574 (0.6%)	46301 (1.1%)	46484 (0.8%)	**46085 (1.6%)**	46204 (1.4%)	46569 (0.6%)	46569 (0.6%)	46201 (1.3%)
	sqrt	24618	19441 (21.0%)	21690 (11.9%)	23685 (3.8%)	**19221 (21.9%)**	19441 (21.0%)	**19221 (21.9%)**	21582 (12.3%)	23685 (3.8%)	23685 (3.8%)	19221 (21.9%)
	voter	13758	11408 (17.0%)	10997 (20.1%)	12681 (7.8%)	9461 (31.2%)	10982 (20.2%)	**9399 (31.7%)**	9492 (31.0%)	12679 (7.8%)	12679 (7.8%)	9606 (30.2%)
Avg. Node Reduction%			12.7%	7.3%	7.9%	16.6%	15.3%	**16.7%**	13.0%	7.9%	7.9%	16.1%
Avg. Runtime (s)			0.366	0.177	0.155	0.459	0.373	0.454	0.226	0.123	0.137	0.478

Following the topological order, the PIs and nodes n, m, d, and p are bypassed as none of the optimizations are applicable. For node g, the algorithm first checks `rw` as per the priority order but finds it inapplicable, proceeding then to `rf`. Since `rf` is applicable, it is applied to update the AIG (indicated by the blue box), with no need to check the transformability of `rs`. The iterative traversal of the entire graph leads to the AIG, optimized via the *Priority-ordered orchestration* algorithm, achieving 4 node reductions compared to the original graph.

These two orchestration algorithms apply stand-alone optimizations within a single AIG traversal, each leveraging distinct strategies. The *Local-greedy orchestration* algorithm selects the most effective operation for logic minimization based on the current local node structure. In contrast, the *Priority-ordered orchestration* algorithm utilizes a variety of pre-defined priority orders, potentially enhancing overall performance. A key distinction lies in their operational approach: the *Local-greedy orchestration* algorithm examines the transformability with respect to all operations at each node, whereas the *Priority-ordered* algorithm progresses to the next node once an applicable operation is found in the given order, effectively minimizing redundant transformability checks. Consequently, in terms of runtime efficiency, the *Local-greedy orchestration* may be less efficient compared to the *Priority-ordered orchestration*. Detailed empirical studies and discussions of these findings are presented in Section IV.

IV. Experiments

Our experimental results include comparisons with stand-alone optimizations in ABC, covering: **(1)** performance and runtime evaluation with single traversal optimization; **(2)** performance evaluation of optimization methods in iterative synthesis; **(3)** end-to-end performance evaluation in existing ABC flows `resyn` by OpenROAD [23], where the orchestration method is integrated into ABC in Yosys [25] for OpenROAD. OpenROAD reports the performance of area minimization (technology mapping) and post-routing area minimization with respect to different optimization methods. Experimental results are conducted on 104 designs from the ISCAS'85/89/99, VTR[21], and EPFL[22] benchmark suites. All experiments are conducted on an Intel Xeon Gold 6230 20x CPU.

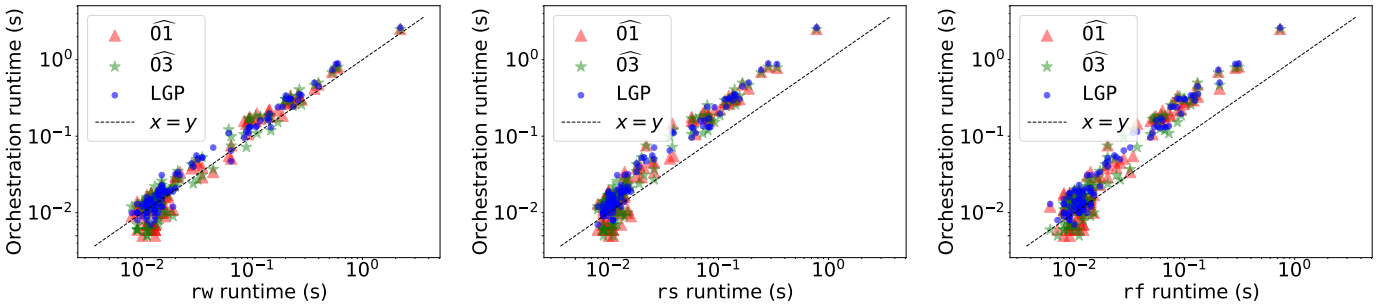
A. Single Optimization Evaluations

Initially, we validate the benefits of the orchestration concept in logic optimization for single AIG traversal. Specifically, we compiled optimization results for all 104 designs from various benchmark suites. These results are related to (1) the *stand-alone optimization* methods, namely `rw`, `rs`, and `rf`, and (2) the *orchestration optimization* methods, which include *priority-ordered orchestration* ($\widehat{\texttt{O1}}$ – $\widehat{\texttt{O6}}$) and *local-greedy orchestration* (`LocalGreedy`). A subset of these results, focusing on large designs, is presented in Table I.

The data indicates that the most effective optimization method for all designs consistently originates from one of the *orchestration* methods, exhibiting notable improvements over *stand-alone* optimizations. Specifically, the best performing *orchestration* algorithm ($\widehat{\texttt{O3}}$) demonstrates an average performance benefit of at least 4.0% compared to stand-alone methods (specifically `rw`). Additionally, the table includes the average runtime cost for each optimization, where the orchestrated algorithms with better optimization performance take runtime overhead at the same time. Specifically, it takes more runtime overhead than `rs` and `rf` while less than `rw`.

B. Single Runtime Evaluations

We also analyze the runtime of the orchestration algorithm *Local-greedy* optimization (`LGP`) and *Priority-*



(a) runtime comparison with `rw` (b) runtime comparison with `rs` (c) runtime comparison with `rf`

Fig. 5: Runtime comparison between selected single-traversal *orchestration* policies ($\widehat{\texttt{O1}}$, $\widehat{\texttt{O3}}$, and `Local-greedy` (labeled as `LGP`)) and *stand-alone* optimizations from ABC: (a) runtime comparison with `rw`; (b) runtime comparison with `rs`; (c) runtime comparison with `rf`.

ordered optimization with $\widehat{\texttt{O1}}$ and $\widehat{\texttt{O3}}$. This analysis, including a comparison with stand-alone optimizations (i.e., `rw`, `rs`, and `rf`), is illustrated for all 104 designs in Figure 5. To effectively showcase the runtime variances, the figure employs a logarithmic scale. The x-axis represents the runtime of the stand-alone ABC optimizations, while the y-axis denotes the runtime of the *orchestration* algorithms. The dotted line ($x = y$) acts as a benchmark, where points above this line indicate a higher runtime cost for the *orchestration* algorithm compared to its stand-alone ABC counterpart. Conversely, points below the line suggest a lower runtime cost. From the runtime data, we draw two main conclusions: **(1)** Generally, orchestration algorithms (with comparable performance, i.e., $\widehat{\texttt{O1}}$, $\widehat{\texttt{O3}}$ and `LGP`) have a comparable runtime, with `LGP` tending to incur a higher runtime overhead than other orchestration methods. **(2)** Orchestration algorithms exhibit runtime overhead when compared to stand-alone optimizations, their runtime is akin to `rw` but notably higher than `rs` and `rf`.

A further analysis of the runtime for each optimization iteration, focusing on different optimization methods, has been conducted. As outlined in Section II-B, logic optimizations predominantly involve two phases: *transformability check* and *graph update*. **Firstly**, the transformability check constitutes the bulk of runtime in logic optimizations. **Secondly**, despite an equal number of total iterations, `rs` and `rf` optimizations are quicker than `rw`, implying that the per iteration runtime cost is lower for `rs` and `rf`. **Thirdly**, a substantial number of iterations are 'wasted' with merely performing transformability checks without contributing to graph optimization. For instance, in Figure 2, design *bfly*, the number of valid iterations is 1764/1374/920 for `rw`/`rs`/`rf`, which is 6%/5%/3% of total iterations, with 94%/95%/97% iterations are wasted. However, with orchestration algorithms, the number of valid iterations is 2306, which is 8% of total iterations with 92% wasted iterations. Despite the orchestration optimization has a higher percentage of valid iterations, it still incurs runtime overhead due to these wasted iterations. Specifically, in orchestration, nodes in wasted iterations undergo transformability checks for all three optimizations, significantly increasing the runtime. Particularly, *Local-greedy* orchestration suffers the most as it requires transformability checks for all optimizations in every iteration. **Consequently**, the runtime inefficiency in orchestration algorithms is mainly due to the substantial number of wasted iterations involving comprehensive transformability checks. The per iteration runtime is heavily influenced by `rw` iterations, leading to an overall runtime overhead for orchestration optimizations compared to stand-alone methods, albeit being comparable to `rw`.

C. Iterative Optimization Evaluations

It is known that DAG-aware synthesis performs better in iterative transformations. However, considering the runtime for fair comparison, in this iterative optimization evaluation, we compare *priority-ordered orchestration* optimization (e.g., $\{\widehat{\texttt{O1}} \rightarrow \widehat{\texttt{O1}} \rightarrow \widehat{\texttt{O1}}\}$, denoted as Seq($\widehat{\texttt{O1}}$)) to the corresponding stand-alone optimization sequence of the priority order (e.g., correspond to $\widehat{\texttt{O1}}$, the sequence is $\{\texttt{rw} \rightarrow \texttt{rs} \rightarrow \texttt{rf}\}$, denoted as Seq(ABC)). We use the same notations and perform experiments on other *Priority-ordered* orchestration algorithms. The results of the iterative-traversal with orchestration algorithms and the corresponding sequence of stand-alone optimizations are shown in Table II. In all permutations of stand-alone optimization sequences, node reduction performance ranges from 16.9% to 17.2%. However, with orchestrated operation sequences, this performance varies between 9.7% and 18.3%. In line with single

TABLE II: Detailed results of selected large size designs. Comparison of Iterative-traversal `orchestration` with corresponding sequence optimizations from ABC.

Design	AIG														
	Baseline	rw → rs → rf	Seq(O1)	rw → rf → rs	Seq(O2)	rs → rw → rf	Seq(O3)	rs → rf → rw	Seq(O4)	rf → rs → rw	Seq(O5)	rf → rw → rs	Seq(O6)	Seq(LGP)	
	#Node	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)	#Node (Δ%)
s38584	12400	10288 (17.0%)	10176 (17.9%)	10230 (17.5%)	10196 (17.8%)	10263 (17.2%)	**10115 (18.4%)**	10219 (17.6%)	10493 (15.4%)	10250 (17.3%)	10793 (13.0%)	10242 (17.4%)	10788 (13.0%)	10187 (17.8%)	
s35932	11948	8561 (28.3%)	8177 (31.6%)	8561 (28.3%)	8177 (31.6%)	8561 (28.3%)	**8177 (31.6%)**	8177 (31.6%)	8177 (31.6%)	8177 (31.6%)	8177 (31.6%)	8177 (31.6%)	8177 (31.6%)	**8177 (31.6%)**	**8129 (32.0%)**
b17_1	27647	22724 (17.8%)	**22503 (18.6%)**	22664 (18.0%)	22620 (18.2%)	22831 (17.4%)	22533 (18.5%)	22841 (17.4%)	23084 (16.5%)	22888 (17.2%)	24203 (12.5%)	22832 (17.4%)	24203 (12.5%)	22506 (18.6%)	
b18_1	79054	62622 (20.8%)	61556 (22.1%)	62425 (21.0%)	62676 (20.7%)	63060 (20.2%)	61568 (22.1%)	63042 (20.3%)	64492 (18.4%)	62552 (20.9%)	68539 (13.3%)	62288 (21.2%)	68540 (13.3%)	**61457 (22.3%)**	
b20	12219	10009 (18.1%)	**9798 (19.8%)**	9972 (18.4%)	9921 (18.8%)	10029 (17.9%)	9799 (19.8%)	9910 (18.9%)	10026 (17.9%)	9933 (18.7%)	10370 (15.1%)	9907 (18.9%)	10372 (15.1%)	9815 (19.7%)	
b21	12782	10150 (20.6%)	**9904 (22.5%)**	10137 (20.7%)	10034 (21.5%)	10175 (20.4%)	9914 (22.4%)	10084 (21.1%)	10163 (20.5%)	10211 (20.1%)	10654 (16.6%)	10033 (21.5%)	10656 (16.6%)	9915 (22.4%)	
b22	18488	14960 (19.1%)	14633 (20.9%)	14891 (19.5%)	14830 (19.8%)	14952 (19.1%)	**14629 (20.9%)**	14856 (19.6%)	14962 (19.1%)	14743 (20.3%)	15654 (15.3%)	14710 (20.4%)	15656 (15.3%)	14676 (20.6%)	
bfly	28910	25914 (10.4%)	25750 (10.9%)	25839 (10.6%)	25945 (10.3%)	26015 (10.0%)	25818 (10.7%)	25931 (10.3%)	26163 (9.5%)	25871 (10.5%)	27373 (5.3%)	25827 (10.7%)	27375 (5.3%)	**25727 (11.0%)**	
dscg	28252	25269 (10.6%)	25093 (11.2%)	25208 (10.8%)	25281 (10.5%)	25377 (10.2%)	25119 (11.1%)	25312 (10.4%)	25566 (9.5%)	25250 (10.6%)	26861 (4.9%)	25175 (10.9%)	26861 (4.9%)	**25052 (11.3%)**	
fir	27704	24751 (10.7%)	24568 (11.3%)	24688 (10.9%)	24818 (10.4%)	24802 (10.5%)	24607 (11.2%)	24757 (10.6%)	24984 (9.8%)	24733 (10.7%)	26191 (5.5%)	24718 (10.8%)	26209 (5.4%)	**24553 (11.4%)**	
syn2	30003	26890 (10.4%)	26708 (11.0%)	26833 (10.6%)	27001 (10.0%)	26962 (10.1%)	26738 (10.9%)	26942 (10.2%)	27188 (9.4%)	26854 (10.5%)	28494 (5.0%)	26810 (10.6%)	28480 (5.1%)	**26700 (11.0%)**	
div	57247	40965 (28.4%)	40869 (28.6%)	40965 (28.4%)	40866 (28.6%)	41006 (28.4%)	40874 (28.6%)	41004 (28.4%)	51414 (10.2%)	41142 (28.1%)	56224 (1.8%)	41104 (28.2%)	56222 (1.8%)	**40849 (28.6%)**	
hyp	214335	207340 (3.3%)	206559 (3.6%)	207343 (3.3%)	211283 (1.4%)	207320 (3.3%)	**206539 (3.6%)**	206648 (3.6%)	207240 (3.3%)	206671 (3.6%)	211991 (1.1%)	206671 (3.6%)	211991 (1.1%)	**206530 (3.6%)**	
mem_ctrl	46836	46177 (1.4%)	45650 (2.5%)	46013 (1.8%)	46005 (1.8%)	46171 (1.4%)	**45360 (3.2%)**	46039 (1.7%)	45855 (2.1%)	46113 (1.5%)	46312 (1.1%)	46077 (1.6%)	46312 (1.1%)	45418 (3.0%)	
sqrt	24618	19327 (21.5%)	19219 (21.9%)	19327 (21.5%)	**19218 (21.9%)**	19333 (21.5%)	19219 (21.9%)	19333 (21.5%)	19223 (21.9%)	19332 (21.5%)	23661 (3.9%)	19328 (21.5%)	23661 (3.9%)	**19217 (21.9%)**	
voter	13758	8755 (36.4%)	**8428 (38.7%)**	9109 (33.8%)	10306 (25.1%)	9056 (34.2%)	8612 (37.4%)	9060 (34.1%)	8589 (37.6%)	9789 (28.8%)	12440 (9.6%)	9870 (28.3%)	12440 (9.6%)	8861 (35.6%)	
Avg. Node Reduction%		17.2%	**18.3%**	17.2%	16.8%	16.9%	**18.3%**	17.3%	15.8%	17.0%	9.7%	17.2%	9.7%	18.2%	
Avg. Runtime (s)		0.443	1.162	0.438	1.149	0.434	1.148	0.430	1.144	0.429	1.143	0.431	1.155	1.162	

TABLE III: Detailed results of selected large size designs. Comparison of `orchestration`-substituted `O-resyn`/`LGP-resyn` with original `resyn` and `resyn3`.

Design		AIG: `resyn`													
		Baseline		`resyn`		`resyn3`		`O-resyn`		`O-resyn3`		`LGP-resyn`		`LGP-resyn3`	
		#Node	Depth	#Node (Δ%)	Depth	#Node (Δ%)	Depth	#Node (Δ%)	Depth	#Node (Δ%)	Depth	#Node (Δ%)	Depth	#Node (Δ%)	Depth
ISCAS	s38584	12400	36	10391 (16.2%)	25	11378 (8.2%)	28	10085 (18.7%)	26	10077 (18.7%)	26	9988 (19.5%)	**24**	**9894 (20.2%)**	**24**
	s35932	11948	19	8518 (28.7%)	12	11916 (0.3%)	19	8177 (31.6%)	13	8177 (31.6%)	13	**8113 (32.1%)**	**11**	**8113 (32.1%)**	**11**
	b17_1	27647	52	23021 (16.7%)	**46**	26067 (5.7%)	47	22046 (20.3%)	47	22011 (20.4%)	47	21475 (22.3%)	**46**	**21459 (22.4%)**	**46**
	b18_1	79054	132	63151 (20.1%)	114	70808 (10.4%)	128	60231 (23.8%)	131	59948 (24.2%)	130	58983 (25.4%)	**127**	**58710 (25.7%)**	**127**
	b20	12219	66	10152 (16.9%)	**64**	11013 (9.9%)	65	9678 (20.8%)	**64**	9622 (21.3%)	**64**	9464 (22.5%)	65	**9304 (23.9%)**	65
	b21	12782	67	10211 (20.1%)	64	11249 (12.0%)	65	9790 (23.4%)	64	9721 (23.9%)	64	9580 (25.1%)	65	**9414 (26.3%)**	**63**
	b22	18488	69	15067 (18.5%)	65	16643 (10.0%)	65	14480 (21.7%)	65	14390 (22.2%)	65	14137 (23.5%)	65	**13910 (24.8%)**	65
VTR	bfly	28910	97	26177 (9.5%)	**68**	26543 (8.2%)	70	25242 (12.7%)	70	25017 (13.5%)	70	24989 (13.6%)	69	**24605 (14.9%)**	69
	dscg	28252	92	25427 (9.9%)	67	25806 (8.7%)	68	24681 (12.6%)	68	24434 (13.5%)	68	24274 (14.1%)	67	**23945 (15.2%)**	**66**
	fir	27704	94	24930 (10.0%)	**67**	25242 (8.9%)	69	24081 (13.1%)	69	23870 (13.8%)	69	23842 (13.9%)	**67**	**23472 (15.3%)**	68
	syn2	30003	93	26911 (10.3%)	**67**	27355 (8.8%)	68	26160 (12.8%)	68	25839 (13.9%)	68	25806 (14.0%)	**67**	**25370 (15.4%)**	**67**
EPFL	div	57247	4372	40889 (28.6%)	**4359**	52336 (8.6%)	4372	40883 (28.6%)	4372	40908 (28.5%)	4372	40796 (28.7%)	4369	**40749 (28.8%)**	4370
	hyp	214335	24801	214240 (0.0%)	24801	208371 (2.8%)	24801	206529 (3.6%)	24801	205734 (4.0%)	24801	206005 (3.9%)	24800	**205182 (4.3%)**	**24799**
	mem_ctrl	46836	114	46611 (0.5%)	111	46484 (0.8%)	114	45676 (2.5%)	114	45190 (3.5%)	114	44063 (5.9%)	111	**42165 (10.0%)**	**108**
	sqrt	24618	5058	19437 (21.0%)	5058	21424 (13.0%)	5058	19219 (21.9%)	5058	19218 (21.9%)	5058	**19217 (21.9%)**	5058	**19217 (21.9%)**	5058
	voter	13758	70	10446 (24.1%)	58	10155 (26.2%)	68	8411 (38.9%)	58	8207 (40.3%)	**57**	8224 (40.2%)	**57**	**8071 (41.3%)**	58
Avg. Node Reduction%				15.7%		8.9%		19.2% (+3.5%)		19.7% (+10.8%)		20.4% (+4.7%)		**21.4% (+11.5%)**	
Avg. Runtime (s)				0.717		0.521		1.148		1.840		1.197		1.908	

TABLE IV: The results reported by OpenROAD with orchestration methods implementation, including the results from logic synthesis, i.e., AIG minimization, technology mapping with nangate 45nm, and post-routing.

	Logic Synthesis (resyn)	Logic Synthesis (LGP-resyn)	Logic Synthesis (O-resyn)	Tech Map (resyn)	Tech Map (LGP-resyn)	Tech Map (O-resyn)	Post-routing (resyn)	Post-routing (LGP-resyn)	Post-routing (O-resyn)
	Node	Node	Node	Area	Area	Area	Area/um^2	Area/um^2	Area/um^2
s38584	10391	9988 (-3.9%)	10085 (-2.9%)	13161.95	13000.48 (-1.2%)	13011.922 (-1.1%)	14313	14013 (-2.1%)	14137 (-1.2%)
s35932	8518	8113 (-4.8%)	8177 (-4.0%)	15368.15	15372.40 (+0.03%)	15368.15 (0)	16045	16055 (+0.06%)	16045 (0)
b17_1	23021	21475 (-6.7%)	22046 (-4.2%)	26798.44	25640.27 (-4.3%)	26019.588 (-2.9%)	29138	28193 (-3.2%)	28561 (-2.0%)
b18_1	63151	58983 (-6.6%)	60231 (-4.6%)	70259.38	68211.18 (-2.9%)	67852.876 (-3.4%)	76811	74499 (-3.0%)	74516 (-3.0%)
b20	10152	9464 (-6.8%)	9678 (-4.6%)	11295.96	10915.57 (-3.3%)	11029.158 (-2.4%)	12247	11861 (-3.1%)	12027 (-1.8%)
b21	10211	9580 (-6.2%)	9790 (-4.1%)	11585.63	11190.09 (-3.4%)	11222.274 (-3.1%)	12596	12246 (-2.7%)	12276 (-2.5%)
b22	15067	14137 (-6.2%)	14480 (-3.9%)	16115.61	15879.67 (-1.4%)	15928.612 (-1.2%)	17856	17404 (-2.5%)	17518 (-1.9%)

traversal results, sequential optimizations using $\widehat{\texttt{O1}}$ and $\widehat{\texttt{O3}}$ surpass their corresponding stand-alone sequences by 1.1% and 1.4% respectively.

Furthermore, we evaluate the performance of the orchestration methods when combined with other orthogonal optimizations in a sequential synthesis flow. Specifically, we evaluate the *orchestration* algorithm in `resyn` and `resyn3` in ABC. The original flow involves iterative transformations such as *rewriting* (`rw`), *resubstitution* (`rs`), *refactoring* (`rf`), and *balance* (`b`). The zero-cost replacement enabled `rw`, `rs`, and `rf` are denoted as `rwz`, `rsz`, and `rfz`, respectively. Similarly for the zero-cost replacement enabled orchestration algorithms are denoted as `Z1` to `Z6`. The optimization flow in `resyn` is {`b;rw;rwz;b;rwz;b`}; the flow of `resyn3` is {`b;rs;rs -K 6;b;rsz;rsz -K 6;b;rsz -K 5;b`}. We follow the permutation of the original flows by replacing the stand-alone optimization with orchestration optimizations to compose orchestration flows. We name the resyn flow where `rw`/`rwz` is replaced with $\widehat{\texttt{O1}}$/$\widehat{\texttt{Z1}}$ (because `rw` has the highest priority in $\widehat{\texttt{O1}}$) as `O-resyn`, the resyn3 flow where `rs` is replaced with $\widehat{\texttt{O3}}$ (because `rs` has the highest priority in $\widehat{\texttt{O3}}$) as `O-resyn3`, and the resyn/resyn3 flow where `rw`(`rwz`)/`rs` is replaced with `Local-greedy`(`Local-greedy-z`) as `LGP-resyn`/`LGP-resyn3`.

Table III shows the AIG optimization results of 16 designs. Upon comparing the average node reduction of each optimization option, we can observe a consistent improvement with orchestration synthesis flows. Specif-

ically, `O-resyn` and `LGP-resyn` outperforms `resyn` by 3.5% and 4.7% more average node reductions, respectively; and `O-resyn3` and `LGP-resyn3` with 10.8% and 11.5% more average node reductions than `resyn3`.

D. End-to-end Evaluations

Finally, we integrate our proposed orchestration optimization methods into the end-to-end design framework OpenROAD (Open Resilient Design for Autonomous Systems) [23] to evaluate the end-to-end performance by the orchestration improved logic synthesis. OpenROAD [23] Project is an open-source project aiming at developing a comprehensive, end-to-end, automated IC (Integrated Circuit) design flow that supports a wide range of design styles and technology nodes. It integrates various open-source tools to streamline chip development. The flow begins with RTL synthesis, where Yosys [25] converts high-level RTL descriptions into gate-level netlists and performs logic synthesis and technology mapping via ABC [5]. As shown in Figure 6, this is the specific integration where we deploy our proposed orchestration methods in ABC in the end-to-end design flow (the dash line box). Next, the OpenROAD flow performs floorplanning, placement, and global routing. Tools such as RePlAce, TritonRoute, and FastRoute are used for these tasks, respectively. Afterward, detailed routing and signoff checks are completed, using tools like OpenROAD's built-in router and Magic.

1) Technology Mapping: We have implemented AIG technology mapping for standard cells using the 45nm Nangate library [26] and applied `resyn`, `O-resyn`, and `LGP-resyn` across all 104 designs in a consistent environment. Selected results for 7 detailed cases are presented in Table IV (columns 5 – 7), with the technology mapping outcomes reported by Yosys in OpenROAD. Generally, flows incorporating orchestration optimizations tend to yield better area minimization, averaging 2.2% more area reduction. This suggests the potential of integrating *orchestration* into existing synthesis flows for enhanced technology mapping performance. However, an exception is observed in the case of *s35932*, where although orchestration-enhanced `resyn` flows surpass the original `resyn` in AIG reduction, they result in larger areas post-technology mapping.

Furthermore, a comparison between the post-technology mapping results and those from logic synthesis reveals that the benefits gained from orchestration methods during logic synthesis can diminish, disappear, or even turn into drawbacks after technology mapping. This discrepancy likely arises from the misalignment between technology-independent logic synthesis and technology-dependent mapping cost models, attributable to the high-level abstractions involved at the logic level.

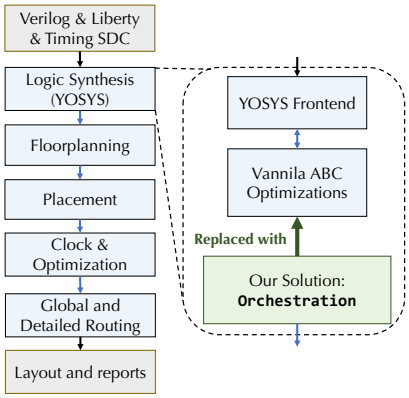


Fig. 6: The OpenROAD framework integrated with proposed orchestration methods. The dash line blue box shows the details in logic synthesis where the original ABC is replaced with our proposed orchestration optimization implemented ABC.

2) Post-Routing: Furthermore, we carry out post-routing evaluations in OpenROAD, applying the three `resyn` flows to various designs. The results, detailed in the last three columns of Table IV, indicate that the orchestration-enhanced flows (`O-resyn` and `LGP-resyn`) generally maintain superiority over the original `resyn` across most designs. However, the margin of this superiority is reduced when compared to the gains observed in logic synthesis. For instance, in the case of the design *b21*, the `LGP-resyn` flow demonstrates a 6.2% improvement in AIG reduction, but this advantage is reduced to 2.7% in terms of area minimization following post-routing. A notable exception is observed in the design *s35932*, where, despite a 4.8% improvement in AIG reduction with orchestration methods, the post-routing area minimization performance degrades. This trend, similar to what was observed in technology mapping, underscores the potential misalignments between the benefits achieved during technology-independent logic synthesis and the outcomes post technology-dependent mapping and routing stages.

In conclusion, our study reveals a modest correlation between the improvements achieved in logic optimization and the enhancements in post-routing performance. However, there is a more pronounced connection between the results following technology mapping and those observed in the post-routing stage. This finding motivates the focus of our future research on developing technology-aware logic synthesis approaches, aiming to align more closely with the subsequent stages of technol-

ogy mapping and routing, thereby enhancing the overall design efficiency.

V. Conclusion

In this work, we propose a novel concept in logic synthesis development – DAG-aware synthesis *orchestration*, which encompasses multiple optimization operations within a single AIG traversal. The proposed concept is implemented in ABC, orchestrating the pre-exisiting stand-alone optimizations, namely *rewriting*, *resubstitution*, *refactoring* for fine-grained node-level logic optimization within a single AIG traversal. Specifically, we provide two algorithms for this *orchestration* process: (1) The *Local-greedy orchestration* algorithm, which selects the optimization operation offering the highest local gain at each node for AIG optimization; (2) The *Priority-ordered orchestration* algorithm, which employs a predefined priority order to select the optimization operation at each node. Our implementations have been rigorously tested on 104 designs from benchmark suites such as ISCA'85/89/99, VTR, and EPFL. In comparison to conventional stand-alone optimizations, our *orchestration* optimization achieves superior performance with a reasonable runtime overhead during single graph traversal. Additionally, this optimization maintains its performance benefits in iterative optimizations and integrated design flows, such as `resyn`, when combined with other optimizations like *balance*. Notably, when implemented within an end-to-end design flow, the orchestration algorithm surpasses stand-alone optimizations in technology mapping and post-routing for the majority of designs. However, it is important to note the observed discrepancies between technology-independent stages (e.g., logic synthesis) and technology-dependent stages (e.g., technology mapping and post-routing). These observations have spurred our interest in future research, specifically aiming to develop end-to-end aware DAG-aware synthesis *orchestrations* that address these optimization miscorrelations.

References

[1] P. Bjesse and A. Boralv, "Dag-aware circuit compression for formal verification," in *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* IEEE, 2004, pp. 42–49.

[2] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süsstrunk, and G. De Micheli, "Deep learning for logic optimization algorithms," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–4.

[3] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 1990.

[4] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Transactions on CAD*, vol. 35, no. 5, pp. 806–819, 2015.

[5] A. Mishchenko *et al.*, "Abc: A system for sequential synthesis and verification," *URL http://www. eecs. berkeley. edu/alanmi/abc*, vol. 17, 2007.

[6] C. Yu, "Flowtune: Practical multi-armed bandits in boolean optimization," in *International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[7] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[8] S. Rai *et al.*, "Logic synthesis meets machine learning: Trading exactness for generalization," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1026–1031.

[9] Y.-S. Huang, J.-H. R. Jiang, and A. Mishchenko, "Quantized neural network synthesis for direct logic circuit implementation," *IEEE Transactions on CAD (TCAD)*, 2022.

[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," in *Design Automation Conference (DAC)*, 2006, pp. 532–535.

[11] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "Fraigs: A unifying representation for logic synthesis and verification," ERL Technical Report, Tech. Rep., 2005.

[12] C. Yu, M. Ciesielski, M. Choudhury, and A. Sullivan, "Dag-aware logic synthesis of datapaths," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[13] M. Soeken, L. G. Amaru, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Transactions on CAD (TCAD)*, 2017.

[14] Ç. Çalık, M. Sönmez Turan, and R. Peralta, "The multiplicative complexity of 6-variable boolean functions," *Cryptography and Communications*, vol. 11, no. 1, pp. 93–107, 2019.

[15] W. Haaswijk, M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic rewriting," in *ASP-DAC*. Ieee, 2017, pp. 151–156.

[16] H. Riener, S.-Y. Lee, A. Mishchenko, and G. De Micheli, "Boolean Rewriting Strikes Back: Reconvergence-Driven Windowing Meets Resynthesis," in *ASP-DAC*, 2022.

[17] W. Haaswijk, A. Mishchenko, M. Soeken, and G. De Micheli, "SAT based exact synthesis using DAG topology families," in *DAC*, 2018.

[18] R. B. Alan Mishchenko, "Scalable logic synthesis using a simple circuit structure," in *Proc. IWLS*, vol. 6, 2006, pp. 15–22.

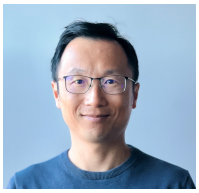
[19] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1989.

[20] S. Davidson, "Itc'99 benchmark circuits-preliminary results," in *International Test Conference 1999. Proceedings (IEEE Cat. No. 99CH37034)*. IEEE, 1999, pp. 1125–1125.

[21] K. E. Murray *et al.*, "VTR 8: High-performance CAD and Customizable FPGA Architecture Modelling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–55, 2020.
[22] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The epfl logic synthesis libraries," *arXiv preprint arXiv:1805.05121*, 2018.
[23] T. Ajayi, D. Blaauw, T. Chan, C. Cheng, V. Chhabria, D. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça *et al.*, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," *Proc. GOMACTECH*, pp. 1105–1110, 2019.
[24] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "Drills: Deep reinforcement learning for logic synthesis," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2020, pp. 581–586.
[25] C. Wolf, "Yosys open synthesis suite," 2016.
[26] J. Knudsen, "Nangate 45nm open cell library," *CDNLive, EMEA*, 2008.

Yingjie Li (Student Member, IEEE) is currently a fourth-year PhD candidate in Computer Engineering at the University of Maryland, College Park under the supervision of Prof. Cunxi Yu. Her research focuses on physics-aware infrastructure for optical computing platforms, hardware-software co-design, and efficient AI/ML algorithms. She also works in electronic design automation (EDA), focusing on machine learning for synthesis and verification. She received B.S. degree in 2018 from Huazhong University of Science and Technology in Wuhan, China, and her master degree from Cornell University in 2019. Her work received the Best Paper Award at DAC (2023), American Physical Society DLS poster award (2022) and Best Poster Presentation Award at DAC Young Fellow (2020). Yingjie won the Second Place at the ACM/SIGDA Student Research Competition (2023) and was selected as the EECS Rising Star (2023).

Mingju Liu (Student Member, IEEE) is currently a second-year PhD student in Computer Engineering at the University of Maryland, College Park under the supervision of Prof. Cunxi Yu. His research spans across hardware-software co-design of deep learning algorithms, focusing on Electronic Design Automation (EDA) challenges. He also works on projects in logic synthesis and combinatorial optimization. He received his B.S. degree in 2020 from the University of Electronic Science and Technology of China in Chengdu, China, and his master's degree from Rutgers University in 2021. He was selected into the DAC Young Fellows Program in 2023.



Haoxing Ren (Fellow, IEEE) is the Director of Design Automation Research at NVIDIA, focusing on leveraging machine learning and GPU-accelerated tools to enhance chip design quality and productivity. Before joining NVIDIA in 2016, he dedicated 15 years to EDA algorithm research and design methodology innovation at IBM Microelectronics and IBM Research. Mark is widely recognized for his contributions to physical design, AI, and GPU acceleration for EDA, which have earned him several prestigious awards, including the IBM Corporate Award and best paper awards at ISPD, DAC, TCAD, and MLCAD. He holds over twenty patents and has co-authored over 100 papers and books, including a book on ML for EDA and several book chapters in physical design and logic synthesis. He holds Bachelor's and Master's degrees from Shanghai Jiao Tong University and Rensselaer Polytechnic Institute, respectively, and earned his Ph.D. from the University of Texas at Austin. He is a Fellow of the IEEE.

Alan Mishchenko (Senior Member, IEEE)received the M.S. degree from the Moscow Institute of Physics and Technology, Moscow, Russia, in 1993 and the Ph.D. degree from the Glushkov Institute of Cybernetics, Kiev, Ukraine, in 1997. In 2002, he joined the EECS Department, University of California at Berkeley, Berkeley, CA, USA, where he is currently a Full Researcher. His current research interests include computationally efficient logic synthesis, formal verification, and machine learning.

Cunxi Yu (Member, IEEE) is an Assistant Professor in the ECE Department at the University of Maryland, College Park. His research interests focus on novel algorithms, systems, and hardware designs for computing and security. Before joining University of Maryland, Cunxi was an Assistant Professor at University of Utah, PostDoc at Cornell University in 2018-2019 and EPFL in 2017-2018, and was a research intern at IBM T.J Watson Research Center (2015, 2016). He received Ph.D. degree from UMass Amherst in 2017. His work received the best paper nomination at ASP-DAC (2017), TCAD Best paper nomination (2018), 1st place at DAC Security Contest (2017), NSF CAREER Award (2021), DLS Best Poster Honorable Mention at (2022), and Best Paper Award at DAC (2023). He served as Organizing Committee in IWLS, ICCD, VLSI-SoC, ASAP, as a TPC member in ICCAD, DATE, ASP-DAC, DAC, and General Chair of IWLS 2023.