A APPENDIX 1

A.1 LOGIC SYNTHESIS

Logic synthesis transforms a hardware design in register transfer level (RTL) to a Boolean gate-level network, optimizes the number of gates/depth, and then maps it to standard cells in a technology library Brayton et al. (1984). Well-known representations of Boolean networks include sum-of-product form, product-of-sum, Binary decision diagrams, and AIGs which are a widely accepted format using only AND (nodes) and NOT gates (dotted edges). Several logic minimization heuristics (discussed in Section A.2)) have been developed to perform optimization on AIG graphs because of its compact circuit representation and directed acyclic graph (DAG)-based structuring. These heuristics are applied sequentially ("synthesis recipe") to perform one-pass logic optimization reducing the number of nodes and depth of AIG. The optimized network is then mapped using cells from technology library to finally report area, delay and power consumption.

A.2 LOGIC MINIMIZATION HEURISTICS

We now describe optimization heuristics provided by industrial strength academic tool ABC Brayton & Mishchenko (2010):

1. Balance (b) optimizes AIG depth by applying associative and commutative logic function tree-balancing transformations to optimize for delay.

2. Rewrite (rw, rw -z) is a directed acyclic graph (DAG)-aware logic rewriting technique that performs template pattern matching on sub-trees and encodes them with equivalent logic functions.

3. Refactor (rf, rf -z) performs aggressive changes to the netlist without caring about logic sharing. It iteratively examines all nodes in the AIG, lists out the maximum fan-out-free cones, and replaces them with equivalent functions when it improves the cost (e.g., reduces the number of nodes).

4. Re-substitution (rs, rs -z) creates new nodes in the circuit representing intermediate functionalities using existing nodes; and remove redundant nodes. Re-substitution improves logic sharing.

The zero-cost (-z) variants of these transformation heuristics perform structural changes to the netlist without reducing nodes or depth of AIG. However, previous empirical results show circuit transformations help future passes of other logic minimization heuristics reduce the nodes/depth and achieve the minimization objective.

A.3 MONTE CARLO TREE SEARCH

We discuss in detail the MCTS algorithm. During selection, a search tree is built from the current state by following a search policy, with the aim of identifying promising states for exploration.

where $Q^k_{MCTS}(s,a)$ denotes estimated Q value (discussed next) obtained after taking action a from state s during the k^{th} iteration of MCTS simulation. $U^k_{MCTS}(s,a)$ represents upper confidence tree (UCT) exploration factor of MCTS search.

$$U^k_{MCTS}(s,a) = c_{\text{UCT}} \sqrt{\frac{\log\left(\sum_a N^k_{MCTS}(s,a)\right)}{N^k_{MCTS}(s,a)}}, \tag{5}$$

$N^k_{MCTS}(s,a)$ denotes the visit count of the resulting state after taking action a from state s. c_{UCT} denotes a constant exploration factor Kocsis & Szepesvári (2006).

The selection phase repeats until a leaf node is reached in the search tree. A leaf node in MCTS tree denotes either no child nodes have been created or it is a terminal state of the environment. Once a leaf node is reached the expansion phase begins where an action is picked randomly and its roll out value is returned or $R(s_L)$ is returned for the terminal state s_L. Next, back propagation happens where all parent nodes $Q_k(s,a)$ values are updated according to the following equation.

$$Q^k_{MCTS}(s,a) = \sum_{i=1}^{N^k_{MCTS}(s,a)} R^i_{MCTS}(s,a)/N^k_{MCTS}(s,a). \tag{6}$$

A.4 ABC-RL AGENT PRE-TRAINING PROCESS

As discussed in Section 2.3, we pre-train an agent using available past data to help with choosing which logic minimization heuristic to add to the synthesis recipe. The process is shown as Algorithm 1.

Algorithm 1 ABC-RL: Policy agent pre-training

procedure TRAINING(θ)
  Replay buffer $(RB) \leftarrow \phi$, $\mathcal{D}_{train} = \{AIG_1, AIG_2, ..., AIG_n\}$, num_epochs=$N$, Recipe length=$L$, AIG embedding network: Λ, Recipe embedding network: $\mathcal{R}$, Agent policy $\pi_\theta := U$ (Uniform distribution), MCTS iterations = K, Action space = A
  for $AIG_i \in \mathcal{D}_{train}$ **do**
    $r \leftarrow 0$, $depth \leftarrow 0$
    $s \leftarrow \Lambda(AIG_i) + \mathcal{R}(r)$
    while depth < L **do**
      $\pi_{MCTS} = MCTS(s, \pi_\theta, K)$
      $a = argmax_{a' \in A} \pi_{MCTS}(s, a')$
      $r \leftarrow r + a$, $s' \leftarrow \mathcal{A}(AIG_i) + \mathcal{R}(r)$
      $RB \leftarrow RB \bigcup (s, a, s', \pi_{MCTS}(s, \cdot))$
      $s \leftarrow s', depth \leftarrow depth + 1$
  for epochs < N **do**
    $\theta \leftarrow \theta_i - \alpha \nabla_\theta \mathcal{L}(\pi_{MCTS}, \pi_\theta)$

B NETWORK ARCHITECTURE

B.1 AIG NETWORK ARCHITECTURE

Starting with a graph G = ($\mathbf{V}, \mathbf{E}$) that has vertices $\mathbf{V}$ and edges $\mathbf{E}$, the GCN aggregates feature information of a node with its neighbors' node information. The output is then normalized using `Batchnorm` and passed through a non-linear `LeakyReLU` activation function. This process is repeated for k layers to obtain information for each node based on information from its neighbours up to a distance of k-hops. A graph-level READOUT operation produces a graph-level embedding. Formally:

$$h_u^k = \sigma(W_k \sum_{i \in u \cup N(u)} \frac{h_i^{k-1}}{\sqrt{N(u)} \times \sqrt{N(v)}} + b_k), k \in [1..K] \tag{7}$$
$$h_G = READOUT(\{h_u^k; u \in V\})$$

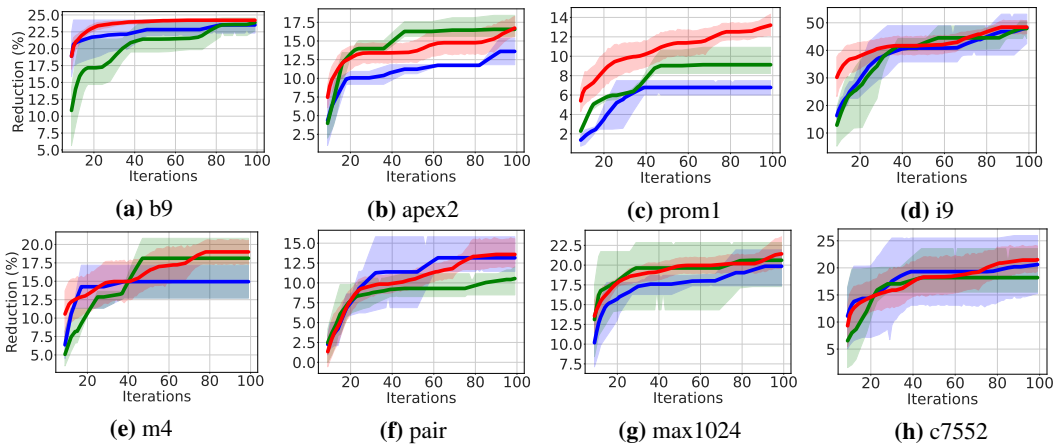
The embedding for node u, generated by the k^{th} layer of the GCN, is represented by h_u^k. The parameters W_k and b_k are trainable, and σ is a non-linear ReLU activation function. $N(\cdot)$ denotes the 1-hop neighbors of a node. The READOUT function combines the activations from the k^{th} layer of all nodes to produce the final output by performing a pooling operation. In our work, we choose $k = 2$ and global average and max pooling concatenated as READOUT operation.

C EXPERIMENTAL DETAILS

C.1 REWARD NORMALIZATION

In our work, maximizing QoR entails finding a recipe P which is minimizing the area-delay product of transformed AIG graph. We consider as a baseline recipe an expert-crafted synthesis recipe `resyn2` Mishchenko et al. (2006) on top of which we improve our ADP.

$$R = \begin{cases} 1 - \frac{ADP(\mathcal{S}(G,P))}{ADP(\mathcal{S}(G,resyn2))} & ADP(\mathcal{S}(G,P)) < 2 \times ADP(\mathcal{S}(G,P)), \\ -1 & otherwise. \end{cases}$$



(a) b9 **(b)** apex2 **(c)** prom1 **(d)** i9

(e) m4 **(f)** pair **(g)** max1024 **(h)** c7552

Figure 7: Area-delay product reduction (in %) compared to resyn2 on MCNC circuits. GREEN: SA+Pred. Chowdhury et al. (2022), BLUE: MCTS Neto et al. (2022), RED: ABC-RL

D RESULTS

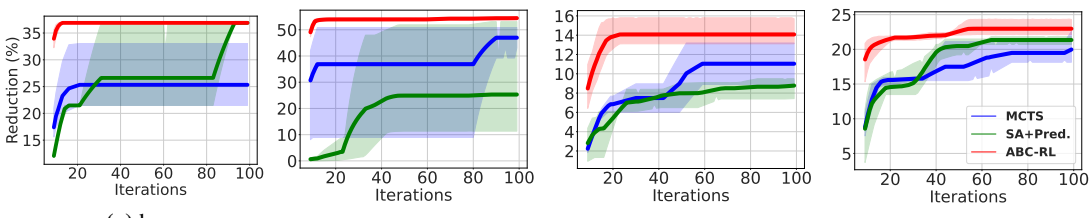
D.1 PERFORMANCE OF ABC-RL AGAINST PRIOR WORKS AND BASELINE MCTS+LEARNING

D.1.1 MCNC BENCHMARKS

Figure 7 plots the ADP reductions over search iterations for MCTS, SA+Pred, and ABC-RL. In `m4`, ABC-RL's agent explores paths with higher rewards whereas standard MCTS continues searching without further improvement. A similar trend is observed for `prom1` demonstrating that a pre-trained agent helps bias search towards better parts of the search space. SA+Pred. Chowdhury et al. (2022) also leverages past history, but is unable to compete (on average) with MCTS and ABC-RL in part because SA typically underperforms MCTS on tree-based search spaces. Also note from Figure 5 that ABC-RL in most cases achieves higher ADP reductions earlier than competing methods (except `pair`). This results in significant geo. mean run-time speedups of $2.5\times$ at iso-QoR compared to standard MCTS on MCNC benchmarks.

D.1.2 EPFL ARITHMETIC BENCHMARKS

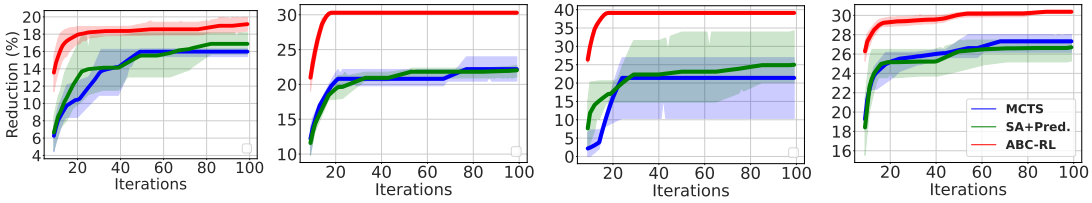
Figure 8 illustrates the performance of ABC-RL in comparison to state-of-the-art methods: Pure MCTS Neto et al. (2022) and SA+Prediction Chowdhury et al. (2022). In contrast to the scenario where MCTS+Baseline underperforms pure MCTS (as shown in 2), here we observe that ABC-RL effectively addresses this issue, resulting in superior ADP reduction. Remarkably, ABC-RL achieved a geometric mean $5.8\times$ iso-QoR speed-up compared to MCTS across the EPFL arithmetic benchmarks.



(a) bar **(b)** div **(c)** square **(d)** sqrt

Figure 8: Area-delay product reduction (in %) compared to resyn2 on EPFL arithmetic benchmarks. GREEN: SA+Pred. Chowdhury et al. (2022), BLUE: MCTS Neto et al. (2022), RED: ABC-RL

D.1.3 EPFL RANDOM CONTROL BENCHMARKS



(a) cavlc **(b)** mem_ctrl **(c)** router **(d)** voter

Figure 9: Area-delay product reduction (in %) compared to resyn2 on EPFL random control benchmarks. On `cavlc` and `router`, ABC-RL perform better than MCTS where baseline MCTS+Learning under-perform. GREEN: SA+Pred. Chowdhury et al. (2022), BLUE: MCTS Neto et al. (2022), RED: ABC-RL.

D.2 PERFORMANCE OF BENCHMARK-SPECIFIC ABC-RL AGENTS

ABC-RL+MCNC agent: For 6 out of 12 MCNC benchmarks, ABC-RL guided by the MCNC agent demonstrated improved performance compared to the benchmark-wide agent. This suggests that the hyper-parameters (δ_{th} and T) derived from the validation dataset led to optimized α values for MCNC benchmarks. However, the performance of the MCNC agent was comparatively lower on EPFL arithmetic and random control benchmarks.

ABC-RL+ARITH agent: Our EPFL arith agent resulted in better ADP reduction compared to benchmark-wide agent only on A4(`sqrt`). This indicate that benchmark-wide agent is able to learn more from diverse set of benchmarks resulting in better ADP reduction. On MCNC benchmarks, we observe that ARITH agent performed the best amongst all on C6(`m4`) and C10 (`c7552`) because these are arithmetic circuits.

ABC-RL+RC agent: Our RC agent performance on EPFL random control benchmarks are not that great compared to benchmark-wide agent. This is primarily because of the fact that EPFL random control benchmarks have hardware designs performing unique functionality and hence learning from history doesn't help much. But, ABC-RL ensures that performance don't detoriate compared to pure MCTS.

D.3 ABC-RL VERSUS MCTS+L+FT

MCNC Benchmarks: In Fig. 10, we depict the performance comparison among MCTS+finetune agent, ABC-RL, and pure MCTS. Remarkably, ABC-RL outperforms MCTS+finetune on 11 out of 12 benchmarks, approaching MCTS+finetune's performance on `b9`. A detailed analysis of circuits where MCTS+finetune performs worse than pure MCTS (`i9, m4, pair, c880, max1024,` and `c7552`) reveals that these belong to 6 out of 8 MCNC designs where MCTS+learning performs suboptimally compared to pure MCTS. This observation underscores the fact that although finetuning contributes to a better geometric mean over MCTS+learning (23.3% over 20.7%), it still falls short on 6 out of 8 benchmarks. For the remaining two benchmarks, `alu4` and `apex4`, MCTS+finetune performs comparably to pure MCTS for `alu4` and slightly better for `apex4`. Thus, ABC-RL emerges as a more suitable choice for scenarios where fine-tuning is resource-intensive, yet we seek a versatile agent capable of appropriately guiding the search away from unfavorable trajectories.

EPFL Arithmetic Benchmarks: In Fig. 11, we present the performance comparison with MCTS+finetune. Notably, for designs `bar` and `div`, MCTS+finetune achieved equivalent ADP as ABC-RL, maintaining the same iso-QoR speed-up compared to MCTS. These designs exhibited strong performance with baseline MCTS+Learning, thus aligning with the expectation of favorable results with MCTS+finetune. On `square`, MCTS+finetune nearly matched the ADP reduction achieved by pure MCTS. This suggests that fine-tuning contributes to policy improvement from the pre-trained agent, resulting in enhanced performance compared to baseline MCTS+Learning. In the case of `sqrt`, MCTS+finetune approached the performance of ABC-RL. Our fine-tuning experiments affirm its ability to correct the model policy, although it require more samples to converge towards ABC-RL performance.

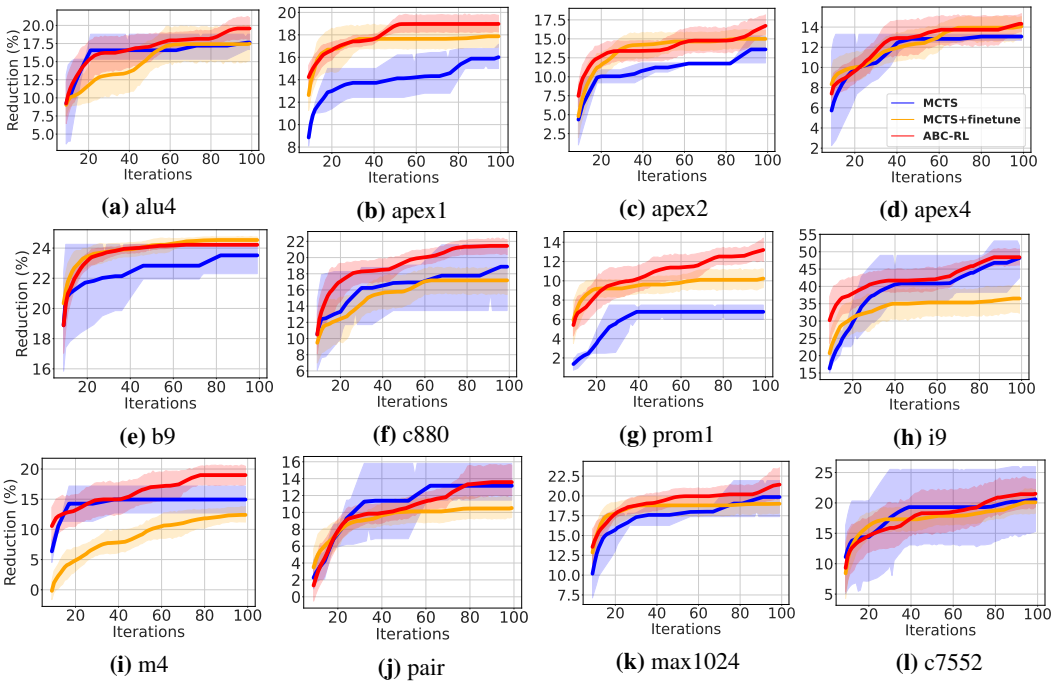


Figure 10: Area-delay product reduction (in %) compared to resyn2 on MCNC benchmarks. YELLOW: MCTS+Finetune, BLUE: MCTS Neto et al. (2022), RED: ABC-RL

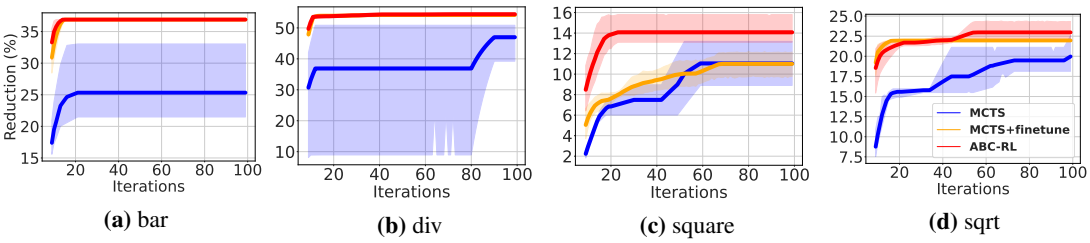


Figure 11: Area-delay product reduction (in %) compared to resyn2 on EPFL arithmetic benchmarks. YELLOW: MCTS+Finetune, BLUE: MCTS Neto et al. (2022), RED: ABC-RL

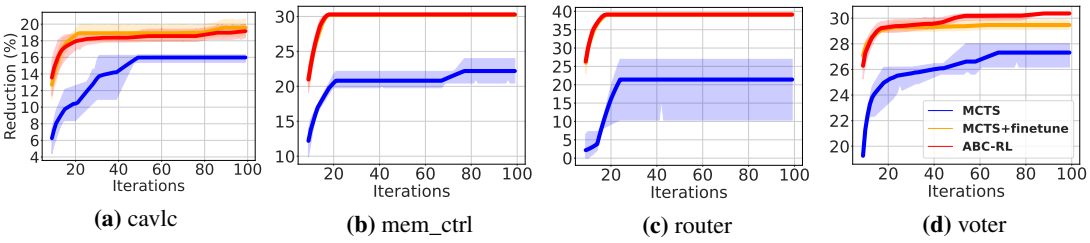


Figure 12: Area-delay product reduction (in %) compared to resyn2 on EPFL random control benchmarks. YELLOW: MCTS+FT, BLUE: MCTS Neto et al. (2022), RED: ABC-RL