Node Reduction via Targeted ODC-Based Wire Additions for AIG Optimization

Zhuofan Lin
Thrust of Microelectronics
Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
zlin037@connect.hkust-gz.edu.cn

Shiju Lin
Thrust of Microelectronics
Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
shijulin@hkust-gz.edu.cn

Abstract

High-effort logic optimization methods such as *(GPU-)transduction* and *rewiring* are based on redundancy addition and removal (RAR) to achieve high-quality optimization results for AIGs. Despite their strong optimization capability, these methods follow a speculative process in which legal wire additions are applied before their reduction effects are known. Many wire additions fail to expose profitable reductions, yet still incur temporary updates to the AIG and trigger recomputation of frequently used data such as observability don't-cares (ODCs) and truth tables. Moreover, because these wire additions are guided mainly by legality rather than explicit reduction targets, the resulting restructuring becomes increasingly ineffective as AIG size grows. This paper extracts the effective action embedded in successful RAR trials and reorganizes it into a targeted workflow. Our method selects an explicit reduction target before ODC analysis and wire search, and jointly determines the wire additions and target removal before updating the AIG. On the IWLS 2022, 2023, and 2024 benchmark suites, our method achieves 45.2%–49.1% converged size reduction with the best depth results, compared with 43.3%–48.3% for *GPU-transduction* and 33.0%–40.1% for *rewiring*; while converging in 3.0–4.3s, compared to over 2800s and 11.8–16.3s, respectively. The source code is publicly available at https://github.com/Lin-HKUST-Guangzhou/target-reduction.

1 Introduction

Logic synthesis is a critical stage in modern integrated circuit design, transforming high-level hardware descriptions into optimized gate-level implementations. AIGs are widely used to represent Boolean networks in technology-independent synthesis, where minimizing the number of nodes is a primary objective for reducing circuit area and enabling better downstream optimization. Over the past decades, efficient AIG optimization techniques, such as *rewriting* [6, 8], *resubstitution* [11], and *refactoring* [5], have become the foundation of AIG optimization. However, because these techniques search for transformations within small bounded local contexts and typically commit immediate area reductions, iterative flows can become trapped in local minima.

Corresponding author: *Shiju Lin*

This work is licensed under a Creative Commons Attribution 4.0 International License.
ASPDAC '27, Tokyo, Japan
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2923-2/2027/01
https://doi.org/10.1145/3840404.3848165

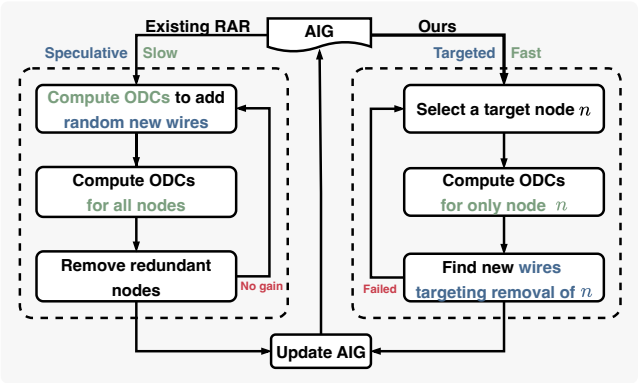

Figure 1: Comparison of the existing RAR flow and ours.

To overcome this limitation, recent high-effort AIG optimization methods have revisited RAR [3] as a powerful optimization paradigm. RAR optimizes an AIG by first introducing redundancies that temporarily increase area, thereby expanding the structural search space for subsequent reductions. *(GPU-)Transduction* [7, 9] and *rewiring* [2] are two existing RAR-based high-effort optimization methods that operate on AIGs or subcircuits with a limited number of primary inputs (PIs). Both methods leverage ODCs to perform area-increasing transformations for AIG minimization. *(GPU-)Transduction* mainly uses single-node multi-wire addition, while *rewiring* explores a broader optimization space through multi-node multi-wire additions and versatile reduction strategies. These two methods demonstrate that RAR can substantially improve optimization quality when sufficient optimization effort is available.

Figure 1 compares the workflows of existing RAR implementations, *GPU-transduction* and *rewiring*, with ours. These two methods share a highly speculative process. In each iteration, they simulate the circuit and compute ODCs before adding new wires that preserve the AIG function. After the AIG is updated, the ODCs must be recomputed before the subsequent reduction procedure searches for redundant nodes. If the added wires expose no reduction opportunity, the entire iteration is wasted. Such waste is particularly costly because ODC computation and simulation account for a large fraction of runtime. Beyond the runtime issue, legality-driven wire additions introduce redundancies through a largely unguided and weakly controlled transformation process. Although heuristics may prioritize candidate wires, they do not explicitly control whether the resulting transformation benefits subsequent reductions.

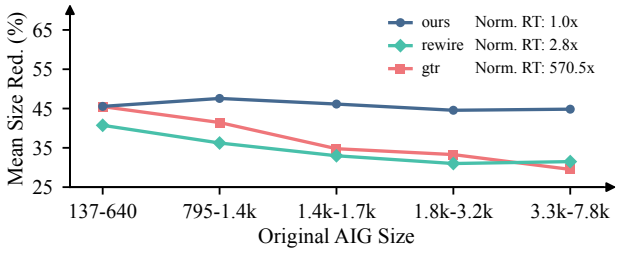

Figure 2: Mean size reduction percentage versus original AIG size on all 55 benchmarks with 15 primary inputs drawn from the IWLS 2022, 2023, and 2024 benchmark suites. The benchmarks are grouped into five equal-size bins, each containing 11 benchmarks.

In the preliminary experiments shown in Figure 2, each method is run until convergence on all 55 benchmarks with 15 PIs from the IWLS benchmark suites. The size reductions achieved by *GPU-transduction* (`gtr`) [7] and *rewiring* (`rewire`) [2] decline as the AIG size increases, indicating that stochastic restructuring becomes less effective at consistently producing high-quality results on larger AIGs.

To address these limitations on AIGs of varying sizes with the same number of PIs, we propose the targeted ODC-based wire addition technique shown in Figure 1, which reorganizes how wire addition and node removal are performed in existing RAR methods. Rather than first applying legality-driven wire additions and then searching the modified AIG for removable nodes, the proposed method begins with an explicit reduction target, derives the ODC conditions required for its removal, and searches only for wires that satisfy those conditions. Wire addition and target removal are thus jointly determined and committed as a single transformation. If no complete transformation is found, the method moves to another target without modifying the current AIG. This target-oriented workflow preserves the reduction-enabling principle of RAR while avoiding speculative intermediate updates and redundant recomputation. Preliminary results in Figure 2 support this design: our method maintains a substantial and stable size reduction as the AIG size increases at the lowest runtime cost.

The main contributions of this paper are:

- We identify node reduction through targeted wire additions as a useful action pattern within ODC-based RAR, and derive the care-on preservation and required care-off coverage conditions that make a target fanin removable.
- We develop a targeted ODC-based wire addition algorithm that computes the required pattern set for a target fanin, filters function-preserving candidate wires, and selects single- or multi-wire additions with a gain-aware commit rule.
- We evaluate our method on the IWLS 2022, 2023, and 2024 benchmark suites against GPU-transduction and rewiring. From `strash` starting points, our method achieves the highest converged size reduction while maintaining best depth reduction and converging faster; from heavily optimized starting points, it provides greater additional reduction with less depth degradation.

2 Background

This section introduces the background for node reduction via targeted ODC-based wire additions, including *(multi-input) AIGs*, *observability don't-cares*, and *redundancy addition and removal*.

2.1 (Multi-Input) AIGs

Technology-independent synthesis commonly uses AIGs to represent Boolean logic networks. An AIG is a directed acyclic graph whose internal nodes are two-input AND gates and whose edges may be complemented to represent inversion. Primary inputs (PIs) are source nodes with no fanins, and primary outputs (POs) are driven by PIs or internal nodes. The number of internal AND nodes (size) and the logic level (depth) are commonly used as quality metrics for AIG optimization.

To provide flexibility during wire addition and redundancy removal, many high-effort AIG optimization methods use a *multi-input AIG* (MIAIG) as an intermediate representation [2, 7, 9, 10]. Unlike a standard AIG, where each internal node has exactly two fanins, an MIAIG allows an internal AND node to have multiple fanins. This representation can directly capture temporary wire additions as extra fanins of an AND node. After optimization, multi-input nodes can be decomposed into two-input AND nodes and structurally hashed back into a standard AIG.

2.2 Observability Don't-Cares

For an (MI)AIG node n, an input pattern p is an ODC if negating the value of n under p does not change any PO value. Therefore, the ODC set of n, denoted by $S_{\text{ODC}}(n)$, contains the input patterns under which a change at n is not observable at any PO. Equivalently, the *care set* of n, denoted as $S_{\text{care}}(n)$, is the complement of $S_{\text{ODC}}(n)$. ODCs are fundamental to redundancy analysis in technology-independent synthesis [1, 2, 4, 7, 9–11]. The care set can be further divided into care-on and care-off patterns according to the value of n:

$$\begin{aligned} S_{\text{care-on}}(n) &= \{p \in S_{\text{care}}(n) \mid n(p) = 1\}, \\ S_{\text{care-off}}(n) &= \{p \in S_{\text{care}}(n) \mid n(p) = 0\}. \end{aligned} \tag{1}$$

2.3 Redundancy Addition and Removal

RAR is a high-effort restructuring framework widely used in AIG optimization. Recent *(GPU-)transduction* and *rewiring* methods are representative RAR-based techniques [2, 7, 9, 10]. RAR consists of two conceptual steps: adding function-preserving redundancies and removing logic that becomes redundant after the structural perturbation.

Consider an AND node f and a candidate signal g to be added as a new fanin of f. Let $S_{\text{care}}(f)$ be the care set of f, and let $f(p)$ and $g(p)$ denote the values of f and g, respectively, under input pattern p. Adding g as a new fanin changes the node function from f to $f' = f \wedge g$. To preserve the observable behavior of the network, this addition must not change f on any care pattern where $f = 1$. Thus, the redundancy addition is legal only if

$$\forall p \in S_{\text{care-on}}(f), g(p) = 1. \tag{2}$$

Accordingly, the added wire may change the internal value of f only on patterns where f is unobservable or where $f = 0$.

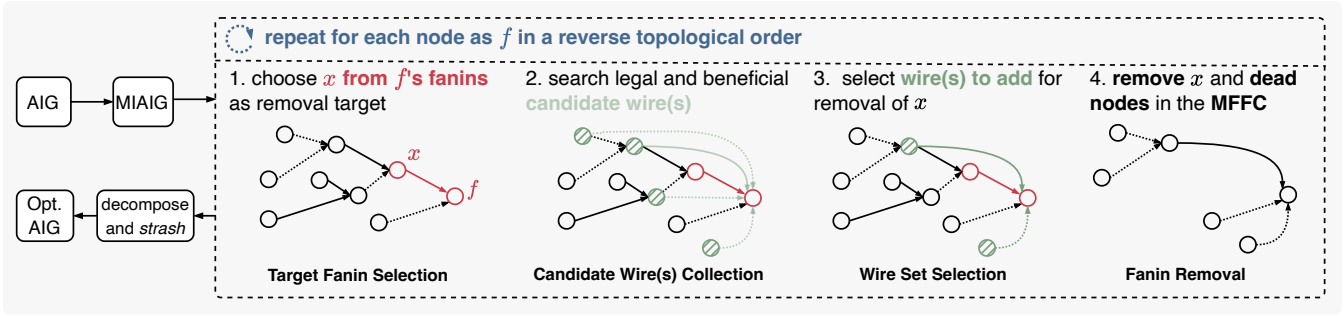

Figure 3: Overall workflow.

After redundancy addition, RAR applies various reduction procedures to identify newly redundant edges and nodes in the network. For example, an existing fanin may become removable if the added wire makes its contribution unnecessary on all observable patterns. The modified network is committed only when the subsequent reduction improves the optimization objective; otherwise, the trial is discarded.

3 Methodology

Our method performs node reduction through targeted ODC-based wire additions. It starts from an explicit reduction target and focuses on how added wires affect that target.

3.1 Overall Workflow

Figure 3 shows the overall workflow of our method. The input AIG is first converted into an MIAIG, which provides a flexible representation for wire additions. The method then traverses each node f in reverse topological order and collects its single-fanout fanins as potential removal targets. In Step 1, *Target Fanin Selection*, each single-fanout fanin x of f is selected as the removal target. The care set of f and the required care-off patterns associated with x are then derived to establish the node-reduction conditions, as described in Sections 3.2 and 3.3. In Step 2, *Candidate Wire(s) Collection*, the method collects candidate wires that preserve the observable behavior of f and cover at least one required pattern. In Step 3, *Wire Set Selection*, the method searches for a wire set W that covers all required patterns, first using the single-wire mode and then the multi-wire mode, as detailed in Section 3.4. In Step 4, *Fanin Removal and Cleanup*, the selected wires are added to f, the target fanin x is removed, and the resulting dead nodes in the MFFC of x are deleted. These steps are repeated for the remaining single-fanout fanins of f. After processing f, its transitive fanout cone (TFO) is re-simulated to maintain valid simulation data for subsequent nodes. Once the traversal is finished, the optimized MIAIG is decomposed and structurally hashed into a standard AIG.

3.2 Care Set Calculation

For each node f, the method first computes its care set, which is shared by all target fanins of f. This care information is used by Step 2 in Figure 3 to collect legal and beneficial candidate wires. The calculation compares the original network with a modified network in which f is replaced by $\bar{f}$, producing a complete care set for node f.

Algorithm 1: Care Set Calculation for One Node

Data: node f, original PO functions $\{z_1, \ldots, z_q\}$, the transitive fanouts TFO of node f
Result: $S_{\text{care}}(f)$, $S_{\text{care-on}}(f)$, $S_{\text{care-off}}(f)$

$S_{\text{care}}(f), S_{\text{care-on}}(f), S_{\text{care-off}}(f) \leftarrow \emptyset$
for each input pattern p **do**
  simulate TFO after replacing f with $\bar{f}$
  calculate the modified PO values $\{z'_1(p), \ldots, z'_q(p)\}$
  if $\exists j \in \{1, \ldots, q\}$ such that $z_j(p) \neq z'_j(p)$ **then**
    $S_{\text{care}}(f) \leftarrow S_{\text{care}}(f) \cup \{p\}$
    if $f(p) = 1$ **then** $S_{\text{care-on}}(f) \leftarrow S_{\text{care-on}}(f) \cup \{p\}$
    else $S_{\text{care-off}}(f) \leftarrow S_{\text{care-off}}(f) \cup \{p\}$

Let $z_1, \ldots, z_q$ be the PO functions of the original AIG, and let $z'_1, \ldots, z'_q$ be the PO functions after replacing f with $\bar{f}$ in the network. Then, for an input pattern p, the change at f is observable if at least one PO value differs:

$$\Delta_f(p) = \bigvee_{j=1}^{q} \left(z_j(p) \oplus z'_j(p) \right). \tag{3}$$

Therefore, the care set of f is

$$S_{\text{care}}(f) = \{p \mid \Delta_f(p) = 1\}. \tag{4}$$

This definition produces the complete care set used in subsequent candidate-wire collection and selection. Algorithm 1 details the calculation of the care, care-on, and care-off sets for a node f. The method collects the nodes in the transitive fanout (TFO) of f in topological order and re-simulates their truth tables after replacing f with $\bar{f}$ to obtain $z'_1, \ldots, z'_q$. Comparing these functions with $z_1, \ldots, z_q$ yields the care, care-on, and care-off sets.

3.3 Node-Reduction Conditions

In this section, we formulate the conditions under which a set of added wires can make a target fanin x of f redundant.

Consider an MIAIG node f with fanins $x, x_1, \ldots, x_k$:

$$f = x \wedge \bigwedge_{i=1}^{k} x_i. \tag{5}$$

The method chooses the existing fanin x as the reduction target and attempts to add a wire set $W = \{w_1, \ldots, w_m\}$ that makes x redundant at f. After adding W and removing x, the function of f becomes

$$f' = \left(\bigwedge_{i=1}^{k} x_i\right) \wedge \left(\bigwedge_{w \in W} w\right). \tag{6}$$

The goal is to ensure that replacing f with f' preserves the observable behavior of the AIG.

The first requirement is to preserve the value of f on $S_{\text{care-on}}(f)$. For any pattern in $S_{\text{care-on}}(f)$, the value of every added wire must be 1:

$$\forall p \in S_{\text{care-on}}(f), \quad \bigwedge_{w \in W} w(p) = 1. \tag{7}$$

The second requirement handles the relevant care-off patterns for which x is the only fanin forcing f to 0. We define the required care-off pattern set of x as

$$S_{\text{r}}(x) = \left\{ p \in S_{\text{care-off}}(f) \mid x(p) = 0 \wedge \bigwedge_{i=1}^{k} x_i(p) = 1 \right\}. \tag{8}$$

For each pattern in $S_{\text{r}}(x)$, removing x without adding any compensating wire would change the value of f from 0 to 1. Therefore, the added wires must reproduce the missing 0-valued input of the AND node: on every pattern in $S_{\text{r}}(x)$, the value of at least one added wire must be 0, so that the value of f' remains 0:

$$\forall p \in S_{\text{r}}(x), \quad \bigwedge_{w \in W} w(p) = 0. \tag{9}$$

Together, the conditions in Equations (7) and (9) are sufficient to preserve the observable behavior of f. On the care-on set $S_{\text{care-on}}(f)$, Equation (7) keeps $f' = 1$. On the required care-off patterns $S_{\text{r}}(x)$, Equation (9) keeps $f' = 0$ by ensuring that the value of at least one added wire is 0. On the remaining care-off patterns, the value of some fanin other than x is already 0, so removing x cannot change the output of the AND node. On ODC patterns, changes at f are unobservable by definition. Therefore, a wire set satisfying these conditions can safely make x redundant at f.

3.4 Candidate Wire Selection

Algorithm 2 summarizes Steps 2 and 3 in Figure 3, which collect candidate wires and select a wire set for removing the target fanin x of f. The procedure searches for a small wire set W that preserves the care-on set $S_{\text{care-on}}(f)$ and covers the required care-off set $S_{\text{r}}(x)$.

In the candidate collection stage, the method first enumerates legal source nodes. A source is legal only if it is not f, not an existing fanin of f, not in the transitive fanout cone of f, and does not have a higher level than f in the MIAIG. The TFO condition prevents cycles, while the level condition reduces the risk of a depth increase caused by the added wires. For each remaining source, both polarities, s and $\bar{s}$, are considered candidate wires. Any candidate wire is function-preserving only if its value is 1 on every pattern in $S_{\text{care-on}}(f)$, as shown in Equation (7). Among these function-preserving wires, useful candidates are those that force f' to 0 on at least one required care-off pattern. Their coverage is defined as

$$\text{cov}(w) = \{p \in S_{\text{r}}(x) \mid w(p) = 0\}. \tag{10}$$

Only candidates with non-empty coverage are inserted into the candidate set C, whose size is capped by the predefined limit L. Before full coverage evaluation, the implementation prunes candidates with no coverage on an evenly spaced sample of approximately 25% of the active required words. When $S_{\text{care-on}}(f)$ is non-empty, the opposite polarity is pruned once one polarity passes the care-on preservation condition, because the two polarities cannot simultaneously satisfy this condition on a non-empty care-on set.

Algorithm 2: Candidate Wire Selection for Removing x

Data: Node f, target fanin x, care-on set $S_{\text{care-on}}(f)$, required care-off set $S_{\text{r}}(x)$, wire budget B, candidate limit L
Result: Selected wire set W or none

Ⓐ **Candidate collection**
$C \leftarrow \emptyset$
$\mathcal{S}_f \leftarrow \{s \mid s \notin \{f\} \cup \text{fanin}(f) \cup \text{TFO}(f),\ \text{level}(s) \le \text{level}(f)\}$
for each source node $s \in \mathcal{S}_f$ **do**
  if $L > 0$ and $|C| \ge L$ **then break**
  for each polarity $w \in \{s, \bar{s}\}$ **do**
    if $\forall p \in S_{\text{care-on}}(f),\ w(p) = 1$ **then**
      $\text{cov}(w) \leftarrow \{p \in S_{\text{r}}(x) \mid w(p) = 0\}$
      if $\text{cov}(w) \neq \emptyset$ **then**
        $C \leftarrow C \cup \{w\}$
      if $S_{\text{care-on}}(f) \neq \emptyset$ **then break**
Ⓑ **Single-wire mode**
for each $w \in C$ **do**
  if $\text{cov}(w) = S_{\text{r}}(x)$ **then return** $\{w\}$
Ⓒ **Multi-wire mode**
$W \leftarrow \emptyset$
$U \leftarrow S_{\text{r}}(x)$ // uncovered required patterns
while $U \neq \emptyset$ and $|W| < B$ **do**
  $w^* \leftarrow \arg\max_{w \in C,\ \text{src}(w) \notin \text{src}(W)} |\text{cov}(w) \cap U|$
  if $\text{cov}(w^*) \cap U = \emptyset$ **then return** none
  $W \leftarrow W \cup \{w^*\}$
  $U \leftarrow U \setminus \text{cov}(w^*)$
if $U = \emptyset$ and $\text{gain}(W, x) \ge 0$ **then return** W
return none

Before returning a wire set W, the method calculates its gain to the MIAIG as

$$\text{gain}(W, x) = 1 + |\text{MFFC}(x) \setminus \text{Preserve}(W, x)| - |W|, \tag{11}$$

where $\text{Preserve}(W, x)$ denotes the subset of $\text{MFFC}(x)$ that remains reachable from the source nodes of the wires in W, including the selected source nodes and their required transitive fanin cones. The gain counts the removed fanin and deletable MFFC nodes, and subtracts the number of added wires.

The method then tries the single-wire mode. If a candidate wire covers all patterns in $S_{\text{r}}(x)$, the single-wire set $W = \{w\}$ satisfies Equation (9) and the resulting gain is non-negative according to Equation (11). If no single wire covers $S_{\text{r}}(x)$, the method enters the multi-wire mode. It maintains the uncovered set U, initialized as $S_{\text{r}}(x)$, and greedily selects the candidate that covers the largest number of currently uncovered patterns. After a wire is selected, its covered patterns are removed from U. The process stops when all required patterns are covered or when the wire budget B is exhausted. Finally, the selected set W is committed only if its gain is non-negative.

Table 1: Comparison of `gtr` [7], `rewire` [2], and ours on the IWLS 2022, 2023, and 2024 benchmark suites.

Bench. Suite	Single Iteration									Iterations Until Convergence								
	Mean Size Red. (%)			Mean Depth Red. (%)			Mean Runtime (s)			Mean Size Red. (%)			Mean Depth Red. (%)			Mean Runtime (s)		
	gtr	rewire	ours	gtr	rewire	ours	gtr	rewire	ours	gtr	rewire	ours	gtr	rewire	ours	gtr	rewire	ours
`strash`†																		
IWLS'22	32.7	26.5	**36.1**	-32.3	**4.7**	2.0	83.2	1.4	**1.1**	43.3	33.0	**45.2**	-351.2	2.3	**2.4**	4003.1	11.8	**4.3**
IWLS'23	36.7	30.2	**41.1**	-48.1	-7.0	**-0.6**	52.4	1.6	**1.1**	48.3	40.1	**49.1**	-384.0	-15.8	**1.1**	2877.7	16.1	**3.6**
IWLS'24	34.8	29.4	**39.9**	-41.6	-5.1	**0.2**	49.6	1.5	**0.9**	46.0	39.4	**48.3**	-383.3	-16.7	**0.7**	3046.3	16.3	**3.0**
`strash` + `&deepsyn`‡																		
IWLS'22	2.7	0.8	**4.3**	-34.9	-30.5	**-16.6**	38.7	0.6	**0.4**	5.3	1.1	**6.1**	-266.9	-27.8	**-18.8**	1002.0	**1.4**	2.2
IWLS'23	2.9	0.9	**4.4**	-35.4	-27.7	**-17.3**	21.7	0.5	**0.3**	6.1	1.3	**6.8**	-323.3	-24.8	**-20.9**	670.6	2.4	**1.8**
IWLS'24	2.4	0.5	**4.4**	-36.2	-26.8	**-17.0**	17.1	0.4	**0.2**	5.5	0.9	**6.7**	-308.5	-23.9	**-19.6**	850.6	**1.4**	**1.4**

† Benchmarks are converted to AIGs using ABC `strash`; the resulting AIGs serve as the starting points.
‡ For the second starting point, `&deepsyn` is applied iteratively to the `strash` AIGs until an iteration yields no further size reduction.

3.5 Elimination of Unnecessary Computation

Candidate-wire selection repeatedly performs word-level operations over packed truth tables to check care-on legality and required-pattern coverage. During the optimization of a fixed target fanin x, no intermediate transformation is committed before a complete wire set is selected. Therefore, the truth tables, $S_{\text{care-on}}(f)$, and $S_{\text{r}}(x)$ remain unchanged throughout candidate collection and wire-set selection.

To exploit this stable data context, we introduce a sparse word representation (SWR) for packed truth tables. For $S_{\text{care-on}}(f)$ and $S_{\text{r}}(x)$, it records only the indices of non-empty words. The coverage $\text{cov}(w)$ of each retained candidate is computed once over the active required words and stored for subsequent wire-set selection. The single-wire mode directly checks whether the stored coverage contains all required patterns, whereas the multi-wire mode intersects it with the shrinking uncovered set for candidate scoring. Thus, the sparse word representation avoids both empty-word scans and repeated coverage computation.

4 Experimental Results

4.1 Setup

All experiments were conducted on an HPC cluster. Each job was allocated two CPU cores on an Intel Xeon Platinum 8358P processor running at 2.60 GHz, one NVIDIA A40 GPU with 48 GB of VRAM, and 48 GB of DRAM.

We evaluate our method on the IWLS 2022, 2023, and 2024 benchmark suites, where each suite contains 100 benchmarks. We use *GPU-transduction* (`gtr`) [7] and *rewiring* (`rewire`) [2] as high-effort logic optimization baselines. All methods follow the same benchmark-processing flow implemented in ABC[1]. In all experiments, we set the maximum wire budget to $B = 3$ and the candidate limit to $L = 2048$. For a fair comparison, each method is evaluated in two modes: a single iteration and repeated iterations until an iteration produces no further size reduction. We report AIG size reduction, depth reduction, and runtime. For each benchmark, size reduction and depth reduction are measured as the percentage decreases in the number of AIG nodes and the AIG depth, respectively, relative to the input network. A negative depth reduction indicates an increase in depth.

4.2 General Results

We first compare our method with `gtr` and `rewire` on AIGs directly generated by `strash`. As shown in the upper half of Table 1, our method achieves the highest single-iteration size reduction on all three benchmark suites. Its size reduction ranges from 36.1% to 41.1%, compared with 32.7%–36.7% for `gtr` and 26.5%–30.2% for `rewire`. Meanwhile, our method keeps the depth nearly unchanged and achieves the lowest runtime on every suite.

The same trend holds when each method is iterated until convergence. Our method reaches 45.2%–49.1% size reduction and consistently outperforms both baselines across the three suites. It also maintains positive depth reduction on all three suites, whereas `gtr` increases depth on every suite and `rewire` increases depth on two of the three suites. Moreover, our method converges in 3.0s–4.3s, compared with 11.8s–16.3s for `rewire` and over 2800s for `gtr`. These results demonstrate a consistent advantage in size reduction, depth preservation, and convergence efficiency from the same `strash` starting points.

4.3 Further Optimization Capability

To evaluate whether our method can still find useful reductions after strong conventional optimization, we further apply all methods to AIGs generated by `strash` followed by iterative `&deepsyn`. This setting is more challenging because `&deepsyn` has already removed many available reduction opportunities. As shown in the lower half of Table 1, our method still achieves the highest additional size reduction on all three benchmark suites. In a single iteration, it obtains 4.3%–4.4% additional reduction, compared with 2.4%–2.9% for `gtr` and 0.5%–0.9% for `rewire`. After convergence, our method further improves the reduction to 6.1%–6.8% and remains consistently better than both baselines.

Although all methods increase depth relative to these highly optimized starting points, our method limits the degradation to

[1] https://github.com/berkeley-abc/abc

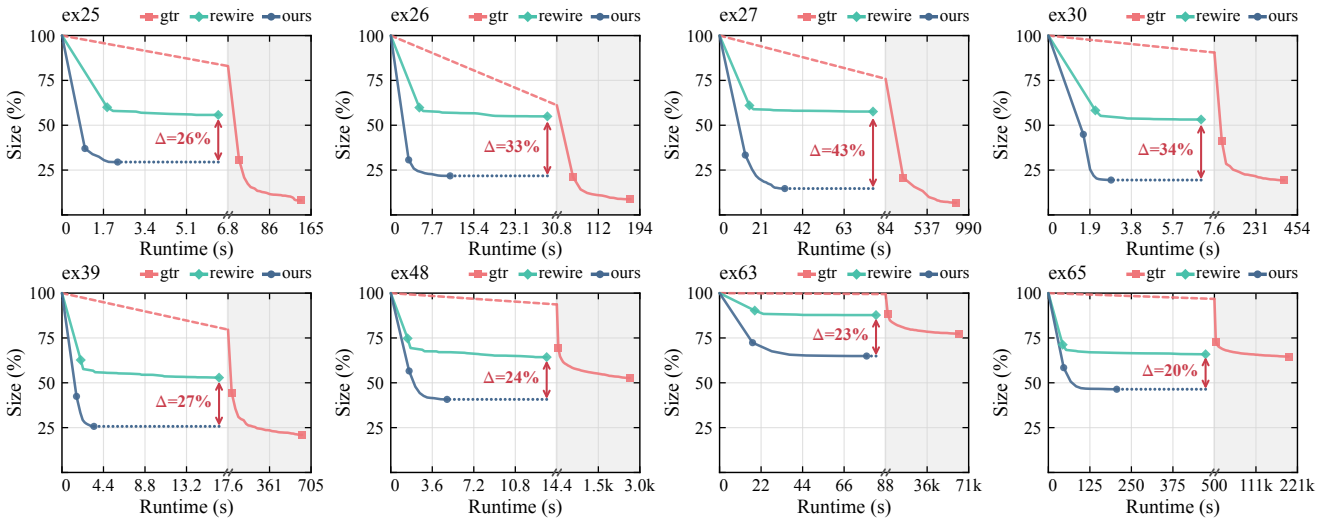

Figure 4: Size-runtime convergence curves of `gtr`, `rewire`, and ours on eight large benchmarks from the IWLS 2022 benchmark suite selected in the GPU-transduction work [7].

16.6%–17.3% in a single iteration and 18.8%–20.9% at convergence. These degradations are substantially smaller than those of `gtr` and `rewire`, especially after repeated optimization. Our method also achieves runtimes comparable to those of `rewire` and is substantially faster than `gtr`. These results demonstrate that the target-driven wire-addition space continues to expose useful reductions after `&deepsyn` while better controlling depth degradation.

4.4 Optimization Efficiency

Figure 4 compares the size-runtime convergence curves on eight large benchmarks from IWLS 2022 selected in the previous GPU-transduction work [7]. For each benchmark, all methods start from the same `strash` AIG and run iteratively until convergence. Across all eight benchmarks, our method reduces the AIG size rapidly in the first iteration and reaches convergence earlier than both baselines. Compared with `rewire`, it achieves a normalized final size that is 20%–43% lower while also requiring less runtime.

Although `gtr` can eventually reach a smaller final size on some benchmarks, it requires substantially more runtime to converge than ours. Our method instead obtains most of its reduction early, providing a better quality-runtime tradeoff before convergence. The suite-level results in Table 1 show the same efficiency trend beyond these eight benchmarks. These results support the use of a targeted node-reduction objective to reduce the speculative cost of high-effort wire addition.

4.5 Effectiveness of Sparse Word Representation

To evaluate the SWR technique introduced in Section 3.5, we compare the implementation with and without SWR on the three benchmark suites using `strash` starting points. All other settings are held constant, and both configurations are run iteratively until convergence. As shown in Table 2, the SWR achieves speedups of 9.3×, 3.4×, and 3.9× on IWLS 2022, 2023, 2024, respectively. These results confirm that eliminating unnecessary word-level operations substantially reduces the overall runtime.

Table 2: Runtime impact of the sparse word representation.

Bench. Suite	Mean Runtime (s)		Speedup
	w/o SWR	w/ SWR	
IWLS'22	39.8	4.3	9.3×
IWLS'23	12.1	3.6	3.4×
IWLS'24	11.8	3.0	3.9×

5 Conclusion

This paper extracts the effective action embedded in successful RAR trials and reorganizes the speculative workflow into a targeted transformation for AIG optimization. Rather than applying legality-driven wire additions before their reduction effects are known, our method first selects an explicit reduction target and jointly determines the wire additions and target removal before updating the AIG. We derive the key conditions for this optimization and develop a practical implementation that improves runtime efficiency. Experiments on the IWLS 2022, 2023, and 2024 benchmark suites demonstrate better size reduction, depth preservation, and convergence efficiency than *GPU-transduction* and *rewiring*. Future work will explore suitable windowing methods and parallel acceleration to extend our method to extremely large circuits with varying numbers of PIs.

Acknowledgments

This work was partially supported by Guangdong Provincial Project 2024QN11X113 and GDIC.

References

[1] Alessandro Tempia Calvino and Giovanni De Micheli. 2024. Scalable Logic Rewriting Using Don't Cares. In *2024 Design, Automation & Test in Europe Conference Exhibition (DATE)*. 1–6. doi:10.23919/DATE58400.2024.10546807

[2] Jiun-Hao Chen, Jie-Hong R. Jiang, and Alan Mishchenko. 2025. Versatile Rewiring and Concurrent Resynthesis for High-Quality Customized Optimization. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9. doi:10.1109/ICCAD66269.2025.11240927

[3] L.A. Entrena and Kwang-Ting Cheng. 1995. Combinational and sequential logic optimization by redundancy addition and removal. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14, 7 (1995), 909–916. doi:10.1109/43.391740

[4] Rongliang Fu, Libo Shen, Ziyi Wang, Zhengxing Lei, Zixiao Wang, Junying Huang, Bei Yu, and Tsung-Yi Ho. 2026. DCLOG: Don't Cares-based Logic Optimization using Pre-training Graph Neural Networks. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 793–799. doi:10.1109/ASP-DAC66049.2026.11420303

[5] Jitka Kocnova and Zdenek Vasicek. 2019. EA-Based Refactoring of Mapped Logic Circuits. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5. doi:10.1109/ISCAS.2019.8702084

[6] Shiju Lin, Jinwei Liu, Tianji Liu, Martin D. F. Wong, and Evangeline F. Y. Young. 2022. NovelRewrite: node-level parallel AIG rewriting. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 427–432. doi:10.1145/3489517.3530462

[7] Zhuofan Lin and Shiju Lin. 2026. GPU-Accelerated Efficient Transduction for Logic Optimization. In *2026 Design, Automation & Test in Europe Conference (DATE)*. 1–7. doi:10.23919/DATE69613.2026.11539334

[8] A. Mishchenko, S. Chatterjee, and R. Brayton. 2006. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In *2006 43rd ACM/IEEE Design Automation Conference*. 532–535. doi:10.1145/1146909.1147048

[9] Yukio Miyasaka. 2024. Transduction Method for AIG Minimization. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 398–403. doi:10.1109/ASP-DAC58780.2024.10473816

[10] Yukio Miyasaka, Alan Mishchenko, John Wawrzynek, Dino Ruić, and Xiaoqing Xu. 2025. High-Effort Logic Synthesis Using Randomized Transduction. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*. New York, NY, USA, 58–64.

[11] Yang Sun, Tianji Liu, Martin D.F. Wong, and Evangeline F.Y. Young. 2026. Scalable Parallel Logic Resubstitution with Advanced Don't Care Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2026), 1–1. doi:10.1109/TCAD.2026.3671670