Randomized Transduction for High-Effort Logic Synthesis

Yukio Miyasaka[1, 2], Alan Mishchenko[1], John Wawrzynek[1], Dino Ruić[2], Xiaoqing Xu[2]
[1]*UC Berkeley*
[2]*X, the moonshot factory*

***Abstract*—High-effort logic synthesis has become an important research direction due to the increase in silicon cost and the growth of design complexity. The emphasis on security leads to complex cryptographic circuits, while the acceleration of AI/ML results in custom arithmetic blocks—all of which need to be highly optimized by EDA tools. In such applications, high-effort logic synthesis allows for an efficient exploration of larger solution spaces, leading to area and power savings beyond the capacity of traditional methods. This paper presents a novel variation of high-effort logic synthesis called transduction, which performs transformation and reduction using don't-cares to restructure the circuit. Integrating the proposed method into a stochastic optimization flow with dynamic scheduling saved 6.8% AIG nodes on average, compared to the original flow using the same runtime. An additional experiment further demonstrated the strength of the proposed method, which discovered a new lower bound for 46 out of 100 benchmarks.**

I. Introduction

In the long history of EDA (electrical design automation), scalability has always been the primary concern to keep up with exponential transistor scaling. In particular, hardware developers expect EDA tools to run extremely fast for efficient design space exploration, even though most EDA problems are NP-hard. But the demands on EDA are evolving because, in recent years, many custom processors and programmable accelerators have been developed, especially for accelerating AI/ML applications. Custom processors and accelerators are designed to exploit the parallelism existing in applications. Typically their architecture comprises an array of small identical processing units equipped with complex arithmetic blocks. Because the architecture itself is quite simple, more time can be invested into the low-level implementation to further improve the performance. In contrast to the traditional use-case, EDA tools are now asked to generate a better implementation at the cost of a higher compute resource usage.

Logic synthesis is one of the most important steps in the EDA flow. It is performed in the early stage of the flow and converts an RTL (register transfer logic) into a netlist. Logic synthesis can have a large impact on the final quality of the design as the quality of downstream optimizations is highly dependent on the generated netlist. Logic synthesis is usually divided into two phases: technology-independent synthesis and technology mapping. The former optimizes the design in the form of a simple logic representation such as an AIG (and-inverter graph), XAIG (xor-and-inverter graph), MIG (majority-inverter graph), etc., while the latter converts those representations into a netlist composed of components in the technology library. The advantage of using simple logic representations during synthesis is that they can be handled efficiently without having exceptions for complex gates, and the algorithms can be shared across different technologies. It is also known that there exists a high correlation between the quality of designs before and after technology mapping. In this paper, we focus on technology-independent synthesis on AIGs.

Due to the recent shift in design trends, logic synthesis has also been directed towards high-effort optimization. One concept that is drawing attention these days, is that of *area-increasing transformations*. In technology-independent synthesis, *area* means the number of nodes in the representation, since it highly correlates with area after mapping. Conventionally, only transformations that monotonically decrease the area are used in logic synthesis. After generating an initial circuit based on SOP (sum of product), BDD (binary decision diagram), or another decomposition method, local transformations that never increase the area are repeated until convergence. However, since those local transformations are biased by the circuit structure, they often get stuck at a local minimum. Area-increasing transformations, on the other hand, allow the circuit to grow temporarily so that a larger design space can be explored to find a better local minimum.

In this paper, we propose a new method, *randomized transduction*, which performs an area-increasing transformation using don't-cares in a randomized manner. It was integrated into an existing state-of-the-art flow with dynamic scheduling so that it can efficiently explore the design space. On the IWLS 2022 benchmark suite [1], which consists of truth tables that can be synthesized into AIGs ranging from 10 to 6,000 nodes, the proposed flow outperformed the baseline by an average of 6.8% in terms of AIG node count, given the same time limit. Moreover, it was able to generate smaller AIGs than the previous best results, with or without using them, for 46 out of 100 functions in the benchmark suite.

Our contributions are summarized as follows:

- A novel way of using transduction is demonstrated, where it is used to randomly restructure the circuit and make new optimization opportunities for other algorithms.
- Dynamic scheduling is performed to efficiently explore the design space using different types of area-increasing transformations.

The remainder of this paper is organized as follows. Section II gives some background on transduction and other related

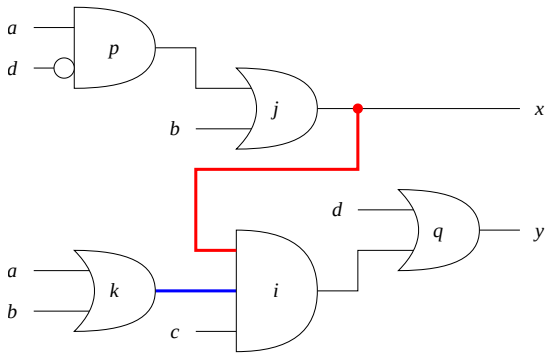


Fig. 1. Example of transduction. By adding the red wire from j to i, the blue wire from k to i becomes redundant.

work. Section III explains our proposed method. Section IV shows the experimental results compared with the state-of-the-art flow. Section V concludes the paper and lists possible future extensions.

II. Background

A. Transduction

Transduction is a coined word meaning a combination of transformation and reduction [2]. The transformation increases the area by adding redundant wires and extra nodes to the circuit. Those wires are redundant in the sense that they do not change the function of primary outputs. In circumstances where a redundant wire can be added, the underlying circuit satisfies the *connectable* condition, which can be calculated via *observability don't-cares* [3]. A signal j is connectable to an AND gate i if $f_j \vee \neg f_i \vee g_i \equiv 1$, where f_j and f_i denote the functions of the signal j and the output of the gate i, respectively, and g_i denotes the don't-care of the output of the gate i (g_i takes the value one when i is don't-care). If the condition is satisfied, a wire can be added from the signal j to the input of the gate i without affecting the primary output functions. This process of adding redundant wires/nodes is often referred to as *redundancy addition*.

Reduction, on the other hand, removes redundant wires from the circuit. If a wire takes the value one for all patterns other than its don't-cares, we can replace the wire with a constant one without changing the primary output functions. Let k be an input signal of the gate i, f_k be its function, and $g_{k,i}$ be the don't-care of the wire between k and i. The wire between k and i can be replaced with a constant one when $f_k \vee g_{k,i} \equiv 1$. Additionally, nodes that end up having no fanouts are also removed. This algorithm is also known as *redundancy removal*.

Example: In Fig. 1, before adding the wire from j to i, the functions in the circuit can be expressed by the truth tables shown below in terms of primary inputs $\{a, b, c, d\}$, where f_x denotes the function at the output of node x.

$$
\begin{aligned}
f_a &= && 1111\_1111\_0000\_0000\\
f_b &= && 1111\_0000\_1111\_0000\\
f_c &= && 1100\_1100\_1100\_1100\\
f_d &= && 1010\_1010\_1010\_1010\\
f_p &= f_a \wedge \neg f_d = && 0101\_0101\_0000\_0000\\
f_j &= f_p \vee f_b = && 1111\_0101\_1111\_0000\\
f_k &= f_a \vee f_b = && 1111\_1111\_1111\_0000\\
f_i &= f_k \wedge f_c = && 1100\_1100\_1100\_0000\\
f_q &= f_d \vee f_i = && 1110\_1110\_1110\_1010
\end{aligned}
$$

Because the other input of OR gate q is d, node i receives a don't-care when d takes the value one, denoted by g_i below. Here, the connectable condition is satisfied between j and i. In fact, the function of the output of i is modified to f_i' below by adding a wire from j to i, but the function at the output of q is unaffected. Meanwhile, the wire between k and i gains the don't-care denoted by $g'_{k,i}$ below, which makes $f_k \vee g'_{k,i} \equiv 1$ meaning the wire from k to i is replaceable with constant one.

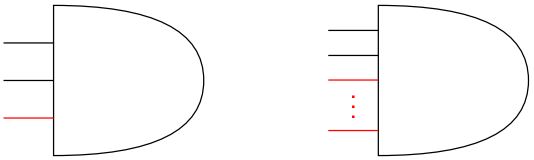
$$
\begin{aligned}
g_i &= f_d = && 1010\_1010\_1010\_1010\\
f_i' &= f_j \wedge f_k \wedge f_c = && 1100\_0100\_1100\_0000\\
g'_{k,i} &= \neg f_j \vee \neg f_c = && 0011\_1011\_0011\_1111
\end{aligned}
$$

Transduction iterates transformation and reduction to optimize the circuit. Transformation itself may increase the number of wires and nodes in the circuit, but it changes the distribution of don't-cares, and thus the subsequent reduction may be able to remove more wires/nodes than previously added.

Transduction was later rediscovered as *redundancy addition and removal* [4] and extended to *rewiring* [5] based on ATPG (automatic test pattern generation) techniques, where rewiring focused on removal of a particular wire by adding alternative wire(s).

B. Area-Increasing Transformation

Recently, other types of area-increasing transformations have been proposed. LUT mapping/unmapping [6] performs mapping to LUTs (look-up tables) and converts them back to the original representation. The circuit is dramatically restructured during this process, where some nodes are duplicated and packed into different LUTs, and resubstitution may be performed while changing the function of each LUT [7]. Although it largely increases the circuit size, by applying several local transformations, it is possible to recover and even reduce the circuit size. It is implemented as the command `&deepsyn` in ABC, a state-of-the-art logic synthesis tool [8]. The command `&deepsyn` iteratively performs LUT mapping with randomized configurations followed by a conventional script of local transformations: `compress2rs` or `&dc2`, both of which consist of a sequence of `rewrite`, `refactor` [9], `resub` [10], and `balance` [11]. A recent report [12] showed that a similar approach achieved 7% improvement on the MIG node count over the best results at the time.



(a) Single-wire addition (b) Multi-wire addition

Fig. 2. Redundancy addition by adding redundant wire(s) highlighted in red.

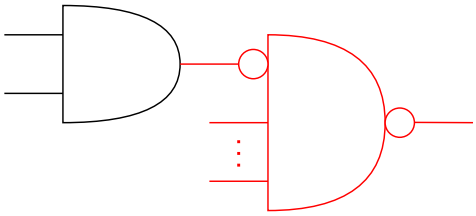


Fig. 3. Redundancy addition with inserting a complemented node highlighted in red.

In this context, LMS (lazy man's synthesis) [13] may be interpreted as an area-increasing transformation. LMS is known as the strongest delay (logic depth) optimization algorithm and replaces each cut with a functionally equivalent subcircuit in a precomputed library as in *rewriting* [9]. The main difference to rewriting is that it creates a library by recording subcircuits observed during the optimization of some benchmark circuits. Because recording is computationally cheaper than topological enumeration, it can use larger cut sizes (number of inputs per cut) while capturing the cumulative results of other algorithms. Although it was not directly useful for area optimization, it demonstrated significant delay improvement by creating a library in favor of logic depth. From another perspective, it trades area for delay, where each cut may be replaced with a larger subcircuit of smaller delay.

The final method, called *reshuffling* [14], increases the area by adding a redundant node that has the same function (up to a complement) as one of its fanins. Reshuffling iterates this area-increasing transformation with don't-care-based *0-resubstitution* [10] (node merging), which reportedly leads to a 1% area reduction after mapping.

III. Proposed Method

We propose a method for high-effort area optimization of AIGs. Extensions for different optimization criteria and other representations will be discussed in Section V.

A. Redundancy Addition

There are various ways to add redundant wires/nodes to the circuit. We adopted single-wire addition and multi-wire addition, with or without inserting a complemented node.

First of all, to better exploit the don't-cares in the circuit, we use an MIAIG (multi-input-AIG) as an intermediate representation. An MIAIG consists of multi-input AND gates along with inverters. We can convert an MIAIG into an AIG simply by decomposing each N-input MIAIG node into cascaded $N-1$ AIG nodes. The benefit of using MIAIG is that it can reduce the structural bias in don't-cares. Our method is based on compatible don't-cares [3], which is a subset of all don't-cares where at least one of the controlling fanins will remain unchanged for each controlling pattern. In MIAIGs, we can efficiently shuffle those don't-cares by permuting the fanins of multi-input nodes.

We perform redundancy addition by increasing the number of inputs of the target node with extra fanin(s) as shown in Fig. 2. We traverse the circuit except the transitive fanout (TFO) of the target node to find connectable signal(s). If the function of the node (or its negation) satisfies the connectable condition with respect to the target node, a wire is added from the node (respectively, its negation) to the target node.

For single-wire additions, we perform redundancy removal every time a new wire is added. If nothing else can be removed by redundancy removal (i.e., only the newly added wire is redundant), the changes are canceled by removing the added wire. Otherwise, we keep the changes, as some circuit restructuring has occurred. On the other hand, for multi-wire addition, redundancy removal is performed only once after all connectable signals are added to the input of the target node. In this case, the changes are saved if the number of removed wires is not less than the number of added wires, and otherwise they are discarded. We discard the changes when the number of wires has increased because keeping them was experimentally about 2-4% worse on average in our flow.

Additionally, to increase the generality, we also try redundancy addition by inserting a complemented node as shown in Fig. 3. We first insert a node with both input and output complemented at the output of the target node such that all previous fanouts of the target node become the fanouts of the inserted node with proper negations. Then, redundant wires are added to the input of the inserted node. This redundancy addition corresponds to the insertion of an OR gate in [15], although ours is less exhaustive as they consider the insertion for each fanout.

B. Randomized Transduction

Traditionally, transduction or rewiring has been used in an isolated manner, where the algorithms iterate over all existing nodes in the circuit. This reflects a long-standing idea in logic synthesis that optimization should end as soon as possible while the gain is likely to converge after one round. However, this approach suffers from the phase ordering problem: once one transformation has happened, some other transformations are no longer available. In contrary, our proposed algorithm is based on a new idea that regards logic synthesis as a search problem—the longer it runs, the better the result becomes.

Randomized transduction is an algorithm intended to be used in a stochastic flow. It performs redundancy addition and removal as described above with a target node selected in a random order, and it terminates when it finds a different structure that contains a smaller or equal number of nodes. Algorithm 1 shows an example using single-wire addition

Algorithm 1 Randomized transduction using single-wire addition without inserting a complemented node

Input: L: list of nodes; I: list of primary inputs
Output: List of nodes

for $i \in L$ in a random order **do**
  for $j \in (I \cup L \setminus \{i\}) \setminus TFO(i)$ in a random order **do**
    if j is connectable to i **then**
      $L' :=$ Copy of L
      Single-wire addition from j to i in L'
      Redundancy removal in L'
      Decompose all multi-input nodes in L'
      if L'.length $\leq$ L.length and $L \neq L'$ **then**
        return L'
      end if
    end if
  end for
end for
return L

without inserting a complemented node. In the unlikely case that it cannot find such a structure, it returns the original one.

Our actual implementation is a little more complex. For each node in the inner loop, we also add its negation to the input of the target node if that is connectable. Additionally, for each target node, we try wire addition both with and without inserting a complemented node, while their order is randomized as well.

C. Dynamic Scheduling

We integrated randomized transduction into the state-of-the-art high-effort AIG-based synthesis command `&deepsyn` in ABC. The problem we face is that we have different kinds of area-increasing transformations which need to be scheduled. One observation is that randomized transduction restructures the circuit more locally than LUT mapping/unmapping. It is known to be efficient in optimization to start from global search (LUT mapping/unmapping) and then gradually shift to local search (transduction), but it would be detrimental to phase out the global search due to the strong non-linearity of the solution space.

We adopted a dynamic scheduling approach that selects the transformation based on the results of previous iterations. Our stochastic optimization flow is shown in Algorithm 2. In the first few iterations, only LUT mapping/unmapping followed by conventional optimization is performed as long as the area decreases (because N remains 0). When the area stops decreasing, we start applying randomized transduction N times in the inner loop, where single-wire addition or multi-wire addition is selected randomly. The parameter N increases when no improvement was made in each iteration.

Our flow uses a random number generator to enable broader exploration. Transduction is randomized by shuffling the order of nodes in a list based on generated numbers. For the conventional optimization, either `compress2rs` or `&dc2` is selected based on a generated Boolean value. As in the original

Algorithm 2 Stochastic optimization flow with dynamic scheduling

Input: G: AIG; s: integer
Output: AIG

$R :=$ Random number generator with a seed s
$G' :=$ Copy of G
$N := 0$
repeat
  Apply LUT mapping/unmapping to G with R
  Apply conventional optimization to G with R
  repeat N **times**
    $f :=$ Random Boolean value from R
    if f **then**
      Apply single-wire transduction to G with R
    else
      Apply multi-wire transduction to G with R
    end if
    Apply conventional optimization to G with R
  end
  if G.size $<$ G'.size **then**
    $G' :=$ Copy of G
  else
    Increase N
  end if
until timeout
return G'

flow, LUT mapping/unmapping is randomized by toggling the option of *structural choice* command (`-f` of `&dch`) [16] and the use of *fast extract* (`&fx; &st`) [17], while the target LUT size is decided based on the value of some internal counter.

D. Implementation Details

We implemented transduction using our own BDD package as a back-end engine. We construct BDDs for functions and don't-cares in the circuit. It is our future work to have a comparison with simulation- or SAT-based engines.

Since the size of BDD is highly affected by the variable order, we perform variable reordering at the beginning of our flow. The variables are reordered after constructing BDDs for functions in the circuit, and the resulting variable order is saved and used throughout our flow.

Moreover, we reuse the BDD manager across different iterations, maintaining the unique table and the cache. Although we reconstruct BDDs every time transduction is called, it saves lots of time because most of the functions are found in the cache. This implementation has another advantage that transduction and other algorithms are completely decoupled, and there was no need to modify any existing code.

IV. Experimental Results

A. Benchmark

We conducted experiments on the IWLS 2022 benchmark suite [1]. It consists of 100 truth tables of completely specified Boolean functions whose input counts range from 5 to 16 and

TABLE I
IWLS 2022 BENCHMARK SUITE

Benchmarks	Description
ex00 - ex01	known random-looking functions
ex02 - ex07	modified or unmodified random functions
ex08 - ex09	s-box and inverse s-box from AES
ex10 - ex15	5- to 15-input majority functions
ex16 - ex27	binary sorters
ex28 - ex49	Espresso benchmarks
ex50 - ex67	custom arithmetic functions
ex68 - ex99	quantized neuron functions

output counts range from 1 to 77. Table I shows the breakdown of the benchmarks. It includes a variety of functions: random-looking decomposable functions, arithmetic functions, cryptographic functions [18], Espresso benchmarks [19], and quantized neuron functions [20]. We synthesized initial AIGs using "`strash`", "`collapse; strash`", "`collapse; sop; fx; strash`", and "`&ttopt`". Command `strash` synthesizes an AIG through algebraic factoring [21]. Command `collapse` constructs a BDD of the function. Command `sop` creates an irredundant SOP from the BDD [22][23]. Command `fx` synthesizes a multi-level circuit from the SOP through fast extract [17]. Command `&ttopt` synthesizes an AIG from a BDD but with more intense variable reordering.

B. Comparison with the State-of-the-Art Flow

We compared the proposed flow with the original state-of-the-art flow (`&deepsyn`). Both flows were designed to continue running until the time limit is reached (by removing the iteration limit). We ran each flow 48 times with different random seeds and calculated the average and minimum AIG sizes for each case. This is because we are comparing stochastic flows, and the results may vary depending on the value of the seed. We set the time limit for each run to 10 minutes. We used the smallest one among the four initial AIGs as a starting point.

Table II shows the results of the comparison. The third column for each track (average and minimum) shows the ratio of the proposed flow over `&deepsyn`, where blue cells show improvement. Overall, our flow generated 6.8% smaller AIGs in geometric mean of the averages. When we compare the minimum AIGs after 48 runs, our flow was 5.9% better in geometric mean. Considering specific test cases, our average and minimum were always smaller or the same, except for ex57 and ex72. To ensure that the improvement arose from transduction not from dependency of the target LUT size on the internal counter, we conducted another experiment randomizing the target LUT size. With this, the result of `&deepsyn` improved only by 0.1% for both average and minimum track, so the improvement is not merely due to the randomized target LUT size.

C. Comparison with the Best Known Results

We did an additional experiment by running the proposed flow for an hour from each starting point. In addition to the four initial AIGs above (labeled as "strash", "collapse",

TABLE II
COMPARISON OF &DEEPSYN AND THE PROPOSED FLOW ON THE AVERAGE AND MINIMUM AIG SIZES OF 48 RUNS WITH A 10-MINUTE TIME LIMIT

		Average			Minimum		
Benchmark	Initial	&deepsyn	Proposed	Ratio	&deepsyn	Proposed	Ratio
ex00	33	25.3	23.0	0.909	25	22	0.880
ex01	38	32.3	25.0	0.774	32	23	0.719
ex02	128	90.7	85.0	0.937	82	77	0.939
ex03	36	24.8	24.1	0.970	24	24	1.000
ex04	430	348.1	341.7	0.981	332	321	0.967
ex05	70	41.9	38.2	0.911	41	37	0.902
ex06	1437	1227.6	1216.2	0.991	1156	1155	0.999
ex07	243	154.3	141.5	0.917	147	134	0.912
ex08	765	647.3	635.4	0.981	610	601	0.985
ex09	763	650.0	632.0	0.972	626	612	0.978
ex10	12	10.0	10.0	1.000	10	10	1.000
ex11	24	23.3	20.0	0.860	20	20	1.000
ex12	40	32.5	30.3	0.933	30	30	1.000
ex13	60	48.0	45.9	0.957	42	42	1.000
ex14	84	67.2	62.3	0.926	60	58	0.967
ex15	112	90.8	84.1	0.927	81	78	0.963
ex16	20	18.0	18.0	1.000	18	18	1.000
ex17	30	24.0	24.0	1.000	24	24	1.000
ex18	42	36.1	32.0	0.887	32	32	1.000
ex19	56	45.7	41.7	0.911	42	38	0.905
ex20	72	60.0	53.7	0.894	54	50	0.926
ex21	90	75.0	66.7	0.888	70	62	0.886
ex22	110	91.8	82.7	0.901	84	76	0.905
ex23	132	114.6	101.0	0.881	101	94	0.931
ex24	156	136.1	120.6	0.886	125	114	0.912
ex25	182	160.7	142.5	0.887	142	132	0.930
ex26	210	188.6	166.8	0.885	172	154	0.895
ex27	240	216.2	193.1	0.893	200	179	0.895
ex28	80	39.0	39.0	1.000	39	39	1.000
ex29	49	40.6	37.9	0.932	37	35	0.946
ex30	759	69.1	68.0	0.984	68	68	1.000
ex31	1893	1557.2	1516.0	0.974	1500	1417	0.945
ex32	54	46.0	44.0	0.957	46	44	0.957
ex33	131	80.8	74.2	0.919	80	71	0.888
ex34	128	47.4	46.1	0.971	47	46	0.979
ex35	17	17.0	15.0	0.882	17	15	0.882
ex36	1860	1776.3	1734.4	0.976	1741	1676	0.963
ex37	214	147.9	140.9	0.953	146	137	0.938
ex38	53	31.3	27.0	0.862	29	27	0.931
ex39	675	225.0	193.6	0.860	213	177	0.831
ex40	404	206.5	185.8	0.900	193	183	0.948
ex41	25	17.0	17.0	1.000	17	17	1.000
ex42	58	28.0	28.0	1.000	28	28	1.000
ex43	90	37.0	37.0	1.000	37	37	1.000
ex44	115	70.8	50.4	0.713	55	48	0.873
ex45	386	217.2	189.1	0.871	201	184	0.915
ex46	54	32.3	31.0	0.961	32	31	0.969
ex47	25	25.0	25.0	1.000	25	25	1.000
ex48	680	509.5	491.6	0.965	498	474	0.952
ex49	76	39.0	39.0	1.000	39	39	1.000
ex50	21	18.0	18.0	1.000	18	18	1.000
ex51	51	31.6	27.2	0.862	28	26	0.929
ex52	26	19.1	18.1	0.949	19	18	0.947
ex53	61	42.8	35.6	0.832	38	34	0.895
ex54	14	13.0	12.0	0.923	13	12	0.923
ex55	245	152.4	142.5	0.935	144	128	0.889
ex56	35	29.0	29.0	1.000	29	29	1.000
ex57	272	191.5	202.0	1.055	142	162	1.141
ex58	145	90.9	81.3	0.895	84	75	0.893
ex59	412	267.2	261.7	0.979	246	242	0.984
ex60	100	67.5	58.7	0.869	61	54	0.885
ex61	2508	1915.8	1915.7	1.000	1860	1860	1.000
ex62	55	40.0	40.0	1.000	40	40	1.000
ex63	1500	1160.1	1121.5	0.967	1113	1070	0.961
ex64	599	423.5	411.2	0.971	397	395	0.995
ex65	2407	1852.4	1794.3	0.969	1787	1715	0.960
ex66	438	341.8	329.9	0.965	323	315	0.975
ex67	6477	6380.2	6312.9	0.989	6352	6220	0.979
ex68	392	250.7	229.0	0.913	239	206	0.862
ex69	439	278.4	254.1	0.913	247	217	0.879
ex70	448	236.1	197.9	0.838	210	171	0.814
ex71	582	349.4	317.8	0.909	323	289	0.895
ex72	802	418.7	433.4	1.035	348	364	1.046
ex73	534	252.9	201.6	0.797	180	154	0.856
ex74	939	552.2	519.9	0.941	477	467	0.979
ex75	822	554.6	514.9	0.928	490	480	0.980
ex76	347	227.4	215.3	0.947	215	202	0.940
ex77	436	319.7	295.8	0.925	296	274	0.926
ex78	458	358.7	340.8	0.950	345	323	0.936
ex79	575	368.4	353.2	0.959	332	332	1.000
ex80	663	523.9	500.1	0.955	502	483	0.962
ex81	485	355.8	340.5	0.957	341	327	0.959
ex82	858	593.6	566.1	0.954	565	542	0.959
ex83	792	633.1	593.3	0.937	606	566	0.934
ex84	194	129.8	122.7	0.946	124	115	0.927
ex85	261	199.7	193.0	0.967	187	184	0.984
ex86	257	171.2	160.8	0.939	164	146	0.890
ex87	491	364.3	354.2	0.972	355	341	0.961
ex88	397	311.9	294.6	0.944	301	278	0.924
ex89	283	202.8	194.0	0.957	195	179	0.918
ex90	614	464.7	449.8	0.968	455	437	0.960
ex91	307	232.1	224.8	0.968	222	207	0.932
ex92	37	31.4	28.9	0.922	30	28	0.933
ex93	59	46.5	39.3	0.844	42	39	0.929
ex94	60	42.5	33.5	0.788	40	33	0.825
ex95	85	66.0	60.3	0.913	63	57	0.905
ex96	98	76.1	69.0	0.907	74	66	0.892
ex97	87	66.6	60.1	0.903	65	57	0.877
ex98	179	138.7	132.7	0.957	132	127	0.962
ex99	119	86.8	79.1	0.911	84	74	0.881
Geomean	173.5	122.2	113.9	0.932	114.9	108.2	0.941

"sop-fx", and "&ttopt", respectively), we used the previous best result from the IWLS 2022 [1] and IWLS 2023 [24] competitions and our previous work [25] for each benchmark as another starting point (labeled as "prev."). The IWLS 2023 competition used the same set of benchmarks as the IWLS 2022 competition, while they were obfuscated under NPN equivalence. The previous best results were collected from the results of Team EPFL [26] at IWLS 2022, the results of TU Wien (TUW) [27] and Google DeepMind (GDM) [28] at IWLS 2023 as well as our previous results [25]. They all use ABC as an optimization engine, while EPFL generated initial circuits with various decomposition methods [29][30][31], TUW interleaved ABC with *exact synthesis* [32], and GDM used circuit neural networks to synthesize initial circuits.

Table III shows the AIG sizes of each starting point and the results of our method, which takes the minimum of 48 runs of the proposed flow with a one-hour time limit. The last column denotes where the previous best result (prev.) came from, or otherwise it is from our previous work. Cells after optimization are highlighted in blue when the resulting AIG is smaller than the previous best result. The second to last column (Corr.) shows the correlation of the AIG sizes before and after optimization excluding those starting from the previous best results, while a cell is highlighted in the darker red for the higher correlation.

Our method was able to produce smaller AIGs than the previous best results for 16 cases starting from some of the four initial AIGs generated independently of the previous best results. When starting from the previous best results, our method was able to reduce the AIG sizes in 43 cases, most of which started from the results of competition participants. This is remarkable because they have already run `&deepsyn` and other ABC commands for days or weeks, which indicates that having randomized transduction in the flow is crucial to find a better local minimum.

Another thing we can see in the results is that there is a high correlation in the AIG sizes before and after optimization for hard instances with over 100 AIG nodes. There are some exceptions such as sorter functions (ex16-ex27), but the correlation was notably high for random functions (ex02-ex07), cryptographic functions (ex08 and ex09), custom arithmetic functions (ex50-ex67), and quantized neuron functions (ex68-ex99). In order to produce comparable AIGs from scratch for those cases, it would be necessary to explore the circuit initialization algorithms, which is out of the scope of this paper.

V. Conclusion

We propose a method based on transduction to restructure AIGs in a randomized manner to create new optimization opportunities for conventional algorithms. Integrated in a state-of-the-art flow with dynamic scheduling, our proposed method produced 6.8% smaller AIGs on average using the same runtime. In the additional experiment, we discovered a new lower bound for 46 functions in the benchmark suite using the proposed method.

The proposed method can be extended to delay-optimization. One possible approach is to prioritize removal of wires in the critical paths as proposed on [33][34]. On the other hand, in many cases LMS [13] can generate sufficiently shallow AIGs. Our method may be used for area recovery after LMS. It is easy to modify the condition to keep or discard the changes after reduction in our method so that the logic depth is maintained. For efficiency, we may need to restrict redundancy addition in such a way that the wires are added only from nodes in a lower level with respect to the target node. Multi-input nodes should be decomposed into a balanced tree of nodes for this purpose.

Our method currently focuses on optimizing AIGs. For other circuit representations, redundancy addition is harder to define, while redundancy removal is still applicable based on observability don't-cares. Alternatively, SPFDs (sets of pairs of functions to be distinguished) [35] may be used to represent the logic flexibilities in a circuit. It is likely that SPFDs are more suitable than don't-cares for XAIGs, where any 2-input function takes only one node. Our future work will focus on the development of a more general redundancy addition/removal and SPFD-based optimization for XAIGs.

We used BDDs in our back-end engine. Although BDDs are not scalable, it worked fine for the selected benchmark suite with functions of up to 6k nodes in an optimized AIG. To scale up for larger circuits, some kind of partitioning would be necessary. Simulation, SAT solving, and combination of them [36] should be evaluated against BDDs for the best quality/runtime trade-off.

References

[1] *Problems and results of IWLS 2022 programming contest*. [Online]. Available: https://github.com/alanminko/iwls2022-ls-contest.

[2] S. Muroga *et al.*, "The transduction method-design of logic networks based on permissible functions," *IEEE TC*, vol. 38, no. 10, pp. 1404–1424, 1989.

[3] A. Mishchenko *et al.*, "SAT-based complete don't-care computation for network optimization," in *Proc. DATE*, 2005, pp. 412–417.

[4] K.-T. Cheng *et al.*, "Multi-level logic optimization by redundancy addition and removal," in *Proc. EDAC*, 1993, pp. 373–377.

[5] S.-C. Chang *et al.*, "Fast boolean optimization by rewiring," in *Proc. ICCAD*, 1996, pp. 262–269.

[6] N. Een, personal communication.

[7] A. Mishchenko *et al.*, "Scalable don't-care-based logic optimization and resynthesis," in *Proceedings of FPGA*, 2009, pp. 151–160.

[8] Berkeley Logic Synthesis and Verification Group, *ABC: A system for sequential synthesis and verification*. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/.

[9] A. Mishchenko *et al.*, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Proc. DAC*, 2006, pp. 532–535.

[10] A. Mishchenko *et al.*, "Scalable logic synthesis using a simple circuit structure," 2006, pp. 15–22.

[11] J. Cortadella, "Timing-driven logic bi-decomposition," *IEEE TCAD*, vol. 22, no. 6, pp. 675–685, 2003.

[12] S.-Y. Lee *et al.*, "Customizable on-the-fly design space exploration for logic optimization of emerging technologies," in *Proc. IWLS*, 2023, pp. 149–155.

[13] W. Yang *et al.*, "Lazy man's logic synthesis," in *Proc. ICCAD*, 2012, pp. 597–604.

[14] E. Testa *et al.*, "Extending rewiring: Reshuffling don't cares to unlock more optimization," in *Proc. IWLS*, 2022, pp. 34–40.

TABLE III

AIG SIZES BEFORE AND AFTER APPLYING THE PROPOSED FLOW WITH A ONE-HOUR TIME LIMIT AND TAKING THE MINIMUM OF 48 RUNS

	Starting point					After applying the proposed method						
Benchmark	strash	collapse	sop-fx	&ttopt	prev.	strash	collapse	sop-fx	&ttopt	prev.	Corr.	Source of prev.
ex00	57	33	34	37	21	22	22	22	22	21	-	
ex01	60	44	38	39	23	24	24	23	24	23	0.475	
ex02	219	149	138	128	69	72	76	74	73	66	-0.518	GDM'23
ex03	132	75	55	36	24	24	28	28	24	24	-0.264	
ex04	904	605	499	430	287	389	365	385	309	286	0.645	GDM'23
ex05	396	168	114	70	37	37	37	39	37	37	-0.336	
ex06	3389	2267	1643	1437	966	1780	1480	1454	1149	966	0.918	
ex07	1822	1327	887	243	112	508	131	126	128	106	0.751	GDM'23
ex08	1450	1097	866	765	513	682	665	639	591	513	0.891	
ex09	1443	1091	848	763	515	673	661	645	590	515	0.814	
ex10	24	12	12	12	10	10	10	10	10	10	-	
ex11	69	26	24	24	20	20	20	20	20	20	-	
ex12	143	44	40	40	30	30	30	30	30	30	-	
ex13	302	66	61	60	40	42	40	40	42	40	0.566	TUW'23
ex14	638	94	91	84	52	56	58	54	58	52	-0.177	GDM'23
ex15	1358	126	143	112	68	69	74	72	78	68	-0.764	TUW'23
ex16	70	32	22	20	18	18	18	18	18	18	-	
ex17	128	50	36	30	24	24	24	24	24	24	-	
ex18	204	70	49	42	32	32	32	32	32	32	-	
ex19	310	96	72	56	38	38	38	38	38	38	-	
ex20	464	124	99	72	46	50	50	50	50	46	-	GDM'23
ex21	668	158	132	90	56	62	62	58	62	56	0.318	
ex22	1006	194	164	110	63	74	74	72	74	63	0.320	GDM'23
ex23	1445	234	208	132	72	72	90	86	90	72	-0.977	GDM'23
ex24	2195	274	255	156	88	82	108	127	110	78	-0.881	
ex25	3160	322	238	182	90	92	128	106	128	90	-0.807	GDM'23
ex26	4861	372	285	210	99	95	156	141	148	97	-0.971	
ex27	7009	426	588	240	119	107	174	186	170	108	-0.973	
ex28	185	141	105	80	39	39	39	39	39	39	-	
ex29	185	71	57	49	35	35	35	35	35	35	-	
ex30	4099	1159	759	1103	68	68	68	68	68	68	-	
ex31	3664	2880	1997	1893	1205	1769	1683	1552	1382	1205	0.919	
ex32	153	62	54	87	44	44	44	44	44	44	-	
ex33	215	205	141	162	70	69	69	69	69	70	-	TUW'23
ex34	372	187	135	128	44	46	46	46	46	44	-	GDM'23
ex35	32	17	17	24	15	15	15	15	15	15	-	
ex36	3328	3190	1986	2081	1345	2008	1992	1672	1382	1297	0.893	EPFL'22
ex37	1056	482	214	365	138	135	135	135	135	133	-	TUW'23
ex38	90	72	53	57	27	27	27	27	27	27	-	
ex39	3898	1224	675	817	163	167	164	164	168	163	0.362	
ex40	1821	940	409	968	178	178	178	179	178	178	-0.714	
ex41	59	39	25	34	17	17	17	17	17	17	-	
ex42	143	116	62	66	28	28	28	28	28	28	-	
ex43	230	172	90	91	37	37	37	37	37	37	-	
ex44	332	172	115	129	49	48	48	48	48	48	-	
ex45	2135	894	470	906	179	182	182	182	181	179	0.181	TUW'23
ex46	70	55	54	55	31	31	31	31	31	31	-	
ex47	380	129	25	48	25	25	25	25	25	25	-	
ex48	3371	2135	866	1288	406	685	580	483	500	406	0.997	TUW'23
ex49	199	133	111	76	39	39	39	39	39	39	-	
ex50	67	35	23	21	18	18	18	18	18	18	-	
ex51	191	97	90	51	26	26	26	26	26	26	-	
ex52	38	28	26	28	18	18	18	18	18	18	-	
ex53	122	82	61	61	34	34	34	34	34	34	-	
ex54	31	14	14	18	12	12	12	12	12	12	-	
ex55	405	316	263	245	112	115	112	113	118	111	-0.218	
ex56	160	70	39	35	29	29	29	29	29	29	-	
ex57	916	1342	902	272	81	103	301	188	88	80	0.838	GDM'23
ex58	295	209	162	145	73	72	71	71	71	70	0.915	
ex59	1435	847	617	412	182	292	247	299	206	173	0.582	GDM'23
ex60	221	139	100	106	53	54	54	54	54	53	-	
ex61	5885	5024	3594	2508	1319	2882	2794	2991	1853	1310	0.703	GDM'23
ex62	273	88	55	66	40	40	40	40	40	40	-	
ex63	8618	17695	10504	1500	168	3821	11405	9976	1063	168	0.910	GDM'23
ex64	2050	1407	1016	599	317	738	696	604	377	302	0.903	GDM'23
ex65	15213	8989	5929	2407	1182	5761	3744	4277	1595	1182	0.904	GDM'23
ex66	1485	960	707	438	239	445	396	429	300	237	0.767	GDM'23
ex67	13502	10860	6949	7070	4911	7209	6655	6179	5470	4842	0.914	EPFL'22
ex68	1347	925	615	392	118	373	225	227	194	113	0.912	GDM'23
ex69	1176	865	538	439	106	292	146	207	174	105	0.621	GDM'23
ex70	1972	1491	655	448	92	640	158	103	94	91	0.839	GDM'23
ex71	1851	1272	932	582	137	636	517	581	253	133	0.801	GDM'23
ex72	2839	2507	1207	802	142	942	616	633	255	136	0.829	GDM'23
ex73	1532	966	676	534	89	108	156	137	105	87	-0.131	GDM'23
ex74	3888	2524	1805	939	194	1487	1253	1205	330	170	0.865	
ex75	3239	2421	1512	822	175	1044	1078	732	376	163	0.920	GDM'23
ex76	1529	808	580	347	167	348	260	229	189	156	0.997	GDM'23
ex77	1250	967	706	436	224	400	368	370	250	212	0.872	GDM'23
ex78	2047	1002	840	458	290	603	430	436	306	289	0.976	GDM'23
ex79	1847	1248	946	575	188	707	413	571	294	179	0.823	GDM'23
ex80	3184	2599	1711	663	435	1328	1367	1050	463	434	0.947	GDM'23
ex81	2233	1332	929	485	215	586	504	517	300	206	0.843	GDM'23
ex82	4208	2936	1965	858	523	1705	1619	1362	511	523	0.898	
ex83	4398	2768	1985	792	549	1818	1480	1459	535	549	0.904	
ex84	572	292	248	194	112	117	115	111	110	107	0.879	GDM'23
ex85	818	511	383	261	168	209	174	194	171	158	0.768	GDM'23
ex86	745	406	296	257	144	171	142	153	144	141	0.850	
ex87	1574	992	814	491	322	638	489	557	328	310	0.873	GDM'23
ex88	1445	949	697	397	261	435	363	409	268	254	0.801	GDM'23
ex89	1791	941	692	283	172	186	255	217	172	164	0.039	GDM'23
ex90	3316	1965	1390	614	413	1102	996	891	422	413	0.875	GDM'23
ex91	1486	838	553	307	200	361	244	243	205	198	0.969	GDM'23
ex92	128	51	37	39	29	28	28	28	28	28	-	
ex93	164	73	59	60	39	39	39	39	39	39	-	
ex94	239	100	71	60	33	33	33	33	33	33	-	
ex95	248	152	115	85	58	56	56	56	56	56	-	
ex96	388	240	177	98	67	66	66	66	66	66	-	
ex97	459	194	119	87	55	55	57	57	57	55	-0.964	
ex98	1014	599	342	179	128	132	126	128	125	125	0.844	GDM'23
ex99	380	228	155	119	76	75	73	74	74	74	0.537	
Ratio	7.120	3.591	2.582	2.026	1.000	1.425	1.366	1.343	1.121	0.984		

[15] S.-C. Chang *et al.*, "Perturb and simplify: Multilevel boolean network optimizer," *IEEE TCAD*, vol. 15, no. 12, pp. 1494–1504, 1996.

[16] S. Chatterjee *et al.*, "Reducing structural bias in technology mapping," *IEEE TCAD*, vol. 25, no. 12, pp. 2894–2903, 2006.

[17] J. Rajski *et al.*, "The testability-preserving concurrent decomposition and factorization of Boolean expressions," *IEEE TCAD*, vol. 11, no. 6, pp. 778–793, 1992.

[18] J. Daemen *et al.*, "The block cipher rijndael," *Lecture Notes in Computer Science - LNCS*, vol. 1820, pp. 277–284, 1998.

[19] *ESPRESSO benchmark*. [Online]. Available: https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/espresso/index.htm.

[20] Y. Umuroglu *et al.*, "Logicnets: Co-designed neural networks and circuits for extreme-throughput applications," in *Proc. FPL*, 2020, pp. 291–297.

[21] R. K. Brayton *et al.*, "The decomposition and factorization of Boolean expressions," in *Proc. ISCAS*, 1982, pp. 29–54.

[22] S.-i. Minato, "Fast generation of prime-irredundant covers from binary decision diagrams," in *Proc. SASIMI*, 1992, pp. 64–73.

[23] O. Coudert *et al.*, "Implicit prime cover computation: An overview," in *Proc. SASIMI*, 1993.

[24] *Problems and results of IWLS 2023 programming contest*. [Online]. Available: https://github.com/alanminko/iwls2023-ls-contest.

[25] Y. Miyasaka, "Transduction method for AIG minimization," in *Proc. ASP-DAC*, 2024, pp. 398–403.

[26] A. Costamagna *et al.*, "IWLS2022 contest - EPFL team," presented at the IWLS, 2022, [Online]. Available: https://youtu.be/qUnB-01oMiQ.

[27] F. Reichl *et al.*, "Circuit minimization with exact synthesis: From QBF back to SAT," in *Proc. IWLS*, 2023, pp. 98–105.

[28] A. Hillier *et al.*, "Learning to design efficient logic circuits," presented at the IWLS, 2023, [Online]. Available: https://drive.google.com/file/d/1hjZA3PalPRtjE_8z85MhJhl9GVhhheJt/view?usp=sharing.

[29] V. Bertacco *et al.*, "The disjunctive decomposition of logic functions," in *Proc. ICCAD*, 1997, pp. 78–82.

[30] J. Jacob *et al.*, "Unate decomposition of Boolean functions," in *Proc. IWLS*, 2001, pp. 66–71.

[31] A. Oliveira *et al.*, "Learning complex boolean functions: Algorithms and applications," in *Proc. NeurIPS*, 1993.

[32] W. Haaswijk *et al.*, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE TCAD*, vol. 39, no. 4, pp. 871–884, 2020.

[33] K.-C. Chen *et al.*, "Timing optimization for multi-level combinational networks," in *Proc. DAC*, 1991, pp. 339–344.

[34] L. A. Entrena *et al.*, "Timing optimization by an improved redundancy addition and removal technique," in *Proc. EURO-DAC*, 1996, pp. 342–347.

[35] S. Yamashita *et al.*, "SPFD: A new method to express functional flexibility," *IEEE TCAD*, vol. 19, no. 8, pp. 840–849, 2000.

[36] L. Amarú *et al.*, "SAT-sweeping enhanced for logic synthesis," in *Proc. DAC*, 2020, pp. 1–6.