Synthesizing a Class of Practical Boolean Functions Using Truth Tables

Yukio Miyasaka[1], Alan Mishchenko[1], John Wawrzynek[1], Nicholas J. Fraser[2]
[1]University of California, Berkeley
[2]AMD AECG Research Labs, Dublin, Ireland
yukio_miyasaka@berkeley.edu

***Abstract*— Many EDA applications deal with logic functions representing complex mathematical computations. Although in many cases, these functions depend on a small number of inputs, they often resemble random functions, making it hard to synthesize them using the traditional methods based on SOP minimization. This paper describes efficient synthesis and LUT mapping for this class of functions using a novel method that implements BDD-based minimization based on truth tables. The paper also investigates optimization with don't cares, when the outputs of a function are unspecified for some inputs, which is particularly useful in machine learning applications that trade accuracy for area. Compared to optimization and mapping used in academic and industrial tools, our method works faster and results in 1.5x smaller networks.**

I. Introduction

Logic synthesis, takes Boolean functions in the form of truth tables, sums-of-products (SOPs) or unoptimized circuits, and produces optimized circuits that are used to map the design into a target technology, typically FPGAs or standard cells. Improving logic synthesis methods remains an important challenge for the developers of modern design automation tools, especially given that design sizes keeps growing while users expect tools to get faster.

In practice, different types of functions call for different logic synthesis methods. For example, Boolean functions appearing in control logic blocks (such as state-machines) are amenable to synthesis by algebraic methods [1] applied to minimized SOPs [2]. These functions having compact SOP representations can be described as *sparse* because their primes tend to have relatively few literals and cover large areas of Boolean space.

In contrast, *dense* Boolean functions are those that do not have compact SOPs. A class of *dense* functions can be found in router designs, where this type of logic is often expressed almost exclusively using large multiplexers and one-hot-encoded selectors. An effective way to handle these circuits is to recognize the multiplexers (or avoid bit-blasting them during elaboration) and perform restructuring, followed by specialized logic sharing extraction. Another class of *dense* functions are those rich in XOR gates, appearing in CRC checkers and cryptographic applications.

Despite decades of research, a "universal" synthesis method has not been discovered. Attempts to apply one synthesis method to all types of logic leads to mediocre results, prohibitive runtime, or both. For example, applying algebraic methods to XOR-rich logic leads to poor quality, while applying them to multiplexer-rich designs often gives good results but only after many synthesis iterations, rendering such an approach impractical due to long runtimes.

In this paper, we develop a novel synthesis method targeting a class of logic functions, which can be characterized as *random-looking dense functions with limited support*, especially for LUT mapping. *Random-looking* implies that these functions are hard for synthesis but they are not random, because truly random functions cannot be compacted by synthesis, but they can be decomposed by brute-force cofactoring [3]. On the other hand, the fact that the functions are *dense* means that they do not have compact SOPs. Finally, the last characteristic, *limited support*, implies that they can be efficiently represented using truth tables stored in small arrays of unsigned integers and manipulated in software using bit-wise operators. In practice, functions up to 16 inputs are often manipulated using truth tables, although in some applications, such as equivalence checking, truth tables have been successfully used for blocks up to 32 inputs, if a longer runtime is acceptable.

Random-looking dense functions with limited support arise in several applications. One of these applications is Deep neural networks (DNNs), where quantized Boolean functions of the neurons are *random-looking* and *dense*, although these functions often do not have *limited support*. However, in some types of neural networks, such as LogicNets [4], the neurons are designed to have limited connectivity and therefore, after quantization, they have *limited support*. Synthesis and mapping of Boolean functions arising in the LogicNets project has served as a primary motivation for this work.

DNNs are not the only source of *random-looking dense functions with limited support*. When designing datapaths, complicated logic blocks, such as exponentiation, sigmoid, or trigonometric functions, are often specified in a tabular form because they are often too complex to be implemented natively in hardware. For example, for an N-bit input, it is simpler to list the 2^N output values for each input value, rather than implement complex functionality using multipliers, adders, shifters, etc. When an M-bit output is required, the resulting implementation is an N-bit-input M-bit-output *random-looking dense function*. When N does not exceed 16, such functions form another target application of our proposed method.

To efficiently deal with the selected class of functions, we developed a novel method that employs truth tables to manipulate Binary Decision Diagrams (BDDs) [5]. However, in a somewhat contradictory way, we are using BDDs

without actually constructing them. As a truth table can be seen as an expanded decision diagram, we avoid explicitly maintaining BDDs with unique tables and other auxiliary data structures, and instead count unique cofactors on each level in a top-down manner. This way we can know the exact number of nodes in the BDDs without actually constructing them. Next, we reorder the truth table, as we would have reordered the BDDs while trying to minimize the number of BDD nodes, and using the don't-cares if they are available. The don't-care based minimization is similar to the known methods on BDD minimization [6], as will be discussed in the background section.

We note several advantages of not explicitly using BDDs:

- the runtime with truth tables is about 1.5x faster
- no need to develop or integrate a BDD package
- the code can be simplified by avoiding recursion

The contributions of the paper are:

- a novel truth-table-based method to perform logic synthesis for LUT mapping with or without don't cares
- isolating a practical class of functions, which allows for an efficient solution using the proposed method
- experimental evaluation demonstrating 1.5x reduction in area and 10x reduction in runtime, compared to the methods implemented in the existing CAD tools

The rest of the paper is organized as follows: Section II describes the background. Section III describes the proposed algorithm. Section IV lists experimental results. Section V concludes the paper.

II. Background

A BDD is a binary tree that represents a single-output logic function [5]. Each non-leaf node is associated with an input variable, and depending on the value of the variable, one of its two child nodes is selected to determine the output value. BDDs are *ordered* if the variable associated with a node always precedes the variable associated with its child node in a given variable order. BDDs are *reduced* if every node has a unique function and is not *redundant*. A redundant node is a node with a function that has identical Shannon cofactors with respect to the associated variable. So, a non-redundant node has two unique child nodes. A child node may be shared by multiple nodes. For a multi-output function, a BDD is built for each output. The BDDs are *shared* if they are ordered using the same variable order and reduced together. EDA applications use ordered, reduced, and shared BDDs, in order to achieve compact representation and efficient manipulation. In this paper, BDDs are always assumed to be ordered, reduced, and shared unless otherwise stated.

To further compress the size of a BDD, *complemented edges* are frequently used [7]. When a child node is connected by a complemented edge, the function of the child node is negated. With complemented edges, we need only one leaf node. In this paper, we use the leaf node 0, since the other constant (1) is its complement. When evaluating the output, we count the number of complemented edges passed from the root node, and flip the output value accordingly. We assume the use of complemented edges in this paper.

Since the size of a BDD depends on the variable order, there have been many studies to find a good variable order that makes the BDD small. Variable reordering by sifting [8] is one of the successful approaches. After building a BDD, this method picks up a variable with the largest number of nodes, and iteratively swaps its position with the adjacent variable in the current variable order. The best position where the smallest BDD was observed, is remembered during the iteration and restored after the iteration. The same procedure is repeated for every variable. This approach is efficient because each variable swap only affects the nodes associated with the two swapped variables [9].

BDD minimization using don't cares was studied in [6]. In principle, BDDs can be minimized by merging nodes, while keeping their function unchanged on the care set—the complement of the don't-care set. They proposed three matching criteria for merging nodes and two heuristics on the order of nodes to compare as follows:

- Merging criteria
  - OSDM: $c_j = 0$
  - OSM: $(c_j \implies c_i) \wedge (c_j \implies (f_i = f_j))$
  - TSM: $(c_i \wedge c_j) \implies (f_i = f_j)$
- Comparison order
  - Sibling: child nodes of each node (depth-first)
  - Level: set of nodes on a given level

where (f_i, c_i) and (f_j, c_j) are the pairs, containing the function and the care set of the nodes to be compared. The OSDM and OSM conditions are not symmetric, so the comparison is performed both ways, and if it holds, (f_i, c_i) replaces (f_j, c_j). If the TSM condition holds, both of the nodes are replaced with a new node (f_k, c_k), which satisfies

$$(c_i \implies (f_i = f_k)) \wedge (c_j \implies (f_j = f_k)),$$
$$c_k = c_i \vee c_j.$$

For OSM and TSM, a *complemented match* is also considered, where f_j in the condition is negated, and the merged node is pointed to by the complemented edge.

The sibling order starts the comparison from the root node and recurs on each child node. If the child nodes have been merged, it recurs only on the merged node. In the level order, they create a set of nodes on a specific level, and perform the comparison for the pairs of nodes in the set. Here, the set may include redundant nodes. Finding the optimal order to pick up a pair is NP-complete, so they use a heuristic where once the nodes are merged, the merged node is compared with the rest of nodes before proceeding to another pair. In the end, [6] proposed applying the sibling order OSM first and the level order TSM later for a partitioned BDD.

III. Proposed method

A. Truth-table-based node counting

Our method is as simple as constructing and reordering BDDs, but we implemented it in a novel way. The advantage of our implementation comes from small memory usage.

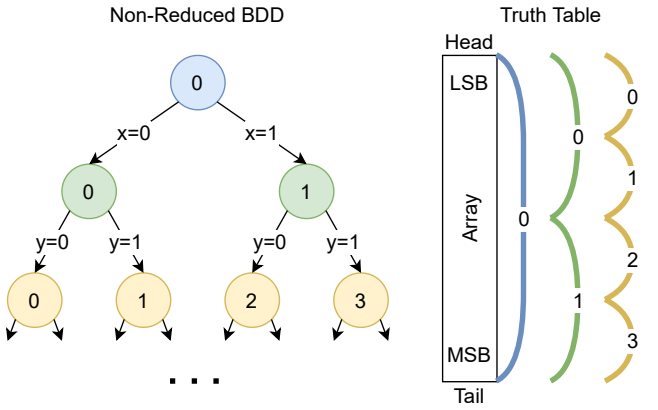

Fig. 1. Indices of potential BDD nodes and the corresponding segments of the truth table for a single output function with input $\{x, y, \dots\}$

Instead of storing the structure of BDDs as pointer-connected nodes, we operate on the given truth table and keep track of indices of unique nodes.

We assume that the truth table is stored as an array, where each 64 bits are packed into one element—called *word*. For example, in the first word, the first bit corresponding to an input pattern $00\dots0$ is stored at the LSB, the second bit corresponding to $00\dots01$ is stored at the second LSB, and so on. The 65th bit of the truth table is stored in the LSB of the second word. For multi-output functions, an array is created for each output and then concatenated to form a larger array.

All potential (non-reduced) BDD nodes are 0-based-indexed for each level as shown in Fig. 1. The function of a node is represented by a segment of the truth table. For example, on the 0-th level, each node corresponds to a segment of 2^N bits, where N is the number of inputs. On the k-th level, each node corresponds to a segment of 2^{N-k} bits.

The proposed top-down procedure to count the number of BDD nodes in the function represented by the truth table, is shown in Code 1. On each level, it creates a vector of *unique indices*, whose functions are unique and non-constant (checked in the sub-procedure in Code 1). Below the 0-th level, it checks only prospective unique indices, which are the cofactors of unique indices on the higher level. Meanwhile, if the cofactors are identical (confirmed by comparing their locations in the vector, returned by the sub-procedure), that unique index is redundant, so to exclude them from the final count, it creates another vector memorizing them. To realize the complemented edges, the function of each index is also compared with the complemented functions of the unique indices already existing in the vector. This complementary information is returned by the sub-procedure along with the location, while constants are expressed by negative locations.

B. Variable reordering

The simplest way to perform variable swap is to sort the entries in the truth table and then recount the nodes in the two swapped levels. The code fragment for variable swap in the truth table is shown in Code 2.

```
// number of inputs
int nInput;
// number of outputs
int nOutput;
// vector of unique indices for each level
vector<vector<int>> vvIdx(nInput);
// vector of redundant indices for each level
vector<vector<int>> vvRedIdx(nInput);

int FindOrAdd(int idx, int lev) {
  if(IsConst0(idx, lev))
    return -2;
  if(IsConst1(idx, lev))
    return -1;
  for(int loc = 0; loc < vvIdx[lev].size(); loc++) {
    if(IsEq(idx, vvIdx[lev][loc], lev))
      return loc << 1;
    if(IsComplEq(idx, vvIdx[lev][loc], lev))
      return (loc << 1) | 1;
  }
  int loc = vvIdx[lev].size();
  vvIdx[lev].push_back(idx);
  return loc << 1;
}

int CountNodes() {
  for(int idx = 0; idx < nOutput; idx++)
    FindOrAdd(idx, 0);
  for(int lev = 1; lev < nInput; lev++)
    for(int idx: vvIdx[lev - 1]) {
      int cof0 = FindOrAdd(idx * 2, lev);
      int cof1 = FindOrAdd(idx * 2 + 1, lev);
      if(cof0 == cof1)
        vvRedIdx[lev - 1].push_back(idx);
    }
  int count = 1; // constant node
  for(int lev = 0; lev < nInput; lev++)
    count += vvIdx[lev].size() - vvRedIdx[lev].size();
  return count;
}
```

Code 1. Counting the number of nodes

```
// 64-bit type
typedef unsigned long long wrd;
// array holding the truth table
wrd t[];
// length of the array
int nWrd;

const wrd swapmask[] = {0x2222222222222222ull,
                        0x0c0c0c0c0c0c0c0cull,
                        0x00f000f000f000f0ull,
                        0x0000ff000000ff00ull,
                        0x00000000ffff0000ull};

void VariableSwap_Table(int lev) {
  int nBitPerSeg = 1 << (nInput - lev - 2);
  if(nBitPerSeg >= 64) {
    int nWrdPerSeg = nBitPerSeg / 64;
    for(int i = 0; i < nWrd; i += nWrdPerSeg * 4)
      for(int j = 0; j < nWrdPerSeg; j++)
        swap(t[i + nWrdPerSeg + j],
             t[i + nWrdPerSeg * 2 + j]);
  } else if(nBitPerSeg == 32)
    for(int i = 0; i < nWrd; i += 2) {
      t[i + 1] ^= t[i] >> 32;
      t[i] ^= t[i + 1] << 32;
      t[i + 1] ^= t[i] >> 32;
    }
  else
    for(int i = 0; i < nWrd; i++) {
      wrd mask = swapmask[log2(nBitPerSeg)];
      t[i] ^= (t[i] >> nBitPerSeg) & mask;
      t[i] ^= (t[i] & mask) << nBitPerSeg;
      t[i] ^= (t[i] >> nBitPerSeg) & mask;
    }
}
```

Code 2. Sorting the truth table for variable swap

```
// vector of cofactors for each level
vector<vector<int>> vvCof(nInput);

int CountNodes2() {
  for(int idx = 0; idx < nOutput; idx++)
    FindOrAdd(idx, 0);
  for(int lev = 1; lev < nInput + 1; lev++)
    for(int idx: vvIdx[lev - 1]) {
      int cof0 = FindOrAdd(idx * 2, lev);
      int cof1 = FindOrAdd(idx * 2 + 1, lev);
      if(cof0 == cof1)
        vvRedIdx[lev - 1].push_back(idx);
      /* store cofactors of unique indices*/
      vvCof[lev - 1].push_back(cof0);
      vvCof[lev - 1].push_back(cof1);
    }
  int count = 1;
  for(int lev = 0; lev < nInput; lev++)
    count += vvIdx[lev].size() - vvRedIdx[lev].size();
  return count;
}
```

Code 3. Storing cofactors of unique indices when counting the number of nodes

However, although sorting the truth table is simple, it is not as efficient as variable swap in the form of BDD. Reconstructing BDDs after variable swap takes a linear time over the number of nodes in the level below. On the other hand, sorting takes a linear time over the number of truth table segments therein, which is an upper bound of the former number.

The proposed implementation adopts a hybrid approach. When counting the number of nodes for the first time, we also memorize the cofactors of the unique indices as shown in Code 3. Then, we perform variable swap in a traditional way while instantiating a unique table as shown in Code 4. We reconstruct the lower level based on 2-level cofactors using the unique table; refer to [9] for more details. This unique table is a bit simpler than usual since it uses only a pair of cofactors as a key. The unique indices in the higher level remain the same, while their redundancy will be reevaluated using the new cofactors.

C. BDD minimization using don't cares

We implemented BDD minimization using don't cares on top of our truth-table-based algorithm. We adopted the level order TSM, as presented in Section II. The sibling order does not match our top-down algorithm. The reason why we chose TSM is that, in the case of OSDM and OSM, we have to check redundancy of nodes afterwards in a bottom-up manner. Even if the child nodes look different when they are checked, their functions could be modified by node merging later. For example, let us assume that the 00-cofactor and the 11-cofactor have the same function and no don't care, while the 01-cofactor and the 10-cofactor have different functions but are entirely don't care. Since the 0-cofactor and 1-cofactor do not satisfy the merging condition, they are regarded as different unique indices. However, when checking their level, the 01-cofactor will be merged with the 00-cofactor, and the 10-cofactor will be merged with the 11-cofactor, having the same function. Then, not only the 0-cofactor and 1-cofactor become redundant, their parent node also becomes redundant. This does not happen in TSM

```
// hash table type
typedef unordered_map<pair<int, int>, int> ht;

/* find or add a unique index in the lower level */
int FindOrAdd2(int idx, int cof0, int cof1, int lev,
               ht &ut, vector<int> &vCofLow) {
  /* check constant */
  if(cof0 < 0 && cof0 == cof1)
    return cof0;
  /* convention for complemented edges */
  bool fCompl = cof0 & 1;
  if(fCompl)
    cof0 ^= 1, cof1 ^= 1;
  /* check uniqueness by cofactors */
  if(ut.count({cof0, cof1}))
    return (ut[{cof0, cof1}] << 1) ^ fCompl;
  /* add the index to unique indices */
  int loc = vvIdx[lev].size();
  vvIdx[lev].push_back(idx);
  /* register the location with unique table */
  ut[{cof0, cof1}] = loc;
  /* store cofactors */
  vCofLow.push_back(cof0);
  vCofLow.push_back(cof1);
  /* check redundancy */
  if(cof0 == cof1)
    vvRedIdx[lev].push_back(idx);
  /* return location */
  return (loc << 1) ^ fCompl;
}

void VariableSwap_Node(int lev) {
  // vectors to store new cofactors
  vector<int> vCofHigh, vCofLow;
  // unique table
  ht ut(2 * vvIdx[lev + 1].size());
  /* clear vectors of indices to be updated */
  vvIdx[lev + 1].clear();
  vvRedIdx[lev].clear();
  vvRedIdx[lev + 1].clear();
  /* loop for each unique index in the higher level */
  for(int loc = 0; loc < vvIdx[lev].size(); loc++) {
    int idx = vvIdx[lev][loc];
    /* get locations of cofactors */
    int cof0loc = vvCof[lev][loc * 2] >> 1;
    bool cof0c  = vvCof[lev][loc * 2] & 1;
    int cof1loc = vvCof[lev][loc * 2 + 1] >> 1;
    bool cof1c  = vvCof[lev][loc * 2 + 1] & 1;
    /* get locations of 2-level cofactors */
    int cof00, cof01, cof10, cof11;
    if(cof0loc < 0)
      cof00 = cof01 = -2 ^ cof0c;
    else
      cof00 = vvCof[lev + 1][cof0loc * 2] ^ cof0c,
      cof01 = vvCof[lev + 1][cof0loc * 2 + 1] ^ cof0c;
    if(cof1loc < 0)
      cof10 = cof11 = -2 ^ cof1c;
    else
      cof10 = vvCof[lev + 1][cof1loc * 2] ^ cof1c,
      cof11 = vvCof[lev + 1][cof1loc * 2 + 1] ^ cof1c;
    /* calculate new cofactors */
    int newcof0 = FindOrAdd2(idx * 2, cof00, cof10,
                             lev + 1, ut, vCofLow);
    int newcof1 = FindOrAdd2(idx * 2 + 1, cof01, cof11,
                             lev + 1, ut, vCofLow);
    /* store new cofactors */
    vCofHigh.push_back(newcof0);
    vCofHigh.push_back(newcof1);
    /* check redundancy */
    if(newcof0 == newcof1)
      vvRedIdx[lev].push_back(idx);
  }
  /* update vectors of cofactors */
  vvCof[lev] = vCofHigh;
  vvCof[lev + 1] = vCofLow;
}
```

Code 4. Variable swap based on nodes

because, if a node can be redundant, its child nodes must satisfy the condition and be merged immediately.

We can perform the minimization simply by replacing the equivalence check of nodes by TSM. If an existing unique index j matches the given index i, it updates the function and the care set of j to $(ITE(c_i, f_i, f_j), c_i \vee c_j)$, which is an example of (f_k, c_k) shown above.

With BDD minimization using don't cares, reordering is no longer simple. The result of variable swap is affected by the minimization performed previously. To see the actual effect of variable reordering, we have to restore the original BDDs before variable swap. While this looks complicated, it can be integrated quite easily with our truth-table-based implementation; what we must do is to only reload and sort the original truth table, and rerun the procedure. Additionally, we can save time by remembering the nodes above the swapped level and how they were merged. This approach, performing reordering based on the result of minimization, results in a significant reduction in the number of nodes as compared to just performing the minimization to reordered BDDs.

IV. Experimental results

A. Benchmarks

We used Boolean functions from the LogicNets project [4] as our target. These are functions of quantized sparse neural networks, where each neuron is represented as a truth table to comprise a truth table network. Let β denote the bit-width of each activation and γ denote the number of input activations per neuron, which are uniform across each network. The number of inputs is $\beta \times \gamma$ and the number of outputs is β for each truth table. The properties of the neural networks used to generate functions in our experiments are shown in Table I.

For some of the networks listed in Table I, the number of inputs and outputs to each truth table is different for the first and last layers of the network. Specifically, the input bit-width and activations in the first layer are given by β_i and γ_i, respectively. Similarly, the input activations for the last layer is given by γ_o, and the output bit-width of the last layer is given by δ. The number of inputs and outputs for each truth table in the first layer is $\beta_i \times \gamma_i$ and β, and in the last layer is $\beta \times \gamma_o$ and δ. For all networks in Table I, when $\beta_i \neq \beta$, $\gamma_i \neq \gamma$, $\gamma_o \neq \gamma$ or $\delta \neq \beta$, the values are specified below:

- JSC_L: $\beta_i = 4$, $\gamma_i = 3$, $\gamma_o = 5$, $\delta = 7$
- NID_S: $\beta_i = 1$
- NID_M: $\beta_i = 1$
- NID_L: $\beta_i = 1$, $\gamma_i = 7$

B. Performance of truth-table-based algorithm

We first conducted a runtime comparison against an existing BDD package, CUDD [10]. We performed variable reordering by sifting, starting with a random initial variable order, 20 times for each neuron [11]. We used a Intel Core i7-9750H Processor for this experiment.

TABLE I
LogicNets benchmark

Name	Neurons per layer	β	γ	Accuracy
JSC_S	64, 32, 32, 32	2	3	69.41%
JSC_M	64, 32, 32, 32	3	4	71.90%
JSC_L	32, 64, 192, 192, 16	3	4	73.01%
NID_S	593, 100	2	7	89.36%
NID_M	593, 256, 128, 128	2	7	92.62%
NID_L	593, 100, 100, 100	3	5	93.12%

TABLE II
Performance comparison (time in sec and memory in MB)

	CUDD		Swap Table		Swap Node	
	Time	Mem	Time	Mem	Time	Mem
JSC_S	2.46	14.0	0.06	3.6	0.09	3.6
JSC_M	4.60	14.2	7.31	3.7	2.85	3.9
JSC_L	14.56	18.6	29.09	4.1	9.40	4.4
NID_S	12.29	14.5	7.00	3.8	3.13	4.0
NID_M	22.90	14.8	17.29	4.0	8.52	4.0
NID_L	24.23	16.6	55.55	4.4	14.38	4.4

The runtime and memory footprint are shown in Table II. Our truth-table-based algorithm used less memory than CUDD, and with the node-based variable swap, it worked more than 1.5x faster than CUDD. CUDD takes a large amount of memory to maintain data that are useful to dynamically apply many operations, which is not the case here. Our truth table algorithm uses memory just to store the truth table and the unique indices if variable swap is performed by sorting the truth table. With node-based swap, it also needs a storage for cofactors, which explains the observed overhead. Except JSC_S, node-based swap was the fastest. JSC_S is a special case where each truth table fits in one word per output, and the truth-table-based implementations worked much faster than CUDD.

C. LUT mapping comparison

We performed synthesis and mapping for 6-LUT networks. For each neuron, we construct and reorder BDDs and map them into 6-LUTs. We did not use don't cares in this experiment. Since each BDD node is a multiplexer, we constructed a LUT network by grouping the multiplexers in a depth-first order to have nodes with at most six inputs. Typically, two cascaded multiplexers are grouped into a node with five inputs. Then, as an inter-neuron optimization, we apply ABC's optimization [12] ("mfs2" and "&if -K 6 -a") iteratively as long as the area keeps on reducing. The command "mfs2" optimizes a given network without converting it into AIG, and "&if" reperforms mapping to restructure the network and create more optimization opportunities for "mfs2". Other AIG optimization commands in ABC did not work well to reduce the LUT count for these benchmarks, sometimes resulting in a larger network than without optimization.

We compared our method with Xilinx Vivado 2021.2 with default synthesis settings, but with "resource sharing" set to "on". We also checked the results when we apply ABC's optimization to the original networks without applying the proposed algorithm. Note that the following experiments

TABLE III
SYNTHESIS AND LUT MAPPING

	Vivado		ABC only		Proposed method	
	LUTs	Time(s)	LUTs	Time(s)	LUTs	Time(s)
JSC_S	227	56	242	1	233	3
JSC_M	14865	1159	31647	22	9665	45
JSC_L	35419	2373	87065	63	22997	144
NID_S	85	3191	30	12	29	49
NID_M	2690	7135	4080	51	1969	88
NID_L	6672	15248	13888	205	4057	277

TABLE IV
SYNTHESIS AND LUT MAPPING FOR DISCRETE COSINE TRANSFORM

Vivado		ABC only		Proposed method	
LUTs	Time(s)	LUTs	Time(s)	LUTs	Time(s)
859	71	6960	8	507	2

were done using a different environment: Intel Xeon Silver 4110 Processor.

The results are shown in Table III. In general, Vivado took the longest time, 10x slower than the proposed method on average, while only running ABC did not work well ending up with the largest overall area. Except JSC_S, our method achieved about 1.5x area reduction compared to Vivado.

D. Optimization using don't cares

Finally, we performed don't-care-based optimization when some input patterns are treated as don't cares. To begin with, we assigned don't care for the patterns that do not appear at the inputs of a neuron on any examples from the training set. Furthermore, we used a threshold, called *rarity*, where the patterns that occur at least *rarity* times in the training set are cares, while other patterns are don't cares. The same pattern can appear multiple times in the training set because different examples may result in the same quantized values at the inputs on a neuron. We considered values of *rarity* equal to powers of two.

The results are shown in Fig. 2. The LUT count decreased as *rarity* increased, except for NID_S. Meanwhile, the accuracy decreased quite slowly. For example, JSC_L started at 22997 LUTs and accuracy 73.01%. With don't cares generated by *rarity* 64, the LUT count decreased to 10914, while the accuracy was still 71.06%. Regarding NID_S, the network seems too small for LUT mapping to fully reflect the results of BDD minimization. The runtime was at most three minutes for each run.

E. Additional results of LUT mapping

We conducted another experiment on a mathematical function. It is a 12-input 32-output function that performs discrete cosine transform, taken from [13]. We synthesized and mapped it into 6-LUTs without don't cares. Table IV shows that our method outperformed Vivado achieving more than 1.5 area reduction for this function as well. Comprehensive experiment on mathematical and other functions is part of our future work.

V. CONCLUSION

The paper motivates the development of specialized logic synthesis methods for well-defined classes of Boolean functions. To this end, we isolate a practical class of Boolean functions characterized as *random-looking dense functions with limited support*. We observe that these functions appear in quantized neural networks, datapath applications, cryptographic applications, and possibly other areas.

For the selected class of functions, a novel synthesis method is proposed. The method iteratively reorders the truth table representation of the function while trying to minimize the number of nodes, which would be present in the BDDs of the function. The method efficiently exploits don't cares if they are present in the specification of the function.

The experimental results after LUT mapping show that the area improvements produced by the proposed methods are substantial, leading to 1.5x reduction, compared to the results produced on these benchmarks by the available tools, both academic and commercial. The proposed method is also often at least 10x faster than those used by the tools.

A separate experiment motivates the use of truth tables by showing that they speed up the implementation by about 1.5x, compared to using a state-of-the-art BDD package.

The MUX-based circuits derived by our method have an advantage in Versal programmable architecture, which allows a 7-input function composed of three MUXes, MUX(x0, MUX(x1, C11, C10), MUX(x2, C01, C00)), to be mapped into one 6-LUT by utilizing additional hardware resources. This is in contrast with the previous architectures, which use one 6-LUT only if it is a 6-input function (that is, x1=x2). This would potentially improve the area (LUT count) produced by our method even more, since many MUX structures, which currently require two 6-LUTs, can be packed into one 6-LUT.

REFERENCES

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

[2] R. Rudell *et al.*, "Multiple-valued minimization for PLA optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 5, pp. 727–750, 1987.

[3] T. Sasao *et al.*, "LUTMIN: FPGA logic synthesis with MUX-based and cascade realizations," in *Proceedings of IWLS*, 2009, pp. 310–316.

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

[5] Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[6] T. R. Shiple *et al.*, "Heuristic minimization of BDDs using don't cares," in *Proceedings of DAC*, 1994, pp. 225–231.

[7] K. Brace *et al.*, "Efficient implementation of a BDD package," in *Proceedings of DAC*, 1990, pp. 40–45.

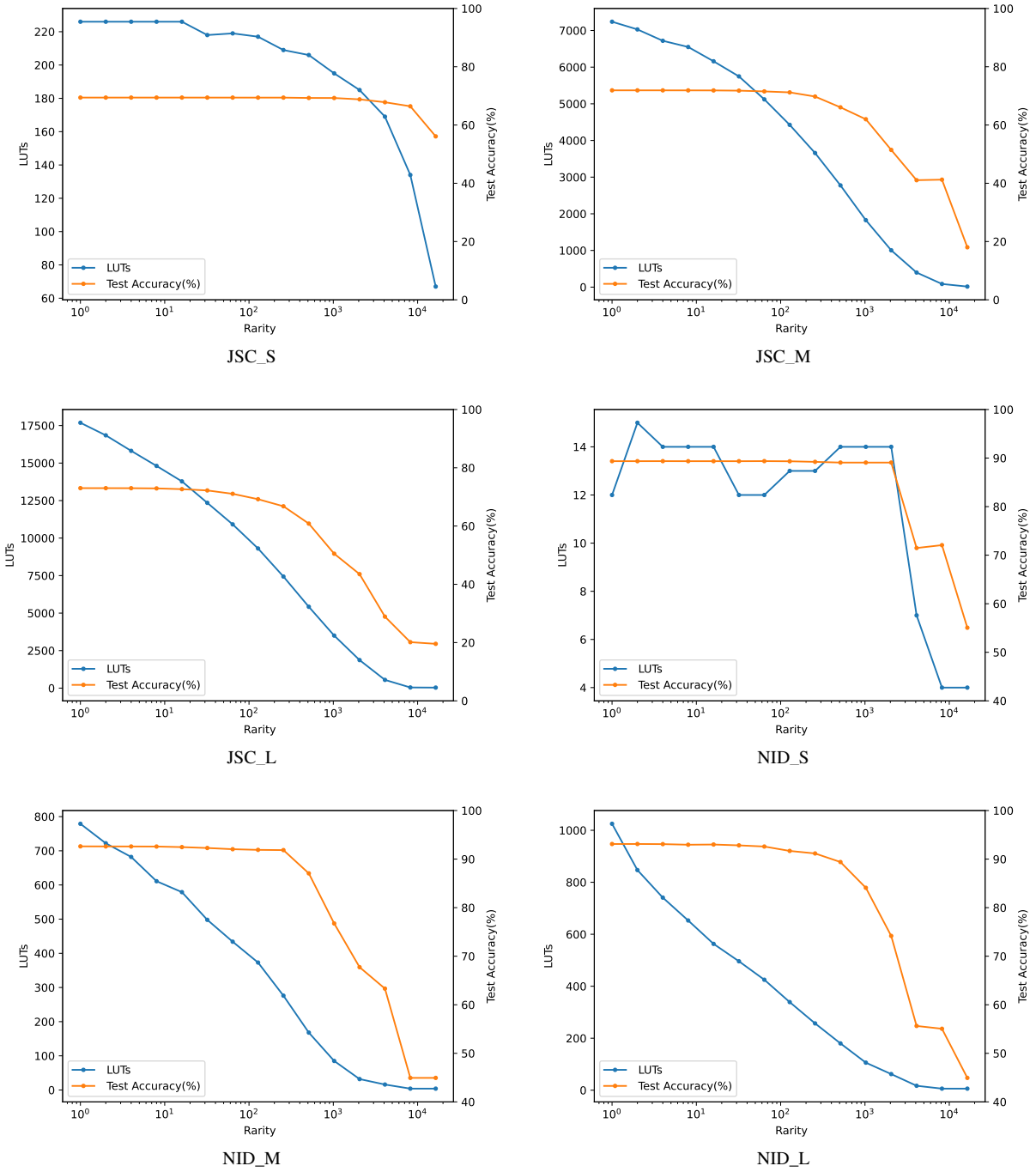

Fig. 2. Effect of the minimization using don't cares when changing rarity

[8] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of ICCAD*, 1993, pp. 42–47.

[9] M. Fujita *et al.*, "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis," in *Proceedings of DATE*, 1991, pp. 50–54.

[10] F. Somenzi, "Efficient manipulation of decision diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 171–181, 2001.

[11] P. Fiser *et al.*, "How much randomness makes a tool randomized?" In *Proceedings of IWLS*, 2011.

[12] Berkeley Logic Synthesis and Verification Group, *ABC: A system for sequential synthesis and verification, release 20306.* `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[13] *OpenCores: Video compression systems*, `https://opencores.org/projects/video_systems`.