Disjoint-Support Decomposition and Extraction for Interconnect-Driven Threshold Logic Synthesis

Hao Chen, Shao-Chun Hung, and Jie-Hong R. Jiang
Department of Electrical Engineering / Graduate Institute of Electronics Engineering
National Taiwan University, Taipei 10617, Taiwan

ABSTRACT

Threshold logic circuits are artificial neural networks with their neuron outputs being binarized, thus amenable for efficient, multiplier-free, hardware implementation of machine learning applications. In the reviving threshold logic synthesis, this work lays the foundations of disjoint-support decomposition and extraction operation of threshold logic functions. They lead to a synthesis procedure for interconnect minimization of threshold logic circuits, an important, but not well addressed, objective in both neural network and nanometer circuit designs. Experimental results show that our method can efficiently and effectively reduce interconnect as well as weight/threshold value over highly optimized circuits, thus suitable for implementation using emerging technologies.

CCS CONCEPTS

• **Hardware → Combinational synthesis**; **Circuit optimization**;

KEYWORDS

decomposition, extraction, threshold logic

ACM Reference Format:
Hao Chen, Shao-Chun Hung, and Jie-Hong R. Jiang. 2019. Disjoint-Support Decomposition and Extraction for Interconnect-Driven Threshold Logic Synthesis. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317801

1 INTRODUCTION

As Moore's law hits the physical limit, new alternative computing architectures and devices are under active research. On the architecture side, there is non-von Neumann architecture for neuromorphic computing, which can achieve effective implementation of neural networks for machine learning applications. On the device side, alternatives beyond CMOS, such as resonant tunneling devices (RTD) [20], quantum cellular automata [3], single electron transistors [12], memristors [9], and spintronic devices [8], have been demonstrated. Threshold logic turns out to be a promising circuit model under these architecture and device innovations based on neuron-like computation. Various nanoscale devices have been used to implement threshold logic circuits [1]. On the other hand, recent progress in deep learning has made neural networks become a popular model to perform tasks in artificial intelligence. Hardware acceleration of neural networks is an active research area. Neural network binarization for effective hardware realization has been demonstrated [6]. Essentially threshold logic networks are neural networks with their activation functions being binarized. The advancements of synthesis and verification of threshold logic circuits may have important practical implications. These trends revive threshold logic research in recent years.

Among prior efforts on threshold logic synthesis, prior work [19, 20] decomposes Boolean functions into a network of threshold logic functions based on Shannon expansion. A decomposition algorithm based on the truth table and binate splitting heuristic is proposed in [19]. In [10], a tree matching method is applied for threshold circuit synthesis. In [17], an implicant-implicit method is proposed. In [16], the and-inverter graph (AIG) and cut-based technology mapping are applied. In [11], [4], and [13], rewiring, merging and collapse operations are proposed, respectively. Common synthesis objectives of threshold logic circuits are minimizing gate counts, circuit depths, and/or the magnitudes of weight/threshold values. In this work, we address the synthesis objective from a different aspect of minimizing the interconnect complexity of threshold logic circuits, which is a crucial issue in machine learning applications as neural networks are often layerwise densely connected. Recently a synthesis method [5] is proposed to reuse partial summation in binarized neural networks for interconnect minimization. However its detailed implementation assumption of a binarized neuron makes it not applicable to threshold logic synthesis.

In this paper, we lay the theoretical foundation toward disjoint-support decomposition for threshold logic functions and extraction of common sub-threshold logic functions for a given set of multiple threshold logic functions. Essentially the enabling technique is the *generalized decision list*, which we define as an extension to the decision list [7, 18], for Boolean function representation. Further we seek practical applications of the extraction operation to interconnect minimization of threshold logic circuits. Such minimization may directly benefit area cost reduction for implementations with the spintronics- and memrister-based technologies. Experimental results demonstrate that effective interconnect reduction by 10% (resp. 8%) on average can be achieved efficiently on threshold logic circuits that have been highly optimized for delay (resp. area). As a by-product, the magnitudes of weight/threshold values of threshold logic gates can be reduced by 14% (resp. 13%) on average for delay-minimized (resp. area-minimized) circuits. This effect also benefits area cost reduction in RTD-based technology for threshold logic implementation.

The remaining sections are organized as follows. Section 2 gives the preliminaries. The decomposition and extraction operations are detailed in Sections 3 and 4, respectively. Section 5 shows the experimental results, and Section 6 concludes this paper.

2 PRELIMINARIES

A *literal* is a variable or the negation of a variable. A *cube* is a conjunction of literals, and will be alternatively treated as a set of literals in the sequel. Given a Boolean function $f(x_1, \ldots, x_i, \ldots, x_n)$, its *negative cofactor*, i.e., $f(x_1, \ldots, 0, \ldots, x_n)$, and *positive cofactor*, i.e., $f(x_1, \ldots, 1, \ldots, x_n)$, on variable x_i are denoted as $f|_{\neg x_i}$ and $f|_{x_i}$, respectively. The notion of cofactor on a single literal

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317801

can be extended to a cube. That is, the cofactor of f on a cube $c = l_1 \wedge \cdots \wedge l_k$ is defined as iterative cofactoring f on literals $l_1, \ldots, l_k$ and denoted as $f|_c$. Given an (ordered) set X of Boolean variables, its set of valuations/assignments is denoted as $[\![X]\!]$. E.g., $[\![(x_1, x_2)]\!] = \{00, 01, 10, 11\}$. An assignment to a set X of variables is alternatively represented as a cube. E.g., assignment $(x_1 = 1, x_2 = 0)$ is represented as $x_1 \neg x_2$. The *onset*, denoted f^1, and *offset*, denoted f^0, of a function f over variables X are the sets of assignments that valuate f to 1 and 0, respectively. A function f is said *positive unate* (resp. *negative unate*) on variable x_i if the implication $f|_{\neg x_i} \rightarrow f|_{x_i}$ (resp. $f|_{x_i} \rightarrow f|_{\neg x_i}$) holds. A function f is called a *unate function* if it is either positive or negative unate on every variable. A function f is said *symmetric* on variables x_i and x_j if $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, x_j, \ldots, x_i, \ldots)$. In fact, f is *symmetric* on variables x_i and x_j if and only if $f|_{x_i \neg x_j} = f|_{\neg x_i x_j}$. As the symmetry property is transitive, i.e., if variables x_1 and x_2 are symmetric and x_2 and x_3 are symmetric, then x_1 and x_3 are symmetric in f. In this case, $\{x_1, x_2, x_3\}$ forms a *symmetric group.*

A *threshold logic function* (TLF) [15] $f : \mathbb{B}^n \rightarrow \mathbb{B}$ over input (support) variables $x_1, \ldots, x_n$ is a Boolean function that can be specified with a vector of parameters, denoted $[w_1, \ldots, w_n; T]$, such that

$$f = \begin{cases} 1, & if \sum_{i=1}^{n} w_i x_i \geq T, \\ 0, & otherwise, \end{cases} \quad (1)$$

where parameter $w_i \in \mathbb{Z}$ represents the weight of input x_i, and $T \in \mathbb{Z}$ is the threshold value. Note that a TLF f must be a unate function. Specifically, f is positive (resp. negative) unate on x_i if $w_i > 0$ (resp. $w_i < 0$). Any TLF can be converted to a function that is positive unate on every variable by inverting its negative unate inputs [15]. In the sequel, unless otherwise stated we assume that the weights of a TLF have been transformed to positive values.

A *threshold logic network* (TLN), or called a *threshold logic circuit* (TLC), is a directed acyclic graph $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. The set V consists of three disjoint subsets: primary inputs (PIs), primary outputs (POs), and intermediate nodes representing *threshold logic gates* (TLGs), where each TLG is referred to as a physical primitive logic unit that implements a TLF. The set $e = (u, v) \in E$ signifies the interconnections from vertex u to v. In this case, we say u is a *fanin* of v, denoted $u \in FI(v)$, and v is a *fanout* of u, denoted $v \in FO(u)$. For a TLG v with n fanins $v_1, \ldots, v_n$, it realizes some threshold function f_v specified by $[w_{v_1}, \ldots, w_{v_n}; T_v]$ over variables $x_{v_1}, \ldots, x_{v_n}$, where x_{v_i} corresponds to the output variable of v_i.

The implementation cost of a TLN can be measured by the total cost of its constituent TLGs and interconnections. To implement a TLG, several approaches including CMOS and emerging technologies have been proposed. For RTD-based implementations, the area of a TLG is mainly affected by the magnitudes of weight and threshold values. Hence the total area of a TLN is estimated by the summation of weights and threshold value in each TLG; while for spintronics-based and memrister-based implementations, the most critical factor which determines the area of a TLG is the total number of fanins [8, 9, 20].

3 THRESHOLD LOGIC DECOMPOSITION

3.1 Generalized Decision List

To characterize the feasibility of the threshold gate decomposition, we extend the concept of decision list [7, 18], and define the generalized decision list.

Definition 1. *The* generalized decision list *(GDL) of a function f over variables $X = \{x_1, \ldots, x_n\}$ is an ordered list of three-tuple nodes*

$$(X_1, \alpha_1, \beta_1), \ldots, (X_m, \alpha_m, \beta_m)$$

where $X_1, \ldots, X_m$ are non-empty variable sets forming a partition on X, i.e., $\bigcup_i X_i = X$ and $X_i \cap X_j = \emptyset$ for $i \neq j$, and $\alpha_i \subseteq [\![X_i]\!]$ and $\beta_i \subseteq [\![X_i]\!]$ are the sets of assignments on X_i, along with any assignment on X_j not in $\alpha_j \cup \beta_j$ for $j = 1, \ldots, i-1$, that valuate f to 1 and 0, respectively, regardless of the assignments on variables $X_{i+1} \cup \cdots \cup X_m$.

Example 1. *The Boolean function $f = x_1 \vee (x_2 \wedge x_3)$ can be expressed by the GDL: $(x_1, \{1\}, \emptyset)$, $((x_2, x_3), \{11\}, \{10, 01, 00\})$.*

Algorithm 1 *ConstructGDL*(f)

Input: A TLF $f(x_1, \ldots, x_n)$ with $[w_1, \ldots, w_n; T]$ for $w_i \geq w_{i+1} > 0$.
Output: A GDL L of f.
1: $L := \emptyset, X := \emptyset$;
2: **for** $i = 1$ to n
3:     $X := X \cup \{x_i\}$;
4:     **if** no non-controlling assignment on X
5:         add node (X, *controlling-1*(X), *controlling-0*(X)) into L;
6:         **break** ;
7:     **else if** $f|_a = f|_b$ for all a, $b \in$ non-controlling($\{x_1, \ldots, x_i\}$)
8:         add node (X, *controlling-1*(X), *controlling-0*(X)) into L;
9:         $X := \emptyset$;
10: **return** L;

For a GDL node (X_i, α_i, β_i), an assignment in α_i (resp. β_i) is referred to as a *controlling-1* (resp. *controlling-0*) *assignment*, and an assignment not in $\alpha_i \cup \beta_i$ is referred to as a *non-controlling assignment.*

In fact, any Boolean function can be represented as a GDL. To characterized the feasibility of decomposition and extraction of threshold logic functions, we use GDL to represent a TLF. Algorithm 1 provides a conversion procedure from a TLF to a GDL. The for-loop in Lines 2-9 iterates over the input variables in a descending order with respect to their weight values. In Line 3, the target input x_i is added into the temporary variable vector X. If there is no non-controlling assignment in $[\![X]\!]$, the corresponding GDL node is added to the list L in Line 5, and then L is ready to be returned in Line 10 as all onset and offset assignments of f have already been covered. Otherwise, if there is some assignment c in $[\![(x_1, \ldots, x_i)]\!]$ under which the value of f remains undetermined, we check whether all such $f|_c$ are functionally equivalent. If yes, a corresponding GDL node is created in Line 8, and vector X is emptied in Line 9. Else, the process continues with another iteration.

The time complexity of algorithm *ConstructGDL* can be analyzed as follows. Given an n-variable TLF f, the for-loop in Lines 2-9 will be executed at most n times. For each iteration, the bottleneck occurs at Line 7, where the checking can be reformulated implicitly as satisfiability checking or done explicitly via enumeration through all non-controlling assignments. Despite the expensive computation, algorithm *ConstructGDL* can be efficient in practical applications. As prior work on threshold logic synthesis mostly restricts the fanin number of a TLG to a low level, e.g., 8 inputs in [16], the runtime overhead for the checking can be low. Moreover, in application benchmarks, the number of non-controlling assignments equals 1 frequently.

To ease our discussion, we define three types of GDL nodes:

Definition 2. *A node (X_i, α_i, β_i) in a GDL that constructed by algorithm* ConstructGDL *is a* type-0 *node if X_i is a singleton set and*

$\beta_i = \{0\}$, *a* type-1 *node if* X_i *is a singleton set and* $\alpha_i = \{1\}$, *and a* type-2 *node if* $|X_i| \geq 2$.

Example 2. *Consider the TLG in Figure 1(a). By Algorithm* ConstructGDL, *its GDL is obtained as* $(x_1, \{1\}, \emptyset)$, $(x_2, \{1\}, \emptyset)$, $(x_3, \emptyset, \{0\})$, $(x_4, \emptyset, \{0\})$, $((x_5, x_6), \{11\}, \{00\})$, $(x_7, \{1\}, \{0\})$. *It is visualized as the list shown in Figure 1. The first two nodes are of type-1, the third and fourth nodes are of type-0, the fifth node is of type-2, and the last node is of either type-0 or type-1.*

Given a GDL L of a TLF f with $[w_1, \ldots, w_n; T]$, it can be observed that any suffix of L must correspond to some TLF. Such suffix TLF can be obtained through the following derivation. Without loss of generality, consider the derivation of the TLF g corresponding to the $(n-1)$-node suffix of L. There are three cases: For the first node being a type-0 node with input x_1, only the non-controlling assignment, i.e., $x_1 = 1$, matters to g. Specifically, function $g = f|_{x_1}$ and can be expressed by $[w_2, \ldots, w_n; T - w_1]$. Similarly, for the first node being a type-1 node, function $g = f|_{\neg x_1}$ and can be expressed by $[w_2, \ldots, w_n; T]$. On the other hand, for the first node being a type-2 node with inputs $x_1, \ldots, x_k$, let c being the assignment in $[\![(x_1, \ldots, x_k)]\!]$ with the minimum value of $\sum_{x_i \in c} w_i$, where "$x_i \in c$" indicates literal x_i is in the cube c. Then function $g = f|_c$ and can be expressed by $[w_{k+1}, \ldots, w_n; T - \sum_{x_i \in c} w_i]$. By repeating the above process, all suffix TLFs can be obtained.

The constructed GDL enjoys the following property.

Lemma 1. *Given an n-variable TLF f and its GDL L, the set of variables in contiguous type-0 (type-1) nodes of L forms a symmetric group of f.*

With Algorithm 1, the following property can be established.

Theorem 1. *Given an n-input TLF f, the GDL L constructed by algorithm* ConstructGDL *is the longest. That is, there exists no other GDL* L' *of f such that* $|L'| > |L|$.

3.2 Disjoint-Support Decomposition

With GDL, we study the feasibility of disjoint-support decomposition of a TLF. Formally, a disjoint-support decomposition of a TLF f over variables $X = A \cup B$ with $A \cap B = \emptyset$ is to rewrite $f(X)$ as $h(A, g(B))$ for h and g being TLFs. We call variables A and B being the *free set* and *bound set*, respectively. To avoid trivial decomposition, we shall assume $A \neq \emptyset$ and $|B| \geq 2$.

The following theorem relates GDL and the decomposability of a TLF.

Theorem 2. *Given a TLF* $f(X)$ *with* $X = A \cup B$ *for* $A \neq \emptyset$ *and* $|B| \geq 2$, *a disjoint-support decomposition* $f(X) = h(A, g(B))$ *is feasible if there exists a GDL L such that one of the following three cases holds:*

1. *The bound set variables B appear in some contiguous* type-0 *nodes in L.*
2. *The bound set variables B appear in some contiguous* type-1 *nodes in L.*
3. *The bound set variables B appear in the nodes of some suffix list of L.*

The theorem can be proved constructively by deriving functions h and g with respect to the three cases of Theorem 2. For a TLF f with $[w_1, \ldots, w_n; T]$, assume the bound set variables $B = \{x_k, \ldots, x_l\}$, where $1 \leq k < l \leq n$. In the first case,

$$g = [1, \ldots, 1; |B|],$$
$$h = [w_1, \ldots, w_{k-1}, w_g, w_{l+1}, \ldots, w_n; T],$$

for $w_g = \sum_{j=k}^{l} w_j$. In the second case,

$$g = [1, \ldots, 1; 1],$$
$$h = [w_1, \ldots, w_{k-1}, w_g, w_{l+1}, \ldots, w_n; T],$$

for $w_g = w_k$. In the third case,

$$g = [w_k, \ldots, w_n; T - \sum_{x_i \in c} w_i],$$
$$h = [w_1, \ldots, w_{k-1}, w_g; T],$$

for $w_g = T - \sum_{x_i \in c} w_i$, where c is the assignment in $[\![(x_1, \ldots, x_{k-1})]\!]$ with the minimum value of $\sum_{x_i \in c} w_i$ as discussed in suffix TLF derivation. It can be verified that the above derivations establish $f(X) = h(A, g(B))$.

Example 3. *Figure 1(b), (c) and (d) show three examples of disjoint-support decomposition for the TLF in (a) corresponding to the three cases stated in Theorem 2, respectively.*

The GDL L constructed by *ConstructGDL* exhibits the nice property that if a bound set B appears in some GDL L' as stated in Theorem 2, then B can also be found in L. The property is formally stated in Theorem 3.

Theorem 3. *In Theorem 2, it suffices to consider only the longest GDL constructed by algorithm* ConstructGDL.

By the above theorems, we know that our GDL construction algorithm can generate the longest GDL of a TLF, and that all options of disjoint-support decomposition stated in Theorem 2 can be covered.

4 THRESHOLD LOGIC EXTRACTION

4.1 Problem Formulation

Given a TLN with multiple TLGs having the same disjoint decomposition option, it is possible to extract a common decomposition TLF for logic sharing among the TLGs. Such an *extraction* operation may reduce interconnect complexity of the TLN. To compute extraction candidates, we define *extraction set* and *extraction pair* of a TLF as follows.

Definition 3. *An* extraction set *is a set of nodes in a GDL whose corresponding variable set forms a legal bound set for disjoint-support decomposition. A set with contiguous type-0 (resp. type-1) nodes is called a* type-0 extraction set *(resp.* type-1 extraction set*). An* extraction pair (S, v) *consists of an extraction set S and a corresponding TLG v where S comes from.*

Example 4. *For the TLN in Figure 3(a), the set* $S = \{(x_3, \emptyset, \{0\}), (x_4, \emptyset, \{0\})\}$ *is a type-0 extraction set, and* (S, v_1) *is an extraction pair.*

Problem 1 (Threshold Logic Extraction). *Given a TLN, the* threshold logic extraction problem *seeks to find common decomposition TLFs for logic sharing among the TLGs such that the total number of interconnections in the TLN is minimized while the function of the revised TLN remains unchanged.*

4.2 Extraction Flow

To achieve a high-quality solution efficiently, we propose an extraction flow shown in Figure 2. In the flow, an input circuit, not necessarily a TLN, is first synthesized into a TLN through existing TL synthesis tools and then the weights and threshold values of the TLGs are minimized. After the pre-processing, the core of our extraction flow includes three main stages: 1) extraction pair initialization, 2) iterative TLG extration, 3) and TLG post-processing, to be detailed in the following.

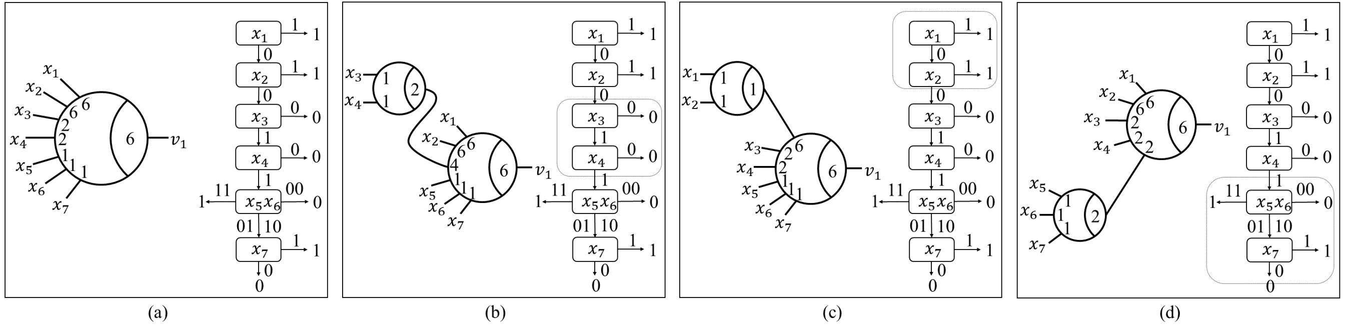

Figure 1: (a) TLG under decomposition and its GDL; (b) decomposition with contiguous type-0 nodes; (c) decomposition with contiguous type-1 nodes; (d) decomposition with suffix nodes.

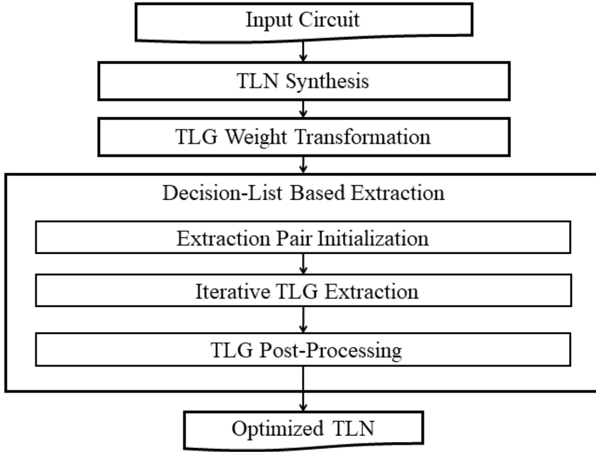

Figure 2: Computation flow of threshold logic extraction.

4.3 Extraction Pair Initialization

Given a TLN $G = (V, E)$ with all its TLGs $\in V$ converted in the positive unate form, the set of extraction pairs can be collected from the GDL, generated by algorithm *ConstructGDL*, of each TLG $v \in V$ by identifying the longest contiguous type-0 and type-1 nodes in the GDL.[1] Algorithm 2 sketches the steps.

Algorithm 2 *InitExtractionPairs*(G)

Input: A TLN $G = (V, E)$ with every TLG $v \in V$ in positive unate form.
Output: A set P of extraction pairs.

1: $P := \emptyset$;
2: **for each** TLG $v \in V$
3:     **for each** $S \in ExtractionSets(v)$
4:         $P := P \cup \{(S, v)\}$;
5: **return** P;

Example 5. *Consider the TLN in Figure 3(a). Assume variables x_1 and x_2 are inverted for TLG v_1 to be in positive unate form. For the TLN, the extraction pairs $(\{\neg x_1, x_3, x_4, x_5\}, v_1)$, $(\{x_3, x_4, x_5\}, v_2)$ and $(\{x_3, x_4, x_5\}, v_3)$ can be derived.*

4.4 Iterative TLG Extraction

Given a TLN $G = (V, E)$, its corresponding set P of extraction pairs, and an upper bound N_{lmax} on logic level, we perform iterative extraction to reduce the interconnect complexity of G while maintaining its logic level within N_{lmax}. The procedure is sketched in Algorithm 3. In Line 1, the extraction pairs P are sorted such that TLGs with larger extraction sets will be extracted first. Because larger extraction sets may potentially contribute to larger common extraction subsets, the procedure *ExtractVertex* is more likely to achieve greater interconnect reduction through vertex extraction. In each iteration of the while-loop in Lines 2-11, we scan through all extraction pairs in P and check if the extract operation can be applied. For each extraction pair $(S, v) \in P$, if v is on a critical path, the pair (S, v) is removed from P as an extraction on v will increase the logic level of G by one. Otherwise, a new vertex is extracted by *ExtractVertex*, and G and extraction pairs are updated.

Algorithm 3 *ExtractNtk*(G, P, N_{lmax})

Input: A TLN $G = (V, E)$, extraction pairs P, and level bound N_{lmax}.
Output: A TLN $G' = (V', E')$ after extraction.

1: sort the pairs $(S, v) \in P$ in a descending order of $|S|$;
2: $G' := G$;
3: **while** $P \neq \emptyset$
4:     $P' := \emptyset$;
5:     **for each** $(S, v) \in P$
6:         **if** $v \in CriticalPath(G')$ and $level(G') \geq N_{lmax}$
7:             $P := P \setminus \{(S, v)\}$;
8:         **else**
9:             $(G', P_E) := ExtractVertex(G', (S, v), P)$;
10:             $P' := P' \cup P_E$;
11:     sort the pairs $(S, v) \in P'$ in descending order of $|S|$;
12:     $P := P'$;
13: **return** G';

The function *ExtractVertex* identifies the largest *common extraction subset*, i.e., a subset shared by at least two extraction sets, based on the selected extraction set to form a new TLG, and adds corresponding new extraction pairs. Given a TLN G', a target extraction pair (S, v) and a set of extraction pairs P, the algorithm *ExtractVertex* finds the largest common extraction subset S_{max} shared by at least two extraction sets. By checking all the extraction sets in P having the same type with S, it obtains S_{max} by set intersection. It then creates a new TLG v' in G' based on S_{max}, and updates the TLGs where v' is extracted from according to the derivation described in Section 3.2.

Example 6. *Figure 3(a) shows a TLN with three TLGs and their respective GDLs, where the maximum common extraction subset is identified and indicated by the dash round rectangles. In Figure 3(b), a new threshold gate v_4 is constructed by applying the* ExtractVertex *operation. The fanout gates of v_4 are rebuilt, and then interconnections*

[1] In our current implementation we do not exploit suffix nodes for extraction, i.e., case 3 of Theorem 2, as the likelihood of having equivalent suffix TLFs from different TLGs can be low.

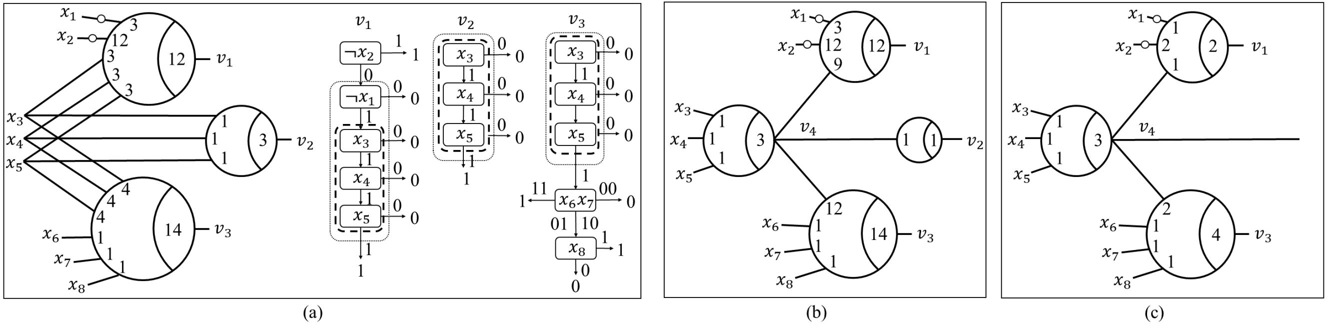

Figure 3: (a) TLN under extraction and GDLs of the vertices; (b) TLN after extraction; (c) TLN simplified by post-processing.

between v_4 and these gates are done. As a result, the interconnect compelxity is reduced from 17 in Figure 3(a) to 13 in Figure 3(c).

4.5 TLG Post-Processing

In this step, we iterate through each TLG and perform the following processing: For a one-input TLG, if the weight is no smaller than the threshold value, it is equivalent to a buffer and can be replaced with a wire. Otherwise, it is equivalent to a constant 0 gate and the circuit can be simplified with constant propagation. For a multiple-input gate, we minimize its weight/threshold values with ILP-based minimization [14].

Example 7. *Figure 3(b) and (c) show the effect of post-processing. Because TLG v_2 is equivalent to a buffer and is removed from the TLN. Furthermore, the weight/threshold values of v_1 and v_3 are minimized.*

5 EXPERIMENTAL RESULTS

The proposed extraction algorithm was implemented in the ABC environment [2] using the C programming language. All experiments were conducted on a Linux workstation with an Intel Xeon 2.1GHz CPU with 128GB memory. Test cases were selected from ISCAS, ITC, and MCNC benchmark suits. We prepared the initial threshold logic circuits using the state-of-the-art threshold logic synthesis approach [16] implemented in ABC. The circuits with less than 200 threshold gates after synthesis of [16] were not included in our experiments.

Gate count and logic level are common metrics in conventional TLN implementation such as LUT-based design. However, gate count is not the dominant factors of TLN area estimation in implementations based on emerging technologies. For example, the sum of total interconnections is the dominant factor of spintronic-based TLN area estimation rather than the total gate count. To evaluate the solution quality of our proposed algorithm in terms of the cost of emerging technologies as mentioned in Section 2, we defined two cost functions for area estimation. For spintronic- and memristor-based TLN implementation, the main factor affecting circuit area A is the total number of interconnections. Hence we have

$$C_{wire} = \sum_{v} |FI(v)|, \tag{2}$$

where $|FI(v)|$ denotes the number of fanins of threshold gate v in the TLN. For RTD-based implementation, the dominant factor of circuit area is the magnitude of weight and threshold values. Hence we have

$$C_{RTD} = \sum_{v} \sum_{i=1}^{|FI(v)|} \alpha \cdot (|w_{vi}| + |T_v|), \tag{3}$$

where α (setting to 1 in the experiments) is the unit area of an RTD, w_{vi} is the weight of the ith fanin of gate v, and T_v is the threshold value of gate v.

Table 1 shows the results of our extraction algorithm applied on circuits synthesized with [16] under delay and area optimization options. Columns 2 and 3 show the numbers of inputs and outputs of the circuits, respectively; Columns 4, 5, 6, and 7 (resp. 13, 14, 15, and 16) report the number of TLGs, number of logic levels, interconnect cost, and RTD implementation cost of the TLNs synthesized by [16] under delay (resp. area) optimization with weights and threshold values minimized by the method in [14]; Columns 8, 9, 10, 11, and 12 (resp. 17, 18, 19, 20, and 21) list the number of TLGs, number of logic levels, interconnect cost, and RTD implementation cost, and CPU time of entire computation, including the GDL construction and extraction operation, performed on TLNs synthesized by [16] under delay (resp. area) optimization.

According to Table 1, for circuits optimized with delay minimization our algorithm achieved an average of 10% reduction on C_{wire} and 14% reduction on C_{RTD} in the cost of 18% increase in logic level. On the other hand, for circuits optimized with area minimization our algorithm achieved an average of 8% reduction on C_{wire} and 13% reduction on C_{RTD} in the cost of 6% increase in logic level. Notice that there are more significant C_{wire} and C_{RTD} reductions on circuits optimized for delay compared to those optimized for area. This fact is understandable as area-driven synthesis may have exploited potential logic sharing extensively. On the other hand, the higher percentage of level increase in delay-optimized, compared to area-optimized, circuits is much due to the original small level values; a slight level increase contributes to an amplified percentage increase. Note that although extraction may potentially increase logic level, the reduced interconnect complexity may nullify or alleviate circuit performance degradation. For the run time, all extraction computations were done within 16 seconds (occurred in the case b19) while most instances were solved almost instantly. The experimental results show that the extraction computation can be effective and efficient.

As the extraction operation reduces interconnect complexity in the cost of increasing logic level, it sometimes may be desirable to explore the trade-off between interconnect complexity and logic level. To achieve such a trade-off, we may specify an upper bound on logic level such that the gates on critical paths with their depths reaching the bound will be ignored in the iterative extraction scheme. We performed a case study on circuit b15 to plot its trade-off curve shown in Figure 4, where interconnection cost C_{wire} and RTD cost C_{RTD} are plotted as functions of the number logic levels N_L. As observed, by setting the upper bound of logic level to about 20, the

Table 1: Results of threshold logic synthesis with and without extraction.

Benchmarks Profile			Delay-Driven									Area-Driven								
			Before Extraction [16]				After Extraction					Before Extraction [16]				After Extraction				
circuit	#pi	#po	N_G	N_L	C_{wire}	C_{RTD}	N_G	N_L	C_{wire}	C_{RTD}	T (s)	N_G	N_L	C_{wire}	C_{RTD}	N_G	N_L	C_{wire}	C_{RTD}	T (s)
c3540	50	22	465	13	2036	6883	495(1.06)	15(1.15)	1717(0.84)	5367(0.78)	0.00	473	23	1858	5902	473(1.00)	24(1.04)	1518(0.82)	4307(0.73)	0.00
c5315	178	123	732	10	2463	7574	741(1.01)	11(1.10)	2444(0.99)	7366(0.97)	0.00	738	19	2265	6283	744(1.01)	19(1.00)	2257(1.00)	6233(0.99)	0.00
c6288	32	32	1424	29	4848	13561	1478(1.04)	33(1.14)	4768(0.98)	12610(0.93)	0.01	1190	60	4027	10168	1203(1.01)	60(1.00)	4006(0.99)	9915(0.98)	0.01
c7552	207	108	950	10	3013	8255	975(1.03)	12(1.20)	2976(0.99)	7811(0.95)	0.00	938	17	2763	7520	953(1.02)	17(1.00)	2743(0.99)	7246(0.96)	0.00
s5378	35	49	578	5	2087	6056	610(1.06)	6(1.20)	2002(0.96)	5521(0.91)	0.00	584	11	1823	4762	613(1.05)	11(1.00)	1771(0.97)	4441(0.93)	0.00
s9234.1	36	39	744	7	2553	7107	774(1.04)	8(1.14)	2426(0.95)	6573(0.92)	0.00	727	16	2432	6569	761(1.05)	16(1.00)	2293(0.94)	6023(0.92)	0.00
s13207	31	121	1257	8	4708	11754	1343(1.07)	9(1.13)	3877(0.82)	9327(0.79)	0.00	1212	14	4654	12104	1326(1.9)	17(1.21)	3760(0.81)	9154(0.76)	0.00
s15850	14	87	1630	10	6097	17158	1715(1.05)	12(1.20)	5207(0.85)	13302(0.78)	0.01	1538	25	5400	15089	1630(1.06)	26(1.04)	4734(0.88)	11523(0.76)	0.00
s35932	35	320	5878	5	15506	37994	5878(1.00)	5(1.00)	15506(1.00)	37994(1.00)	0.03	5246	6	13826	35714	5246(1.00)	6(1.00)	13826(1.00)	35714(1.00)	0.02
s38417	28	106	4857	8	16143	44330	5060(1.04)	9(1.13)	14512(0.90)	37331(0.84)	0.04	4856	16	15180	38451	5013(1.03)	17(1.06)	13771(0.91)	33218(0.86)	0.04
s38584	12	278	4391	8	18167	56137	4868(1.11)	9(1.13)	16719(0.92)	46086(0.82)	0.10	4364	17	18351	57028	4873(1.12)	18(1.06)	16615(0.91)	45483(0.80)	0.09
b04	11	8	270	9	879	2387	275(1.03)	10(1.11)	836(0.95)	2202(0.92)	0.00	251	17	793	2087	256(1.02)	17(1.00)	753(0.95)	1912(0.92)	0.00
b12	5	6	490	5	2494	7193	610(1.24)	7(1.40)	1895(0.76)	4859(0.68)	0.00	443	12	2258	6838	551(1.24)	13(1.08)	1691(0.75)	4485(0.66)	0.00
b14	32	54	2680	13	9282	27810	2770(1.03)	17(1.31)	8964(0.97)	25279(0.91)	0.02	2593	44	7953	21226	2658(1.03)	49(1.11)	7720(0.97)	19531(0.92)	0.01
b15	36	70	4077	16	15870	47274	4291(1.05)	20(1.25)	13725(0.86)	37613(0.80)	0.06	4022	44	15178	41107	4206(1.06)	47(1.07)	12695(0.84)	33201(0.81)	0.04
b17	37	97	13009	22	47685	133990	13695(1.05)	25(1.14)	42072(0.88)	110554(0.83)	0.55	12803	53	45066	121455	13369(1.04)	54(1.02)	39923(0.89)	101991(0.84)	0.44
b18	37	23	36370	43	131849	367445	38271(1.05)	50(1.16)	118096(0.90)	311308(0.85)	4.55	35517	86	127349	346294	37446(1.05)	88(1.02)	113375(0.89)	291194(0.84)	3.71
b19	24	27	72362	46	262169	726624	76255(1.05)	52(1.13)	234636(0.89)	614586(0.85)	15.15	71638	93	253519	688531	75197(1.05)	94(1.01)	226551(0.89)	583424(0.85)	13.94
b20	32	22	5660	15	19657	61786	5863(1.04)	19(1.27)	18945(0.96)	54446(0.88)	0.10	5470	44	16249	41999	5524(1.01)	49(1.11)	16059(0.99)	40939(0.97)	0.06
b21	32	22	5660	16	19131	58198	5848(1.03)	21(1.31)	18505(0.97)	51722(0.89)	0.11	5515	44	16438	42230	5582(1.01)	51(1.16)	16254(0.99)	41140(0.97)	0.07
b22	32	22	8429	16	27774	81010	8581(1.02)	19(1.19)	27263(0.98)	77248(0.95)	0.22	8305	44	24640	63154	8404(1.01)	45(1.02)	24321(0.99)	61301(0.97)	0.15
des	256	245	1563	6	5817	18185	1610(1.03)	7(1.17)	5654(0.97)	17479(0.96)	0.01	1554	12	5005	15587	1565(1.01)	12(1.00)	4989(1.00)	15379(0.99)	0.01
pair	173	137	595	5	2387	7688	624(1.05)	6(1.20)	2314(0.97)	7235(0.94)	0.00	622	15	2112	5988	652(1.05)	15(1.00)	2062(0.98)	5494(0.92)	0.00
apex6	135	99	297	4	1278	4162	320(1.08)	5(1.25)	1163(0.91)	3487(0.84)	0.00	291	8	1236	3989	316(1.09)	10(1.25)	1108(0.90)	3185(0.80)	0.00
alu4	14	8	283	10	1228	4367	302(1.07)	12(1.20)	1176(0.96)	3926(0.90)	0.00	267	20	1095	3795	289(1.08)	20(1.00)	1041(0.95)	3375(0.89)	0.00
dalu	75	16	520	7	2101	7498	551(1.06)	9(1.29)	1837(0.87)	6302(0.84)	0.00	526	17	1886	6425	539(1.02)	17(1.00)	1652(0.88)	5291(0.82)	0.00
x3	135	99	288	4	1150	3472	308(1.07)	4(1.00)	1030(0.90)	3004(0.87)	0.00	290	7	1114	3608	311(1.07)	8(1.14)	1015(0.91)	3041(0.84)	0.00
i6	138	67	200	2	847	2284	200(1.00)	2(1.00)	847(1.00)	2284(1.00)	0.00	200	3	771	2152	205(1.03)	4(1.33)	658(0.85)	1828(0.85)	0.00
i7	199	67	263	2	991	3022	263(1.00)	2(1.00)	991(1.00)	3022(1.00)	0.00	235	3	871	2622	236(1.00)	3(1.00)	870(1.00)	2620(1.00)	0.00
i8	133	81	426	3	2345	8384	444(1.04)	5(1.67)	1566(0.67)	4631(0.55)	0.00	378	7	1448	4293	401(1.06)	8(1.14)	1296(0.90)	3740(0.87)	0.00
i9	88	63	274	3	1746	6369	277(1.01)	4(1.33)	1121(0.64)	4297(0.67)	0.00	226	6	897	3050	234(1.04)	8(1.33)	799(0.89)	2768(0.91)	0.00
i10	257	224	902	10	3361	9643	973(1.08)	13(1.30)	3126(0.93)	8341(0.86)	0.00	843	26	3000	8530	903(1.07)	28(1.08)	2822(0.94)	7518(0.88)	0.00
geomean			1.00	1.00	1.00	1.00	1.05	1.18	0.90	0.86		1.00	1.00	1.00	1.00	1.04	1.06	0.92	0.87	

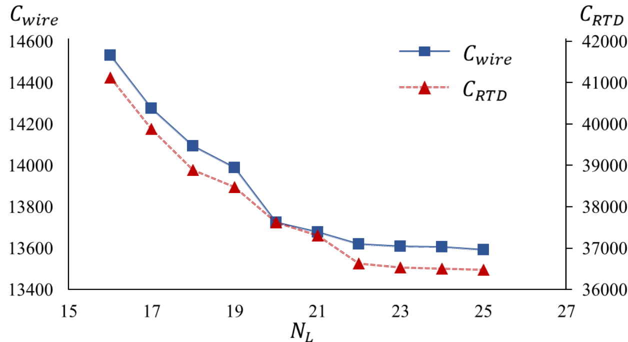

Figure 4: Trade-off between interconnect/RTD cost and logic level for circuit b15.

best trade-off between the cost value and the circuit level can be achieved for circuit b15.

6 CONCLUSIONS

This work has laid the foundations of disjoint-support decomposition of threshold logic functions and proposed an extraction algorithm for interconnect minimization of threshold logic networks. Experimental results has demonstrated the efficiency and effectiveness of the proposed method in reducing interconnect/RTD implementation costs and in the trade-off between interconnect complexity and circuit depth. As interconnect costs become a dominating factor to circuit area and delay, the extraction operation may play a key role for threshold logic synthesis.

ACKNOWLEDGMENT

The authors thank Chia-Chih Chi, Siang-Yun Lee, and Nian-Ze Lee for helping the experiments. This work was supported in part by the Ministry of Science and Technology of Taiwan under grants 105-2221-E-002-196-MY3, 105-2923-E-002-016-MY3, and 106-2923-E-002-002-MY3.

REFERENCES

[1] V. Beiu, J. M. Quintana, and M. J. Avedillo. Vlsi implementations of threshold logic-a comprehensive survey. *IEEE Tran. on Neural Networks*, 14(5):1217–1243, 2003.

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification.* https://people.eecs.berkeley.edu/~alanmi/abc/.

[3] E. P. Blair and C. S. Lent. Quantum-dot cellular automata: An architecture for molecular computing. In *Proc. of SISPAD*, pages 14–18, 2003.

[4] Y.-C. Chen, R. Wang, and Y.-P. Chang. Fast synthesis of threshold logic networks with optimization. In *Proc. of ASP-DAC*, pages 486–491, 2016.

[5] C.-C. Chi and J.-H. R. Jiang. Logic synthesis of binarized neural networks for efficient circuit implementation. In *Proc. of ICCAD*, pages 84:1–84:7, 2018.

[6] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. In *arXiv e-print:1602.02830*, 2016.

[7] Y. Crama and P. L. Hammer. *Boolean Models and Methods in Mathematics, Computer Science, and Engineering.* 2010.

[8] D. Fan, M. Sharad, and K. Roy. Design and synthesis of ultralow energy spin-memristor threshold logic. *IEEE Tran. on Nanotechnology*, 13(3):574–583, 2014.

[9] L. Gao, F. Alibart, and D. B. Strukov. Programmable cmos/memristor threshold logic. *IEEE Tran. on Nanotechnology*, 5(2):115–119, 2013.

[10] T. Gowda, S. Leshner, S. Vrudhula, and G. Konjevod. Synthesis of threshold logic circuits using tree matching. In *Proc. of ECCTD*, pages 850–853, 2007.

[11] P.-Y. Kuo, C.-Y. Wang, and C.-Y. Huang. On rewiring and simplification for canonicity in threshold logic circuits. In *Proc. of ICCAD*, pages 396–403, 2011.

[12] C. Lageweg, S. Cotofana, and S. Vassiliadis. A linear threshold gate implementation in single electron technology. In *Proceedings IEEE Computer Society Workshop on VLSI 2001. Emerging Technologies for VLSI Systems*, pages 93–98, 2001.

[13] N.-Z. Lee, H.-Y. Kuo, Y.-H. Lai, and J.-H. R. Jiang. Analytic approaches to the collapse operation and equivalence verification of threshold logic circuits. In *Proc. of ICCAD*, pages 1–8, 2016.

[14] S.-Y. Lee, N.-Z. Lee, and J.-H. R. Jiang. Canonicalization of threshold logic representation and its applications. In *Proc. of ICCAD*, pages 85:1–85:8, 2018.

[15] S. Muroga. *Threhsold logic and its applications.* 1971.

[16] A. Neutzling, J. M. Matos, A. I. Reis, R. P. Ribas, and A. Mishchenko. Threshold logic synthesis based on cut pruning. In *Proc. of ICCAD*, pages 494–499, 2015.

[17] A. Palaniswamy and S. Tragoudas. Improved threshold logic synthesis using implicant-implicit algorithms. *ACM Journal on Emerging Technologies in Computing Systems*, 10(3):21:1–21:32, 2014.

[18] R. L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

[19] J. Subirats, J. Jerez, and L. Franco. A new decomposition algorithm for threshold synthesis and generalization of boolean functions. *IEEE Tran. on Circuits and Systems I: Regular Papers*, 55(10):3188–3196, 2008.

[20] R. Zhang, P. Gupta, L. Zhong, and N. K. Jha. Threshold network synthesis and optimization and its application to nanotechnologies. *IEEE Tran. on CAD*, 24(1):107–118, 2004.