Threshold Logic Synthesis Based on Cut Pruning

Augusto Neutzling, Jody Maick Matos,
Andre I. Reis, Renato P. Ribas
Institute of Informatics
Federal University of Rio Grande do Sul, Brazil
Emails: {ansilva,jmamatos,rpribas,andreis}@inf.ufrgs.br

Alan Mishchenko
Department of EECS
University of California, Berkeley
Email: alanmi@eecs.berkeley.edu

***Abstract*—This paper presents a novel approach to synthesize circuits based on threshold logic gates (TLGs). Emerging technologies, such as memristors, spintronics devices and tunneling diodes, are able to build this class of gates efficiently. For this reason, threshold logic is a promising alternative to conventional CMOS logic. The proposed approach is based on pruning non-threshold-logic cuts in order to limit the search space during technology mapping. As a result, both the number of TLGs and logic depth of the synthesized circuits are reduced. Experimental results have shown that, compared to the state-of-the-art methods, the TLG count is reduced by 8% and logic depth is reduced by 46%.**

***Index Terms*—Digital circuit, threshold logic synthesis, emerging technologies, technology mapping.**

I. Introduction

The limits of MOS transistor scaling have motivated the investigation of new alternative devices, such as memristors, spintronics, resonant tunneling devices (RTD), quantum cellular automata (QCA) and single electron transistor (SET) [1]–[6]. It has been shown that threshold logic gates (TLGs) are more suitable to build digital integrated circuits (ICs) in these emerging technologies, compared to the standard AND/OR-based CMOS design style [3], [6]. This motivates development of EDA tools, which can synthesize and optimize TLG-based ICs.

Algorithms addressing threshold logic synthesis have been presented in recent years [6]–[10]. However, all these previous methods use SIS tool [11] to generate the initial set of single-output Boolean networks, restricting each network fanin to six. The threshold synthesis starts after the circuit has already been covered by SIS without considering threshold logic. Once the circuit is covered, the previous approaches locally perform a threshold network synthesis for each independent single-output network, aiming to cover the circuit using only TLGs.

This paper presents a novel threshold logic synthesis approach that is based on a new synthesis flow, combining both TLG count and logic depth optimization. In particular, the proposed flow relies on pruning the non-threshold cuts before performing the circuit covering. By doing this, the resulting mapped circuit is composed by only TLFs, which can be implemented into single TLGs. This cut-pruning-based method is implemented in ABC [12] as part of its multi-objective technology mapper. When compared against the state-of-the-art methods, similar results have been obtained in terms of TLG count (8% reduction), with half of logic depth on critical paths (46% reduction). We compare the obtained results against previous approaches by presenting experiments carried out over both ACM/SIGDA (a.k.a. MCNC) [13] and ISCAS'85 benchmarks. We also demonstrate the scalability of the proposed method by synthesizing 12 large OpenCore benchmarks [14].

The main contributions of this paper are the following:

1) Simplicity: the proposed threshold logic synthesis flow is significantly simpler than previous approaches;
2) Circuit-Level Scalability: the proposed method is efficiently based on cut pruning, scaling to large benchmark circuits;
3) Gate-Level Scalability: the proposed threshold logic synthesis method is capable of synthesizing circuits using TLFs up to 9 inputs, while other methods are restricted to 6 input gates;
4) Quality of Results: best in class results compared to previous methods and new OpenCore [14] results in terms of both TLG count and logic depth can be used as a reference in further publications.

The rest of the paper is organized as follows. Section II presents the related works. In Section III, some fundamentals are briefly reviewed for a better understanding of the proposed approach. Section IV describes both the technology mapping for field-programmable gate arrays (FPGAs), which bases the synthesis flow proposed herein, and the proposed approach to synthesize TLG-based circuits. Section V provides the experimental results, whereas the conclusions are outlined in Section VI.

II. Related Works

Previously proposed related methods start from a given single output Boolean network and map the network in one or more TLGs. Circuits with multiple outputs are divided into single-output networks.

In [6], Zhang *et al.* propose the recursive partition of non-threshold functions to merge the nodes respecting fanin restrictions. Unfortunately, the quality of results is very sensitive to the initial Boolean network description.

Further, a method based on truth table descriptions was presented by Subirats *et al.*, in [7]. Subirats' algorithm computes an ordering of variables by using information of on-set and off-set, performing Shannon decomposition up to find threshold logic functions (TLFs). However, this approach produces two-level threshold networks without fanin restriction, which is more suitable for neural networks than for digital IC design.

In [8], Gowda *et al.* apply a factorized tree method to generate the network of threshold gates. The method recursively breaks the given initial expression tree into sub expressions, identifying sub-trees which represent TLFs and assigning the input weights. It is more appropriate for IC synthesis using several TLGs, but it is very time consuming and presents a strong dependence to the initial expression structure, including the ordering of the initial tree. The method proposed by Palaniswamy and Tragoudas, in [9], presents some improvements to Gowda's method [8].

Finally, the method proposed by Neutzling *et al.*, in [10], is based on a TLG association method through the principle called functional composition (FC), based on dynamic programming. The algorithm associates simpler sub-solutions, with known costs, in order to produce a final solution with minimum cost. This method presents better results in terms of TLG count when compared against previous approaches. However, it does not present significant optimization in terms of logic depth.

Kuo *et al.* and Lin *et al.* proposed in [15], [16], respectively, methods for TLG circuits rewiring. The methods start from an already synthesized TLG network and use both weight-inputs and threshold value summation as cost functions, differently from the mentioned works.

978-1-4673-8388-2/15/$31.00 ©2015 IEEE

III. PRELIMINARIES

A. Boolean Network

A Boolean network is a directed acyclic graph (DAG) where nodes correspond to logic gates and directed edges represent the wires connecting the gates. It is assumed that each node has a unique ID (integer number).

A *fanin* (*fanout*) cone of node n is a subset of all nodes of the network reachable through the *fanin* (*fanout*) edges from the given node.

A node n has zero or more *fanins* (nodes driving n) and zero or more *fanouts* (nodes driven by n). The primary inputs (PIs) are nodes without *fanins*, whereas the primary outputs (POs) are a subset of nodes from the network connecting it to the environment.

B. AIG

And-inverter graph (AIG) is a specific type of Boolean network where each node has either zero incoming edges – PIs – or two incoming edges – AND nodes. Each edge can be complemented or not. Some nodes are marked as POs.

C. Structural Cuts

A *cut* C of a node n is a set of nodes of the network, called *leaves* of the cut, such that every path between a PI and n contains a node in C. A cut of n is irredundant if no subset on it is a cut. A K-feasible cut is an irredundant cut containing K or fewer nodes.

Node n is called the root of cut C. The cut size is the number of its leaves. A trivial cut is the node itself. A local function of an AIG node n, denoted by $f_n(x)$, is a Boolean function of the logic cone rooted in n and expressed in terms of the leaves, x, of a cut of n.

Cut enumeration is a technique used by a cut-based technology mapper to perform cut computation using dynamic programming, starting from PIs and ending at POs [17], [18].

D. Threshold Logic Function

Threshold logic functions are a subset of Boolean functions which respects the following operation principle. Each input has a specific weight and the gate has a threshold value. If the sum of ON input weights (inputs with value equal to 1) is equal or greater than the threshold value, the resulting function value is equal to 1. Otherwise, the resulting function value is 0. This operating behavior can be expressed as follows [19]:

$$f = \begin{cases} 1, & \textit{if} \sum_{i=1}^{n} W_i X_i \geq T \\ 0, & \textit{otherwise} \end{cases} \tag{1}$$

where X_i represents each input value $\{0, 1\}$, W_i is the weight of each input, and T is the function threshold value. A TLF can also be called a 'linearly separable' function.

E. Threshold Logic Gates

Threshold logic gate is an electronic circuit that implements a TLF. A TLG is completely represented by a compact vector $[w_1,w_2,\ldots,w_n;T]$, where $w_1,w_2,\ldots,w_n$ are the input weights and T is the function threshold value. For instance, the corresponding TLG of the given functions $f = x_1x_2x_3$ and $g = x_1 + x_2 + x_3$ are [1,1,1;3] and [1,1,1;1], respectively.

TLGs can implement complex functions. For example, TLG [4,3,3,1,1;7] implements $f = x_1x_2 + x_1x_3 + x_2x_3x_4 + x_2x_3x_5$. Using larger threshold functions has the potential benefit of reducing the total number of gates needed to implement digital circuits.

Several implementations of TLGs have been proposed for both CMOS and new nanometric technologies. A survey with more than 50 TLG implementations was presented by Beiu *et al.*, in [5]. More recent implementations based on memristors [2], spintronics devices [3], [4], and RTDs [6] have also been proposed.

F. Threshold Logic Identification

Although some complex functions are TLFs, there exist some simple functions which are not TLFs. For example, function $h = x_1x_2 + x_3x_4$ cannot be implemented in a single TLG. The threshold logic identification process corresponds to the task responsible to determine if a Boolean function is TLF (or not), and compute the input weights and the gate threshold value. In this work, we have adopted the identification process presented in [20], instead of the integer linear programming based algorithms applied in previous works [6]–[9]. This is mainly due to fast runtime and good quality of results obtained.

IV. THRESHOLD LOGIC SYNTHESIS

Threshold logic has been revisited as a promising logic style when considering emerging technologies, such as memristors, spintronics devices and tunneling diodes. However, state-of-the-art threshold logic synthesis tools are not based on specific technologies when defining cost functions. Works in [6]–[10] adopt two main design costs: (1) TLG count on mapped netlists, when looking at the circuit area; and (2) logic depth on critical paths, when looking at the circuit delay. Such cost functions are similar to those used for LUT-based FPGA synthesis, which adopt LUT count and logic depth as cost functions.

Technology mapping for FPGAs is a well-known research field and a number of techniques are consolidated on the literature. Such methods are able to achieve impressive solutions on area recovery with near-optimum logic depth on critical paths. Due to these reasons, this work proposes a novel synthesis flow which adapts the FPGA technology mapper in ABC [12] to provide a mapped netlist composed of only threshold logic functions. The following subsections present a brief overview of FPGA technology mapping methods, the threshold synthesis flow we propose in this work (which is mainly based on the FPGA technology mapping flow), and a comparison of this flow against previous approaches.

A. Technology Mapping for FPGAs

The technology mapping process transforms a technology-independent logic network, named subject graph, into a network of logic nodes. For FPGAs, each logic node is represented using a K-input LUT implementing any Boolean function up to K inputs. The subject graph is often represented as an AIG.

The delay of an FPGA circuit is determined by two factors: (1) the delay in K-LUTs; and (2) the interconnection delay. Each K-LUT has a constant delay independent of the function it implements (the access time of a K-LUT). The interconnection delay is dominated by the physical configuration of the FPGA, which is not available during the synthesis task. Thus, state-of-the-art FPGA technology mappers assume that each edge in the mapping solution has a constant delay. Due to these reasons, the circuit delay is commonly estimated by the logic depth and the circuit area is determined by the number of K-LUTs of the mapping solution.

State-of-the-art FPGA technology mappers [17], [21], [22] produce near-optimum logic depth while minimizing the number of LUTs in the resulting network. A typical procedure consists of the following steps:

1) Near-optimum delay mapping
   - Compute arrival time at each node by computing the depth of all priority cuts and choosing the best one
2) Area recovery
   - Perform area recovery using several heuristics (for example, area flow and exact local area [14])

3) Choose the resulting cover

The depth-optimality requires computation of all cuts in delay mapping, which is not scalable (the number of K-cuts in a network with n nodes is $O(nk)$). The concept of priority cuts is based on, instead of computing all K-feasible cuts at each node, it computes a small number, c, of K-feasible cuts at each node (typically, $4 \geq c \geq 8$). The priority cuts computation does not guarantee the depth-optimality. However, the depth is optimal in 95% of the cases [23]. This one-pass depth-oriented mapping substantially increases the number of LUTs, which is brought down on area recovery step. For a detailed description of these steps, we refer the reader to [21] and [17].

Since the area and delay cost functions for threshold logic synthesis are, respectively, TLG count and logic depth, it is straightforward to relate them with FPGA technology mapping. The following subsection presents the technology mapping approach we propose for threshold logic, which is mainly based on the FPGA technology mapping flow.

B. Technology Mapping for Threshold Logic

Threshold logic synthesis for a given circuit finds an optimized netlist containing only TLFs. Notice that each TLF derives a single TLG in the mapped netlist. Fig. 1 illustrates the threshold synthesis flow used in previous works and the flow we propose in this paper. State-of-the-art threshold logic synthesis tools are based on identifying TLFs only after the circuit was covered by single-output Boolean networks ("Traditional Covering" in Fig. 1). Then, to achieve a netlist comprised only by TLGs, they propose to synthesize threshold networks to replace the non-TLFs in the covering.

The main advantage of the synthesis flow proposed herein is to identify TLFs before the circuit covering. By doing this, we are able to discard those non-TLFs and to perform a circuit covering by using only TLFs, what derives a mapped netlist composed only by TLGs. This improvement allows us to explore the multi-objective FPGA technology mapper, described in Section IV-A. In order to achieve these claims, we propose to pre-compute Boolean functions of cuts obtained from the AIG, identify TLFs over this set of computed cuts, and to discard the non-TLFs by a cut pruning approach.

An efficient method to pre-compute Boolean functions of cuts in a design (or a suite of designs) relies on fast algorithms to compute NPN-canonical forms and compactly store them. We adapt the DSD manager data-structure [24], which stores representatives of each NPN class as a shared tree. The DSD manager provides a convenient way of checking functional properties, such as symmetry, unateness, and decomposability, and, in our specific case, identifying TLFs over the computed cuts.

Algorithm 1: Pseudo-code of the proposed approach.

Input: circuit description
Output: TLG-based netlist
1 extract AIG from the input circuit;
2 compute cuts and populate the DSD manager with pre-computed cut functions;
3 identify and mark TLFs among these functions;
4 discard cuts, which have non-threshold functions, during technology mapping;
5 derive final mapping using only TLFs;
6 return the resulting mapped circuit;

The method proposed herein starts by computing priority cuts in the input AIG, pre-computing Boolean functions from these cuts, and storing them in the DSD manager. Since the k-cuts are pre-computed in the input design, it is possible to mark a matchable/unmatchable label for each cut and indicate these marks to the technology mapper. Thus, we filter those pre-computed functions and indicate only TLFs to be matched. The TLFs are identified by applying the threshold logic identification method presented in [20]. Once the identification is performed, those non-threshold cuts are labeled as unmatchable and not allowed to be selected as the best cut of a node while mapping the design.

Notice that the proposed method always find a TLF-only cover, since the trivial cut (the node itself) is present in every set of c cuts at each node [17], [21]. Once we propose to use AIGs as subject graphs, the entire design is already decomposed into AND nodes, which are TLFs.

When the final mapping is derived, a subset of best cuts is selected and, since they are always matchable, the resulting mapping only contains the cuts that can be expressed using TLFs. The pseudo-code of the proposed approach is presented in Algorithm 1.

The complexity of mapping step is $O(Knc^2)$, dominated by the cut computation (linear in the size K of cuts and the number of circuit nodes n, and quadratic in the number of cuts c stored at each node). Matching of cut functions against threshold logic functions (TLFs) is performed in constant time for each cut (hash table lookup). The complexity of the pre-computation step is $O(K \cdot log(K)mm')$, being m and m' the number of primes related to the on-set and off-set of the candidate TLF, respectively. As TLFs are unate functions, the number of primes m (or m') is at most $\frac{K!}{\lfloor K/2 \rfloor ! \cdot \lceil K/2 \rceil !}$. Notice that these complexities are bearable for small K, up to 9 inputs. Additionally, pre-computation can be done only once to generate the candidate TLF set.

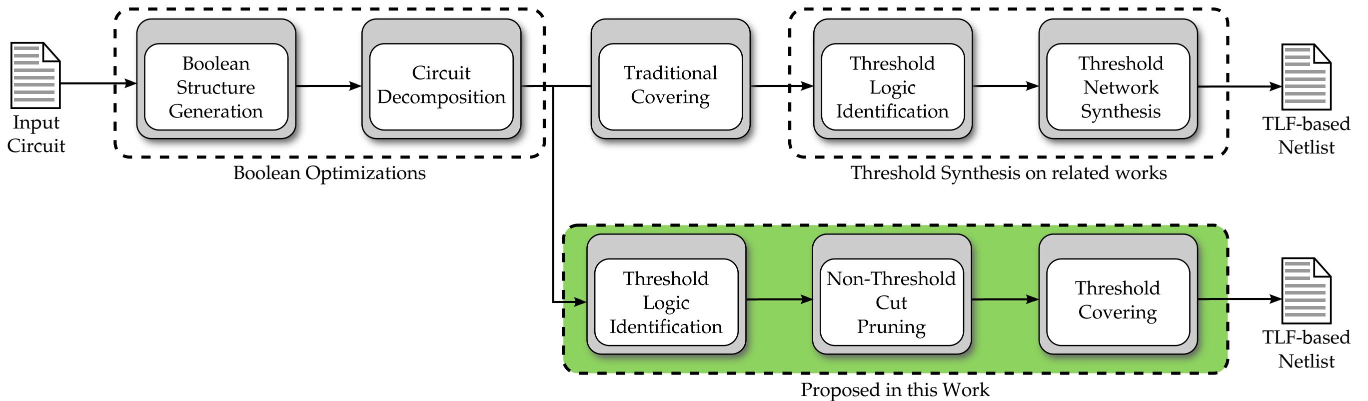

Fig. 1. Threshold logic synthesis flow proposed in this work.

V. Experimental Results

In order to validate the proposed threshold logic synthesis flow, experiments were carried out over different sets of benchmark circuits. The proposed approach is implemented in ABC [12] using C programing language and compiled with *gcc 4.7.2* compiler. The experiments were performed on a computer with Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz, 8Gb RAM.

For each benchmark circuit, we have pre-computed the set of NPN classes of 9-input functions by running the script (*&synch2; &if -n -K 9*), which was iterated three times per design. This script performs logic synthesis with choices (*&synch2*), computes 9-input cuts together with their Boolean functions and saves them in the DSD manager (enabled by switch "*-n*" in "*if*").

Once the DSD manager has been populated, we filter the pre-computed functions and prune those non-threshold cuts by using the script (*dsd_filter -t*). This script identifies the threshold logic functions by applying the method in [20] and selects only the TLFs used in mapping step (switch "*-t*"). Finally, we map the design using only matchable cuts iterating the following script three times (*&synch2; &if-k -K 9*). Each node in a resulting netlist represents a TLF due to the filtering step (enabled by switch "*-k*") and does not exceed 9 inputs (switch "*-K 9*").

In order to compare our results against the state-of-the-art approaches presented both by Neutzling *et al.* [10] and by Zhang *et al.* [6], ACM/SIGDA benchmarks (a.k.a. MCNC) [13] were synthesized. Table I shows the results obtained in terms of TLG count and circuit logic depth.

When comparing Neutzling *et al.* [10] to Zhang *et al.* [6] results, the former presents 54% reduction in TLG count and 30% reduction in logic depth. For this reason, Neutzling's results have been adopted as reference metric. The proposed method presents TLG count lesser or equal to the reference in 76% of benchmarks (around 8% reduction, on average). The logic depth is reduced in all benchmarks (around 44% reduction, on average). The execution time is around 1 second per circuit, on average

It is important to remark that previous approaches provide synthesized netlists comprised with TLFs up to six inputs. The reason firstly presented in [6] is that, when increasing the number of inputs, the percentage of functions that are threshold decreases drastically. This statement holds when considering the universe of all Boolean functions. However, it has been observed through the carried out experiments that 55% of the identified TLFs in MCNC (smaller) circuits have more than six inputs and about 83% in Opencore (larger) circuits.

We also synthesized the MCNC benchmark limiting the number of inputs up to six. In the proposed flow, changing fanin limitation from nine to six does not impact significantly in running time. When comparing the obtained results with limited fanin to Neutzling's approach, similar results have also been obtained in terms of TLG count (4% increasing), with 40% reduction in terms of logic depth.

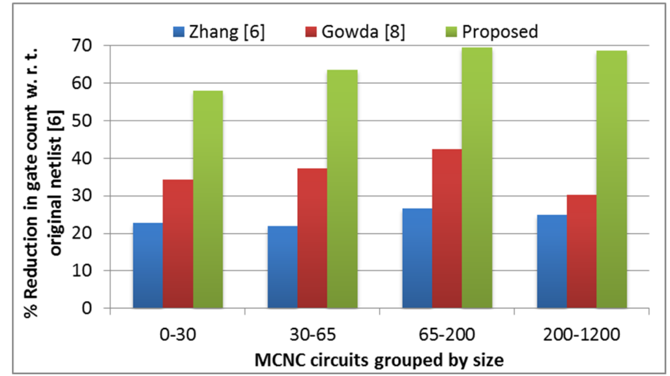

Fig. 2. Percentage gate count reduction in each approach, compared to the original netlist [6].

The results presented by Gowda *et al.*, in [8], show a reduction in threshold gate count compared to the results provided in [6]. However, in [8], the authors only compare the gate count and present the results for MCNC circuits grouped by number of gates. Fig. 2 shows the gate reduction of each approach, compared to the original netlist [6], which uses only the traditional OR and AND description.

The graphic shown in Fig. 2 demonstrates that the reduction in gate count is larger than the reduction presented both in [6] and in [8]. The proposed method has provided an average reduction of 50% in comparison to the original netlist, against a reduction of 23% and 34% obtained in [6] and in [8], respectively. The execution time is around 1 second per circuit, on average.

The most recent work, proposed by Palaniswamy and Tragoudas [9], presents two different improvements to the method proposed in [8], called BDM and ZDM. The results shown in Fig. 3 presents the TLG count reduction obtained both by the method proposed herein and the Palaniswamy's one. The ISCAS'85 set of benchmarks was synthesized for this experiment. The reference values are the results obtained by Gowda *et al.* BDM and ZDM methods provide an average TLG count reduction of 12% and 17%, respectively. The average reduction obtained by the proposed method is about 58%.

Finally, in order to verify the scalability of the proposed method, we synthesize 12 large OpenCore benchmarks [14]. The obtained results are presented in Table II. These benchmarks were synthesized to TLGs for the first time and can be adopted as reference for further comparisons. The execution time of both

TABLE II
Obtained results when synthesizing OpenCore benchmarks [14].

				Cut Computation			Mapping Results		Execution Time (s)		
Circuit	**PI**	**PO**	**AIG Nodes**	**Total Cuts**	**Unate Cuts**	**Threshold Cuts**	**TLGs**	**Levels**	**Identification Step**	**Mapping Step**	**Total**
oc_ethernet	12,284	1,272	10,820	60,332	5,759	1,027	3,893	7	5.8	13.5	19.3
oc_cordix_p2r	12,615	719	11,846	42,492	11,218	1,961	4,876	7	12.4	13.5	25.9
oc_cfft_1024x12	14,941	1,051	13,838	51,715	10,221	2,238	5,170	7	11.3	16.0	27.3
oc_cordic_r2p	16,822	1,015	15,773	57,321	14,858	2,796	6,043	6	16.9	15.8	32.7
oc_mem_ctrl	18,667	1,825	16,727	34,679	3,145	321	6,680	9	3.1	19.5	22.6
oc_fpu	25,853	659	24,932	364,570	25,436	2,741	9,561	265	29.8	53.3	83.1
oc_aquarius	26,999	1,477	25,058	208,686	16,823	1,831	9,776	21	19.2	50.2	69.4
oc_des_perf	32,002	1,976	29,905	94,051	1,271	131	11,322	7	1.1	60.2	61.3
oc_video_dct	51,885	3,549	46,433	521,682	33,743	3,195	17,127	13	40.4	73.5	113.9
oc_video_jpeg	62,293	3,972	56,601	425,542	28,527	3,191	21,735	12	34.9	76.6	111.5
radar20	87,635	6,001	78,342	195,782	17,401	2,330	33,936	14	19.8	106.1	125.9
uoft_raytracer	205,126	13,079	187,683	659,183	41,575	4,886	74,205	21	51.5	327.3	378.8

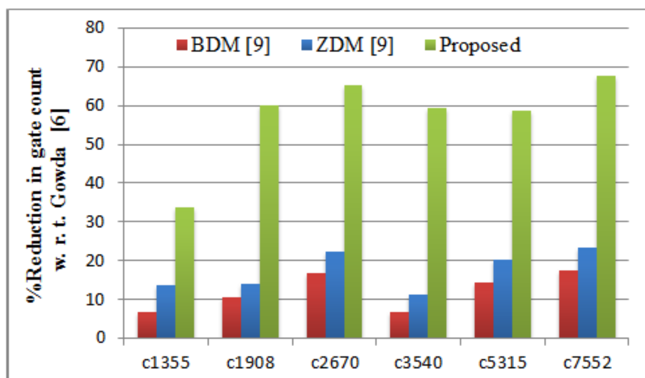

Fig. 3. Percentage gate count reduction in each approach, compared to Gowda [8].

identification and mapping steps are also presented in Table II. Notice that the execution time of the pre-computation step is proportional to the number of unate cuts. The TLF identification method performs an unateness checking as it first step avoiding binate functions, since all threshold functions are unate. This unateness checking has an insignificant execution time when compared to the complete identification process.

VI. Conclusions

In this paper, a novel approach to synthesize circuits using threshold logic gates (TLGs) is presented. The proposed method is based on a new synthesis flow, which allows us to use the multi-objective FPGA-based technology mappers and combines both TLG count and logic depth optimizations. The main contributions of this work are the following: (1) a simpler threshold logic synthesis flow, compared to previous work; (2) an efficient method based on cut pruning, scalable to large benchmark circuits; (3) threshold logic synthesis, which produces circuits using TLFs up to 9 inputs; and (4) experimental comparison and new results to be used as a reference in further publications. When compared against the state-of-the-art methods, the proposed method reduces the TFL count by 8% and logic depth by 46%.

We intend to improve the method proposed herein by identifying TLFs before populating the DSD manager. Currently, the priority cuts computation does not consider TLFs while sorting cuts. We plan to propose a TLF-based priority cuts computation, which would identify TLFs while sorting cuts. In this method, we are expecting to increase the occurrences of TLF cuts at each node, allowing the mapper to explore solutions disregarded in the current approach.

Acknowledgement

Research partially supported by Brazilian funding agencies CAPES, CNPq and FAPERGS, under grant 11/2053-9 (Pronem).

References

[1] *International technology roadmap for semiconductors*, 2011.

[2] L. Gao, F. Alibart, and D. B. Strukov, “Programmable cmos/memristor threshold logic,” *IEEE Trans. on Nanotechnology*, vol. 12, no. 2, 2013.

[3] D. Fan, M. Sharad, and K. Roy, “Design and synthesis of ultralow energy spin-memristor threshold logic,” *IEEE Trans. on Nanotechnology*, vol. 13, no. 3, 2014.

[4] N. S. Nukala, N. Kulkarni, and S. Vrudhula, “Spintronic threshold logic array (stla) - a compact, low leakage, non-volatile gate array architecture,” in *Proc. of Int’l Symp. on Nanoscale Architectures*, 2012.

[5] V. Beiu, J. Quintana, and M. Avedillo, “Vlsi implementations of threshold logic - a comprehensive survey,” *IEEE Trans. on Neural Netw*, vol. 14, no. 5, 2003.

[6] R. Zhang, P. Gupta, L. Zhong, and N. K. Jha, “Threshold network synthesis and optimization and its application to nanotechnologies,” *IEEE Trans. on Comput.-Aided Design of Integr. Circuits Syst.*, vol. 24, no. 1, 2005.

[7] J. L. Subirats, J. M. Jerez, and L. Franco, “A new decomposition algorithm for threshold synthesis and generalization of boolean functions,” *IEEE Trans. on Circuits Syst. I*, vol. 55, no. 10, 2008.

[8] T. Gowda, S. Vrudhula, N. Kulkarni, and K. Berezowski, “Identification of threshold functions and synthesis of threshold networks,” *IEEE Trans. on Comput.-Aided Design of Integr. Circuits Syst.*, vol. 30, no. 5, 2011.

[9] A. K. Palaniswamy and S. Tragoudas, “Improved threshold logic synthesis using implicant-implicit algorithms,” *ACM Journal on Emerg. Tech.*, vol. 10, no. 3, 2014.

[10] A. Neutzling, M. G. Martins, R. P. Ribas, A. Reis, *et al.*, “A constructive approach for threshold logic circuit synthesis,” in *Proc. of Int’l Symp. on Circuits and Syst.*, 2014.

[11] E. M. Sentovich, K. J. Singh, L. Lavagno, *et al.*, “Sis: a system for sequential circuit synthesis,” 1992.

[12] Berkeley Logic Synthesis and Verification Group, “Abc: a system for sequential synthesis and verification,” Release 20130425. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/.

[13] S. Yang, *Logic synthesis and optimization benchmarks user guide: Version 3.0*. Microelectronics Center of North Carolina, 1991.

[14] J. Pistorius, M. Hutton, A. Mishchenko, and R. Brayton, “Benchmarking method and designs targeting logic synthesis for fpgas,” in *Proc. of Int’l Workshop on Logic and Synthesis*, vol. 7, 2007.

[15] P.-Y. Kuo, C.-Y. Wang, and C.-Y. Huang, “On rewiring and simplification for canonicity in threshold logic circuits,” in *Proc. of Int’l Conf. on Comput.-Aided Design*, 2011.

[16] C.-C. Lin, C.-Y. Wang, Y.-C. Chen, and C.-Y. Huang, “Rewiring for threshold logic circuit minimization,” in *Proc. of Conf. on Design, Automation & Test in Europe*, 2014.

[17] A. Mishchenko, S. Chatterjee, and R. K. Brayton, “Improvements to technology mapping for lut-based fpgas,” *IEEE Trans. on Comput.-Aided Design of Integr. Circuits Syst.*, vol. 26, no. 2, 2007.

[18] P. Pan and C.-C. Lin, “A new retiming-based technology mapping algorithm for lut-based fpgas,” in *Proc. of Int’l Symp. on Field Programmable Gate Arrays*, 1998.

[19] S. Muroga, *Threshold logic and its applications*. 1971.

[20] A. Neutzling, M. G. Martins, R. P. Ribas, A. Reis, *et al.*, “Synthesis of threshold logic gates to nanoelectronics,” in *Proc. of Int’l Symp. on Integ. Circuits Systs. Design*, 2013.

[21] D. Chen and J. Cong, “Daomap: a depth-optimal area optimization mapping algorithm for fpga designs,” in *Proc. of Int’l Conf. on Comput.-Aided Design*, 2004.

[22] J. Cong and Y. Ding, “Flowmap: an optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs,” *IEEE Trans. on Comput.-Aided Design of Integr. Circuits Syst.*, vol. 13, no. 1, 1994.

[23] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, “Combinational and sequential mapping with priority cuts,” in *Proc. of Int’l Conf. on Comput.-Aided Design*, 2007.

[24] A. Mishchenko and R. Brayton, “Faster logic manipulation for large designs,” in *Proc. of Int’l Workshop on Logic and Synthesis*, 2007.

TABLE I

COMPARISON OF PROPOSED METHOD VS RELATED WORKS.

	Statistics		Gates (ratio)						Levels (ratio)					
Names	**input**	**output**	**Neutzling [10]**		**Zhang [6]**		**Proposed**		**Neutzling [10]**		**Zhang [6]**		**Proposed**	
des	256	56	1556	(1.00)	1920	(1.23)	1538	(0.99)	19	(1.00)	16	(0.84)	6	(0.32)
i10	257	55	840	(1.00)	1817	(2.16)	819	(0.98)	31	(1.00)	35	(1.13)	8	(0.26)
pair	173	54	563	(1.00)	907	(1.61)	518	(0.92)	17	(1.00)	12	(0.71)	5	(0.29)
i8	133	53	427	(1.00)	570	(1.33)	330	(0.77)	10	(1.00)	10	(1.00)	3	(0.30)
dalu	75	52	371	(1.00)	810	(2.18)	366	(0.99)	11	(1.00)	23	(2.09)	6	(0.55)
x3	135	51	280	(1.00)	441	(1.58)	268	(0.96)	7	(1.00)	7	(1.00)	4	(0.57)
apex6	135	50	279	(1.00)	396	(1.42)	270	(0.97)	10	(1.00)	12	(1.20)	4	(0.40)
alu4	14	49	275	(1.00)	410	(1.49)	250	(0.91)	22	(1.00)	23	(1.05)	8	(0.36)
i9	88	48	266	(1.00)	275	(1.03)	241	(0.91)	8	(1.00)	8	(1.00)	3	(0.38)
i7	199	47	197	(1.00)	304	(1.54)	262	(1.33)	3	(1.00)	5	(1.67)	2	(0.67)
x4	94	46	152	(1.00)	189	(1.24)	136	(0.89)	5	(1.00)	8	(1.60)	3	(0.60)
example2	85	45	151	(1.00)	182	(1.21)	122	(0.81)	6	(1.00)	8	(1.33)	4	(0.67)
i6	138	44	141	(1.00)	276	(1.96)	202	(1.43)	3	(1.00)	5	(1.67)	2	(0.67)
alu2	10	43	134	(1.00)	197	(1.47)	123	(0.92)	18	(1.00)	25	(1.39)	7	(0.39)
x1	51	42	107	(1.00)	203	(1.90)	76	(0.71)	5	(1.00)	7	(1.40)	3	(0.60)
i3	132	41	86	(1.00)	158	(1.84)	66	(0.77)	5	(1.00)	6	(1.20)	3	(0.60)
apex7	49	40	78	(1.00)	118	(1.51)	66	(0.85)	7	(1.00)	9	(1.29)	4	(0.57)
cht	47	39	73	(1.00)	82	(1.12)	73	(1.00)	2	(1.00)	5	(2.50)	2	(1.00)
my_adder	33	38	71	(1.00)	96	(1.35)	82	(1.15)	10	(1.00)	18	(1.80)	4	(0.40)
i4	192	37	70	(1.00)	74	(1.06)	66	(0.94)	9	(1.00)	5	(0.56)	3	(0.33)
i5	133	36	66	(1.00)	66	(1.00)	66	(1.00)	5	(1.00)	6	(1.20)	3	(0.60)
ttt2	24	34	62	(1.00)	100	(1.61)	49	(0.79)	6	(1.00)	6	(1.00)	3	(0.50)
i2	201	35	62	(1.00)	198	(3.19)	35	(0.56)	6	(1.00)	7	(1.17)	4	(0.67)
term1	34	33	60	(1.00)	226	(3.77)	43	(0.72)	7	(1.00)	10	(1.43)	4	(0.57)
c8	28	32	58	(1.00)	85	(1.47)	51	(0.88)	5	(1.00)	7	(1.40)	3	(0.60)
count	35	31	55	(1.00)	79	(1.44)	52	(0.95)	11	(1.00)	12	(1.09)	3	(0.27)
unreg	36	30	48	(1.00)	50	(1.04)	48	(1.00)	2	(1.00)	5	(2.50)	2	(1.00)
pcler8	27	28	36	(1.00)	47	(1.31)	30	(0.83)	4	(1.00)	7	(1.75)	2	(0.50)
frg1	28	29	36	(1.00)	59	(1.64)	17	(0.47)	8	(1.00)	9	(1.13)	3	(0.38)
comp	32	27	35	(1.00)	83	(2.37)	31	(0.89)	8	(1.00)	8	(1.00)	3	(0.38)
lal	26	26	32	(1.00)	54	(1.69)	36	(1.13)	4	(1.00)	7	(1.75)	2	(0.50)
parity	16	25	30	(1.00)	45	(1.50)	31	(1.03)	8	(1.00)	9	(1.13)	5	(0.63)
pcle	19	24	27	(1.00)	35	(1.30)	29	(1.07)	4	(1.00)	6	(1.50)	2	(0.50)
sct	19	23	25	(1.00)	38	(1.52)	27	(1.08)	11	(1.00)	5	(0.45)	2	(0.18)
cordic	23	22	24	(1.00)	49	(2.04)	26	(1.08)	6	(1.00)	7	(1.17)	3	(0.50)
f51m	8	21	24	(1.00)	82	(3.42)	32	(1.33)	6	(1.00)	8	(1.33)	3	(0.50)
cc	21	20	23	(1.00)	35	(1.52)	21	(0.91)	3	(1.00)	6	(2.00)	2	(0.67)
cm150a	21	18	21	(1.00)	21	(1.00)	16	(0.76)	5	(1.00)	4	(0.80)	4	(0.80)
cu	14	17	17	(1.00)	24	(1.41)	16	(0.94)	3	(1.00)	4	(1.33)	2	(0.67)
pm1	16	15	16	(1.00)	23	(1.44)	12	(0.75)	4	(1.00)	4	(1.00)	2	(0.50)
tcon	17	16	16	(1.00)	32	(2.00)	16	(1.00)	2	(1.00)	3	(1.50)	2	(1.00)
decod	5	14	16	(1.00)	24	(1.50)	16	(1.00)	1	(1.00)	3	(3.00)	1	(1.00)
cm162a	14	12	15	(1.00)	26	(1.73)	14	(0.93)	5	(1.00)	8	(1.60)	2	(0.40)
cm163a	16	13	15	(1.00)	25	(1.67)	14	(0.93)	5	(1.00)	6	(1.20)	2	(0.40)
i1	25	11	14	(1.00)	23	(1.64)	16	(1.14)	4	(1.00)	5	(1.25)	2	(0.50)
cmb	16	10	13	(1.00)	27	(2.08)	5	(0.38)	4	(1.00)	6	(1.50)	2	(0.50)
x2	10	9	13	(1.00)	15	(1.15)	12	(0.92)	4	(1.00)	4	(1.00)	2	(0.50)
z4ml	7	8	12	(1.00)	19	(1.58)	16	(1.33)	4	(1.00)	5	(1.25)	3	(0.75)
cm151a	12	7	11	(1.00)	12	(1.09)	8	(0.73)	5	(1.00)	5	(1.00)	3	(0.60)
cm152a	11	6	10	(1.00)	11	(1.10)	8	(0.80)	4	(1.00)	4	(1.00)	3	(0.75)
cm42a	4	5	10	(1.00)	13	(1.30)	10	(1.00)	1	(1.00)	3	(3.00)	1	(1.00)
cm85a	11	4	8	(1.00)	14	(1.75)	10	(1.25)	3	(1.00)	5	(1.67)	2	(0.67)
cm82a	5	3	8	(1.00)	12	(1.50)	10	(1.25)	3	(1.00)	4	(1.33)	3	(1.00)
b1	3	2	5	(1.00)	8	(1.60)	5	(1.00)	2	(1.00)	3	(1.50)	2	(1.00)
majority	5	1	1	(1.00)	1	(1.00)	1	(1.00)	1	(1.00)	2	(2.00)	1	(1.00)
Geomean				(1.00)		(1,54)		(0.92)		(1.00)		(1.30)		(0.54)