Vol.2016-ARC-218 No.8
Vol.2016-SLDM174 No.8
2016/1/19

Efficient Graph Matching Method for LUT-Networks

Hossein Izadi Rad[1,a)] Amir Masoud Gharehbaghi[2,b)] Masahiro Fujita[1,2,c)]

Abstract: We introduce a new approach for structural LUT matching based on Subgraph Matching. In LUT networks, same numbers of LUTs are matchable with one another regardless of their programs. Using this method, a new LUT network can be implemented on top of another LUT network with similar topology. One of the applications of this method could be IP reuse. First, we find all the possible matching candidates based on the conflicting criteria and generate a Conflict Graph. Then, we extract the Maximum Independent Set on the conflict graph with our proposed approximate algorithm. Finally, we improve the results iteratively in a controlled manner by considering the neighbours of the circuit elements. Our extensive experiments on ISCAS85 shows efficiency of our approach. Using this method, it takes less than 2 minutes to match the largest circuits of ISCAS85 with more than 2900 logic gates.

1. Introduction

Today's industrial applications are gaining the advantages of FPGAs in many hardware designs, especially when integrating them with other hardwares. Some of the advantages are: 1. Easier design integration with Intellectual Properties (IPs), 2. Freedom of changing the design for new protocol standards and functionality requirements, and 3. Scalling the performance by embedding IPs [1]. Advanced FPGAs have large resources of Configurable Logic Blocks (CLB), embedded microprocessor, RAM blocks, ADC/DAC converters, multi-bit multipiers, configurable interconnects and etc. CLBs are usualy consists of LookUp Tables (LUTs), flip-flops, carry chain and arithmatic logic units, and multiplexers [2]. In this work we focus on LUT netlist representing the given circuits. In other words, the given circuit is mapped to LUT netlist.

Finding common subcircuits among multiple circuits has many applications in the design phase as well as maintenance of designs. Designers can reuse the existing results of synthesis, verification, test generation, etc., from one circuit to another; hence, it avoids redoing the same tasks. In addition, finding similarities and differences among multiple LUT networks has other applications in design maintenance, fraud detection, reverse engineering, redundancy detection, and so forth. Each of the above-mentioned applications has a specific goal. Therefore, the matching engine should be general and flexible enough to be adapted according to the desired solution.

In this paper, we have considered the problem of finding common subcircuits, having similar structure, among multiple LUT networks. To solve the problem, we have introduced a new efficient approach based on graph algorithms.

The basic idea is to generate graphs corresponding to the LUT netlists and find the common subgraphs. The subgraph matching problem is considered to be an *NP-Complete* problem. We propose a two-step approximate solution. In the first step, we find all the possible matching candidates by excluding those which are in conflict with each other. The conflicting criteria can be defined and changed based on the application of matching. In this way, we generate a *Conflict Graph* from the LUT netlists to be matched. One of the main advantages of generating a conflict graph is that it allows us to easily define different criteria for matching, making our approach more general and flexible to meet the requirements of different applications.

As an example, we can think of a very straightforward matching criteria that is defined as each LUT in a netlist can only be matched with an LUT of same number of inputs. For example a 4-input LUT can only be matched with a 4-input LUT. However, we may define a more relaxed criteria for matching that allows matching an LUT with another one with different number of inputs. For example, a 4-input LUT can be matched with a 5-input LUT. This kind of matching is specially useful when we have a slightly modified circuit and we want to match it to the previous circuit structure. The modification may be because of a bug fix, or an *Engineering Change Order* (ECO).

In the second step, we find the *Maximum Independent Set (MIS)* in the conflict graph with help of our proposed approximation algorithm. The MIS shows all the common subcircuits among the input circuits. Our proposed MIS algorithm is based on the Ramsey algorithm [3] that is the basis of many other MIS algorithms. We have introduced different heuristics to improve the Ramsey algorithm, both in terms of the quality of the results and runtime, as also shown in our experimental results. The run time complexity of proposed algorithm is $O(n^3/(logn)^2)$.

Finally, we have improved the result of matching by considering the neighbors of the LUTs iteratively, in a controlled manner.

[1] Dept. of Electrical Engineering and Information Systems, The University of Tokyo, 2-11-16 Yayoi, Bunkyo-ku, Tokyo 113-0032, Japan
[2] VLSI Design and Education Center, The University of Tokyo, 2-11-16 Yayoi, Bunkyo-ku, Tokyo 113-0032, Japan
[a)] izadi@cad.t.u-tokyo.ac.jp
[b)] amir@cad.t.u-tokyo.ac.jp
[c)] fujita@ee.t.u-tokyo.ac.jp

The basic idea is that in the conflict graph, we include only the LUTs that their neighboring LUTs up to N levels are also candidates for matching. Accordingly, the possibility of picking up wrong LUTs as matching candidates decreases; as a result, the size of the common subcircuit may increase. We start by considering the first level neighbors (N = 1), and gradually increase the level as long as the increase in the size of the common subcircuit is satisfactory and the runtime is acceptable. In this paper, we call this approach *Controlled Look-ahead*.

We have performed experiments on ISCAS85 [4] circuits. We have shown that in practice the runtime of our method is near linear. Furthermore, comparing our results to the optimum solution that is obtained by an exact algorithm, we conclude that the average size of the identified common subcircuit is around 9% smaller, but the average processing speed is two to three orders of magnitude faster.

The rest of the paper is organized as follows. In Section 2, we overview on the Graph Matching problem definition and the previous work. Section 3 presents our proposed method. Section 4 gives the experimental results. Section 5 concludes the paper and gives some future directions.

2. Graph Matching Problem

The problem we are dealing with in this paper is subcircuit matching, which is also known as subcircuit recognition. This problem can be considered as graph matching problem by representing circuits as graphs. Graph matching is a classical problem in graph theory that belongs to NP-class of algorithms [6]. In this problem we have two graphs and we would like to understand how similar they are, and where those similarities reside. There are two categories of graph matching problems. The first category tries to find out whether two given graphs are exactly the same or not; which is also known as *Graph Isomorphism*. In the second category, the matching problem is subgraph matching in which the cardinality of the set of vertices are not necessarily the same. In subgraph matching problem, the goal is to find a mapping $H : V_1 \rightarrow V_2$ such that $(u, v) \in E_1$ if and only if $(H(u), H(v)) \in E_2$. When a mapping such as H exists, this is called an Homomorphism, and two graphs are called Homomorphic. This can be considered as quest for the largest subgraph in the smaller graph within the bigger one. For the formal definition of *Isomorphism* and *Homomorphism* please refer to [6]. Intuitively, graph homomorphism (subgraph matching) is a more complex problem than graph isomorphism (graph matching).

Many subgraph isomorphic algorithms have a search-based approach [8] [9]. They usually use two fundamental operations to extract the mapping. The first one is node-to-node mapping, which tries to search exhaustively on nodes to map them. The second one tries to preserve the connection of adjacent nodes. Ullmann's algorithm [12] is the basic algorithm for many studies on graph matching. This algorithm is basically a Depth First Search (DFS) algorithm in addition to some other refinements. The algorithm is actually a brute-force enumeration procedure that attempts to do a tree search for each node to reduce the number of successor nodes to be searched, leading to the reduction of total computing time. This approach is not only inefficient due to blind search each node, but limited with respect to implementation problem due to space limitation for refinement in memory.

Another major approach for graph matching algorithm is based on Corneils' algorithm [13], which is a typical Breadth First Search (BFS) during the procedure of re-labeling the nodes associated with their neighbor nodes. Originally, it focused on *Graph Isomorphism* problem and was extended with some ideas in subcircuit recognition problem. This algorithm is used as the kernel of a famous subcircuit recognition algorithm, SubGemini [14]. As the first step, this algorithm runs a preprocessing step on a representative and reordered representation of the input graphs. Then, it tries to solve the problem with the transformed graphs. This procedure can only provide an incomplete answer to the isomorphism problem and for some cases it cannot decide if the two input graphs are isomorphic or not. Additionally, SubGemini and other similar work are used for transistor level netlists which have different characteristics with gate-level netlists. For example, in transistor level netlist there is no connection that can bridge the nets, then we can consider the transistor netlists as bipartite graphs, but it is not correct for gate-level netlists.

3. The Proposed Method

Our problem is subcircuit matching with a general approach. The proposed framework is flexible enough to provide an efficient solution for the target applications, and fast enough for matching on large circuits.

Although the basic approach of this paper is mainly about finding subcircuits in LUT netlists, an efficient solution can also be applied to other kind of graphs we are dealing with in hardware and software design flow. In some applications we need to match a subcircuit with its slightly structurally different subcircuit. For example, a 3-input LUT can be considered as structurally-different functionally-equivalent with a 2-input LUT. What we need is a flexible framework capable of defining desired rules of similarity depending on the applications. A promising idea would be reducing the complexity of the original problem with a preprocessing step which can be refined in later iterations. Similarity rules can be applied in a preprocessing step, which tries to find matchable candidates and exclude the conflicting ones. Our idea is to do this by building a conflict graph out of the given graphs. This basic structure can be refined in further iterations for more accurate results.

Here, we present our proposed method. In the first step, we create a new graph out of the two graphs that we want to find matching. This new graph is called *Conflict Graph (CG)*. All the rules for similarity can be considered while building the *CG*. We add all the matching candidates as vertices of *CG* and the conflicts between them by edges. The conflict graph has edges between all vertices that are conflicting based on our desired definition of conflict. This means that we do not want a solution returned which contains vertices that are connected by an edge and this type of solution is called *Independent Set (IS)*. In the second step, we solve the circuit matching problem by solving a different problem, called *Maximal Set of Independent Sets (MSIS)*, on the conflict graph. In addition to the maximal matching region, the size of the MSIS tells us how good the match is -the bigger the

answer, the larger the total matching regions.

3.1 Making Conflict Graph

Given two graphs to be matched, as $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ where V_j is a set of vertices, and E_j is a set of edges. Assume the number of vertices in G_1 is N and the number of vertices in G_2 is M.

We form the conflict graph $G_{CG} = (V_{CG}, E_{CG})$ by adding up to $N \times M$ vertices to G_{CG} as follows. Consider a pair of vertices, one drawn from the set V_1 (call it V_i) and the other drawn from the set V_2 (call it V_α). We create a vertex $V_{i,\alpha} \in G_{CG}$ if the type of V_i is compatible with the type of V_α and they have the same number of fanins. Nonetheless, we can define any arbitrary condition to select potential matching points. We can consider the vertex $V_{i,\alpha}$ as being a potential matching point between the two input graphs.

Next, we add edges to G_{CG} between vertices for which there is a conflict. We add edges between two vertices $V_{i,\alpha}$ and $V_{j,\beta}$ if $i = j$ or $\alpha = \beta$, because each LUT can be included in just one subcircuit. In addition, we can add other edges based on the definition of conflict criteria. For example, if the inputs of a LUT are all from primary inputs, we do not want to match it with another LUT having inputs from internal signals.

Once this is done, we have our conflict graph computed and we are ready for the next step. In terms of computational complexity, generating the conflict graph is $O(N \times M)$.

Fig. 1 is a simple example illustrating the procedure of making conflict graph for two given LUT netlists.

3.2 Maximum Independent Set

Finding MIS is one the classical problems in graph theory. This problem is known to be NP-hard even for some restricted classes of graphs [15]. At the same time, there are some solutions for specific classes of graphs which are efficient. The best known exact algorithm [16] which has a depth-first search approach, has the complexity of $O(3^{(n/3)})$. We have re-implemented a fast and well-known algorithm [17] to check the efficiency of other approximation solution for small graphs. The complexity of this algorithm is $O(V \times E \times \mu)$, which V and E are the size vertices and edges respectively, and μ is the size of MIS.

The natural heuristic is *Greedy* algorithm, which selects a pivot node v and search in $N'(v)$ the set non-neighbor vertices of v. This results in a rapid accumulation of the independent set recursively looking at non-neighborhoods. The main concern here is that we completely ignore the neighborhoods of the pivot node, which may have much larger independent set. In other words, if we choose a bad pivot node, we may ignore many vertices in the final independent set.

Another idea is based on the *Ramsey* algorithm [3] for MIS. Similar to Greedy, we choose a vertex and search in the non-neighborhood. But this time we also search for the neighbor vertices of the pivot vertex, and find whichever may result in a bigger IS. The Ramsey algorithm takes a common approach which is called excluding subgraphs. In [3], the authors show that no algorithm using the idea of excluding subgraphs, can do significantly better than their presented Ramsey-based algorithm. We can see the pseudo code of Ramsey algorithm in Fig. 2. The

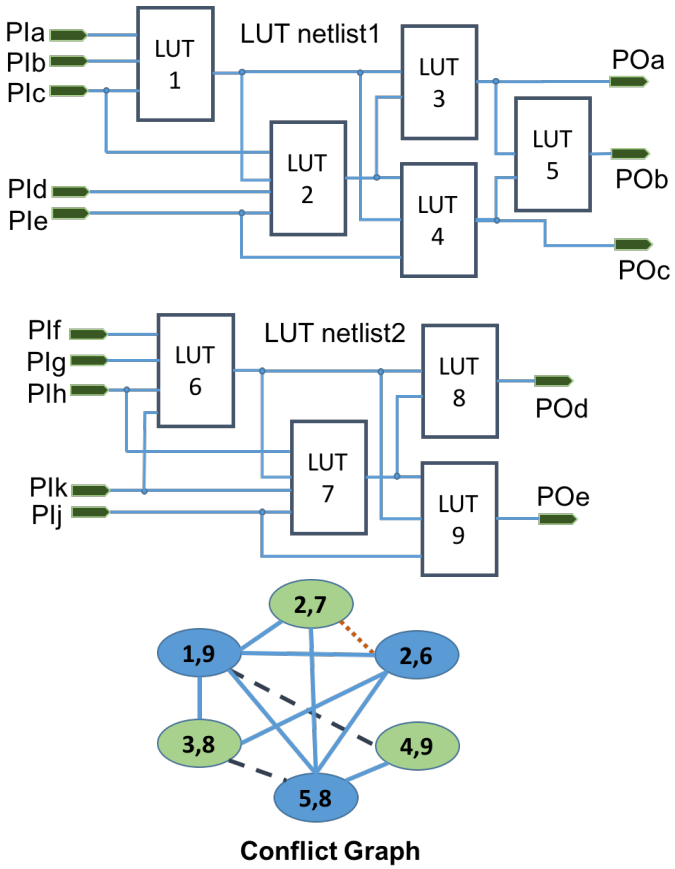

Fig. 1 Example of Building the Conflict Graph

Ramsey Algorithm:
Ramsey(***G***):

1. if G = Φ then return (Φ)
2. Choose v in V(G)
3. $(I_1) \leftarrow$ Ramsey (N(v))
4. $(I_2) \leftarrow$ Ramsey ($N^{'1}$(v))
5. return ***larger of (I_2 U {v}, I_1)***

N': The set non-neighbor nodes of G is V.

Fig. 2 Pseudo Code of Ramsey Algorithm

Ramsey algorithm breaks the problem into a tree-like structure of subproblems [3]. In one sense, the algorithm transforms the graph into a binary tree where each internal node is adjacent to all of its left descendants and non-adjacent to all of its right descendants.

The Ramsey algorithm has been used by NetworkX, a state-of-the-art graph manipulation tool-set. As shown in our experiments, the quality of results for NetworkX is not good for our applications. Therefore, we have proposed an approximation algorithm based on Ramsey, to improve the quality of results, as presented in the next subsection.

3.3 The Proposed Approximation Algorithm

Here, we present the formal description of the algorithm. In the following we define terminology used for description of algorithms.

Expand Independent Set:
EIS(***IS***):

1. while ({Appendable vertices of ***IS***} != Φ)
2. ***Pmax = p(IS)***
3. for i in all Appendable vertices
4. if ***p(IS U {v_i}) > Pmax***
5. ***Pmax = p(IS U {v_i})***
6. ***IS = IS U {v_i}***
7. return ***IS***

Fig. 3 Pseudo Code of EIS Algorithm

Exchange vertex of Independent Set:

EVIS(IS):

1. while (***NVS^1***!= Φ)
2. ***IS' ← exchange node C^2 with N^3***
3. ***IS = EIS(IS')***
4. return ***IS***

NVS^1: The set of vertices out of ***IS***, which are neighbors with just one of vertices inside ***IS***.

C^2: Is the current node inside the ***IS*** which is neighbor with ***N***.

N^3: Is the node out of ***IS*** which has just one neighbor node (***C***) in ***IS.***

Fig. 4 Pseudo Code of EVIS Algorithm

If the unordered vertices pair (u, v) is an edge in G, we say that u is a *neighbor* of v, and the set of neighbor vertices build *neighborhood*. Given an independent set IS and a vertex v outside IS, vertex b is an *Appendable* vertex, if and only if $IS \cup \{v\}$ is still an independent set of G. We define a function $\rho(v)$ the number of all appendable vertices of an independent set IS.

We define two main operations on the conflict graph, and give the pseudo code of the algorithm itself. This algorithm has two main sub-routines. The *EIS* tries to make the largest IS for a pivot vertex. The idea is based on the Ramsey algorithm. The *EVIS* subroutine tries to exchange the vertices in IS with another candidates. These candidates are the vertices out of IS those are neighbors with just one of the vertices in IS. The philosophy behind this exchange is trying to make the IS as large as possible by enumerating all other possible candidates. The *EIS* runs after this exchange and expands the IS.

in *Expand Independent Set (EIS)* we try to find the largest IS including the chosen pivot node. Given a graph G with n vertices and independent set IS of G, if IS has no appendable vertices no expansion is possible, otherwise for each appendable vertex v of IS, find the number $\rho(IS \cup \{v\})$ of appendable vertices of the independent set $IS \cup \{v\}$. Let v_{max} denote the appendable vertex such that $\rho(IS \cup \{v_{max}\})$ is a maximum and obtain the independent set $IS \cup \{v_{max}\}$. The pseudo code of EIS algorithm is shown in Fig. 3.

In *Exchange Vertex Independent Set (EVIS)* sub-routine we exchange the vertices out of IS with included vertices and find the largest IS after that. IS is the independent set of the given graph,

Set of Local Maximal Independent Sets:
SLMIS(***CG***):

1. ***CG*** ← Read Conflict Graph
2. ***n*** ← size of ***CG***
3. for ***i*** from 1 to ***n***
4. ***IS_i*** = {***i***}
5. ***IS_i*** = ***EIS(IS_i)***
6. ***IS_i*** = ***EVIS(IS_i)***
7. return MIS = Largest of {***IS_1***, …, ***IS_n***}

Fig. 5 Pseudo Code of Enhanced SLMIS Algorithm

if there is no vertex v outside IS such that v has exactly one neighbor w inside IS, save IS. If there is a vertex outside IS, find a vertex such that v has exactly one neighbor w inside IS. Define $IS_{v,w}$ by appending v to IS and removing w from IS. We do the *EIS* procedure on $IS_{v,w}$ and save the resulting independent set. The pseudo code of EVIS algorithm is shown in Fig. 4.

Our proposed approximation algorithm is shown in Fig. 5, which uses the two previous procedures for finding the set of local maximal independent sets. For each vertex of CG we run *EIS* and *EVIS* and the result is the independent set for that pivot vertex or IS_v. The *MIS* is the largest IS_v found. *MIS* shows the largest common subcircuit and the set of all *ISs* shows all the dissociated subcircuits.

The *EIS* algorithm is very similar to Ramsey algorithm presented in [3], in terms of structure and computation complexity. Therefore, we claim that *EIS* algorithm has the same run time complexity order, $O(n/(logn)^2)$. As for the *EVIS* algorithm the run time complexity is $O(n^2/(logn)^2)$. Therefore, the run time complexity of our main approximation algorithm, in the worst case, is $O(n^3/(logn)^2)$. However, we can see in experimental results that for real IP circuits the run time scales near-linearly by the size of circuit.

3.4 Enhanced Controlled Look-Ahead Algorithm

In this section, we show our enhanced algorithm by improving the approximation. It takes the number of iterations from user to enhance the results. In each iteration, the input is the conflict graph of the previous iteration and the output is a new conflict graph with less or equal number of vertices. We try to make a new conflict graph by going forward or backward towards primary inputs or primary outputs. After that we add conflict edges, in order to decrease the number of possibilities for matching LUTs. As a result, we may have better performance ratio with acceptable time overhead. In this method we consider primary inputs and primary outputs (PIs and POs) as unique node types. PIs have no input but may have one or more outgoing edges, and POs do not have any outputs but have just one incoming edge. What we described as the proposed method is zero-step look-ahead (0-LA). Actually by increasing the number of iterations, we are doing path matching. For example in 1-step look-ahead we are trying to match 2-LUTs paths, while in 2-step look-ahead these are 3-LUTs paths, and so on.

3.4.1 One-Step Look-Ahead Algorithm (1-LA)

In $1-LA$ we have the conflict graph of $0-LA$ step, $0-LACG$ (Zero-Step Look-Ahead Conflict Graph), in hand and we try to

build a new conflict graph, $1-LACG$, as output. In the first step, for each V_i, $V_\alpha \in E_{0-LACG}$, we decide to expand the path first backward and then forward.

In the second step, we try to add new vertices to $1-LACG$. Assume that in the first step we decided to go backward; then we analyze the LUTs which inputs of V_i are coming from with the LUTs which inputs of V_α inputs are coming from. If there is no mismatch between the inputs of V_i and the inputs of V_α we add some new nodes in $1-LACG$. At the end of this step, we have created all the vertices in $1-LACG$. In the third step, we add edges to $1-LACG$ between the vertices for which there is a conflict. This step is very similar to what we discussed in section 3.1 about adding conflict edges to conflict graph, but with slight modification.

3.4.2 N-Step Look-Ahead Algorithm (N-LA)

In $N-LA$ we have the conflict graph of the $(N-1)-LA$ step, $(N-1)-LACG$, in hand and we try to build a new conflict graph, $N-LACG$ as output. Building $N-LACG$ with $(N-1)-LACG$ is very similar to what we did for creating $1-LACG$ from $0-LACG$. The difference is that the vertex of $(N-1)-LACG$ is like $\{v_{11}, v_{12}, \ldots, v_{1w}\}$ and $\{v_{21}, v_{22}, \ldots, v_{2w}\}$ which $w \leq N$. We will go forward or backward and will have a $N+2$ frame containing a path of $N+1$, which has been detected in $(N-1)-LA$.

Although, the N can be as big as the length of the critical path of the circuit, by observing the experimental results we can see that $2-LA$ approximation provide acceptable performance ratio with reasonable run time overhead.

4. Experimental Results

We have conducted a series of experiments on all of the ISCAS85 circuits [4] to show the efficiency as well as effectiveness of our method.

We have implemented our approximation algorithm as well as our enhanced approximation algorithm in Java. We call the base approximation algorithm (without look-ahead) zero-step look-ahead algorithm (*0-LA*). The enhanced approximation algorithm is currently implemented for 1-step look-ahead (*1-LA*) and 2-step look-ahead (*2-LA*). Moreover, we have implemented the algorithm that identifies the exact solution (*Exact*) as the reference [16].

We used *ABC* tool-set [5] for mapping the verilog circuits into LUT netlist. The benchmark is given in *bench* format [10]. Using the commands *strash* and *if -k 4* the benchmark is mapped to 4-input LUT netlist. The result is an LUT netlist with 1,2,3 or 4-inputs. However, in some cases 1-input LUTs are used as buffer between PIs and POs, or internal signals and POs. In such cases, we removed the buffer LUTs and connect signals directly. Therefore, only the LUTs with functional logic (2,3 or 4-input LUTs) are considered for matching process. Fig. 6 shows the ISCAS85 benchmarks and their characteristics. The total number of LUTs and non-buffer LUTs are shown in this figure.

As for c2670 benchmark, we found a *GND* signal in LUT netwrok. The *OutYgreaterXEqual* signal is an error correction signal, checking the result equality of two Carry Look-ahead Adders (CLAs) [4]. Because two CLAs have similar functionality, this signal is reported as always 0 after mapping by *ABC*. We removed this signal for matching experiments.

All the experiments are conducted on a server with dual Xeon 2.9GHz processor with 128GB memory running Linux kernel 2.6.32 (64 bits).

In the first experiment, we have compared each ISCAS85 circuit against topologically the same circuit. Then the matched subcircuit should have the same number of vertices as the input circuits. Fig. 6 shows the run time and performance ratio of exact and approximation algorithms. 1-LA has increased the runtime compared to 0-LA by 24.5% on average. 2-LA has increased the runtime compared to 1-LA by 28.5% on average, and compared to 0-LA by 45.7% on average. We can compare the quality of the approximation algorithms in terms of performance ratio. All three approximation methods have the performance ratio of 100%.

Fig. 7 shows the runtime of the algorithms for different benchmarks in logarithmic scale. As it can be seen, the Exact algorithm runtime is two to three orders of magnitude more than our approximation algorithm; hence, not practical for larger benchmarks.

Fig. 8 shows the runtime of the approximation algorithms for different benchmarks based on the Conflict Graph nodes. As the result, the proposed approximation algorithm has a sub-linear run time complexity.

In the second experiment, we compared some ISCAS85 benchmarks with structrally-different but functionally-similar netlists. We compared the LUT size of Maximum Common Sub-circuit with exact and approximation algorithms. For example, we compared *c499* vs. *c1908*, a 32-bit SEC circuits and 16-bit SEC/DEC circuits respectivly. The largest sub-circuit found has 50 LUTs.

Fig. 9 shows the comaparison between different circuits and the LUT size of largest sub-circuit found by each algorithm.

Fig. 10 shows the performance ratio of approximation algorithms for each comparison.

5. Conclusions and Future Work

We addressed the problem of finding the maximum common subcircuit among LUT netlists. By performing extensive experiments on ISCAS85 circuits. Comparing our results for ISCAS85 circuits to the optimum solution, the average size of the identified common subcircuit is around 10% smaller, but the average processing speed is two to three orders of magnitude faster. Our future work includes improving both performance of the algorithm and the quality of the results for LUT netlists by searching for cones instead of paths. Moreover, we are going to incorporate logical transformations in order to identify more common subcircuits.

References

[1] Philip H.W. Leong, *Recent Trends in FPGA Architectures and Applications,* 4th IEEE International Symposium on Electronic Design, Test & Applications , 2008.
[2] Mario Vestias, Horacio Neto. *TRENDS OF CPU, GPU AND FPGA FOR HIGH-PERFORMANCE COMPUTING,* 24th International Conference on Field Programmable Logic and Applications (FPL), 2014.
[3] R. Boppana and M. M. Halldorsson, *Approximating Maximum Independent Sets by Excluding Subgraphs,* Springer BIT Numerical Mathematics, Vol. 32, pp. 180-196, 1992.
[4] *http://web.eecs.umich.edu/ jhayes/iscas.restore/*

Circuit										Exact Alg	0-LA Alg.		1-LA Alg.		2-LA Alg.	
Name	Functionality	Gates	Inputs	Outputs	LUTs	GND Signal	Buffer LUTs (1-input)	Logic LUTs (2-4 inputs)	Nodes in Conflict Graph	Time (sec)	Time (sec)	Perf. Ratio %	Time (sec)	Perf. Ratio %	Time (sec)	Perf. Ratio %
c432	Interrupt Controller	161	36	7	85	0	0	85	2817	521.4	0.10	100	0.14	100	0.64	100
c499	32-bit SEC	358	41	32	74	0	0	74	4420	798.9	1.64	100	2.37	100	5.60	100
c880	8-bit ALU	203	60	26	122	0	0	122	6190	923.4	1.15	100	1.30	100	3.49	100
c1355	32-bit SEC	515	41	32	74	0	0	74	4420	798.9	1.66	100	2.40	100	5.66	100
c1908	16-bit SEC/DEC	719	33	25	124	0	0	124	5450	897	1.98	100	3.35	100	7.64	100
c2670	12-bit ALU & Controller	998	234	140	212	1	25	187	17014	1540.6	4.49	100	7.40	100	12.4	100
c3540	8-bit ALU	1447	50	22	402	0	0	402	61794	2919.6	19.12	100	25.30	100	35.1	100
c5315	9-bit ALU	1995	178	123	530	0	28	502	118868	41489.2	26.49	100	33.99	100	44.9	100
c6288	16×16 Mul.	2417	32	30	516	0	0	516	221237	46092.3	51.40	100	64.75	100	82.9	100
c7552	32-bit adder/comparator	2979	207	108	628	0	47	581	131814	43589	29.04	100	39.5	100	54.1	100

Fig. 6 Performance Ratio and Run Time of Algorithms on ISCAS85 Circuits

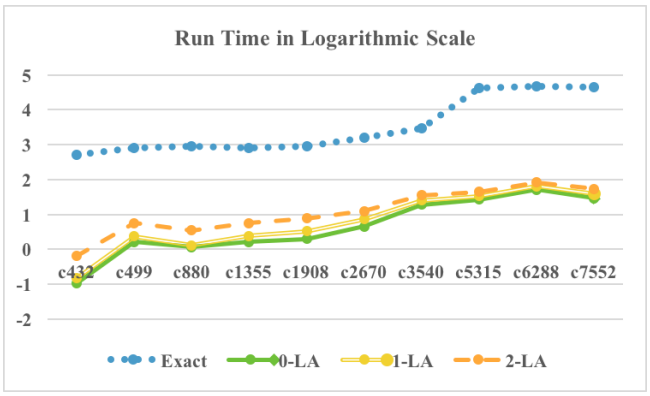

Fig. 7 Run Time of Algorithms in Logarithmic Scale

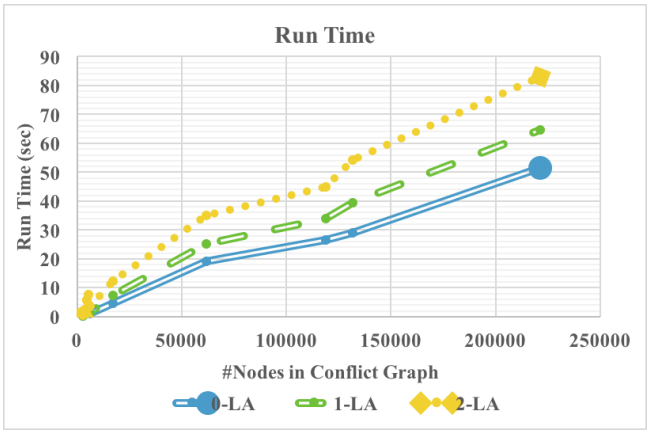

Fig. 8 Run Time of Aprox. Algorithms based on the Conflict Graph Size

	Max. Sub-circuit #LUTs			
	Exact	0-LA	1-LA	2-LA
Circuits	#LUTs	#LUTs	#LUTs	#LUTs
c499 vs. c1908	50	45	50	50
c1355 vs. c1908	51	43	47	49
c880 vs. c2670	110	100	100	104
c880 vs. c3540	103	89	91	98
c880 vs. c5315	100	96	100	100
c2670 vs. c3540	159	128	133	147
c2670 vs. c5315	180	142	153	161
c3540 vs. c5315	330	249	283	296

Fig. 9 Comparing Different Circuits with Similar Functionlaity

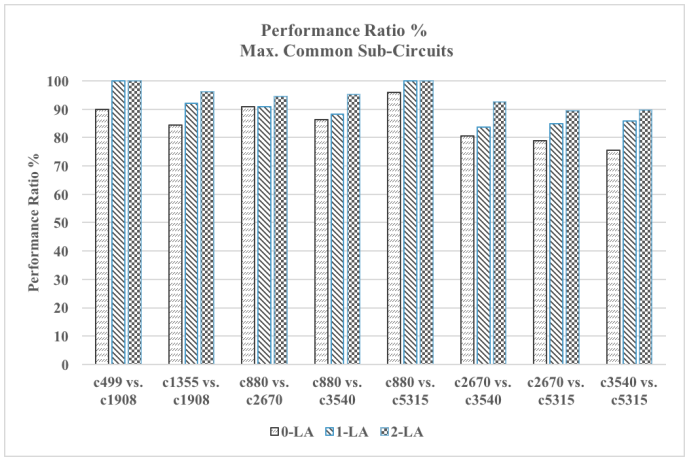

Fig. 10 Performance Ratio of Approx. Algorithms in Comparing Different Circuits with Similar Functionlaity

[5] *http://www.eecs.berkeley.edu/ alanmi/abc/*

[6] D. A. Basin, *A Term Equality Problem Equivalent to Graph Isomorphism,* Elsevier Information Processing Letters, Vol. 51, pp. 6166, 1994.

[7] M. M. Halldorsson and J. Radhakrishnan, *Improved Approximations of Independent Sets in Bounded-Degree Graphs,* Algorithm TheorySWAT'94, Vol. 824, 1994, pp. 195-206, 1994.

[8] V. Lipets, N. Vanetik, and E. Gudes. *Subsea: an efficient heuristic algorithm for subgraph isomorphism,* Springer Journal of Data Mining and Knowledge Discovery, Vol. 19, pp. 320-350, 2009.

[9] L. P. Cordella, P. Foggia, C. Sansone, and Mario Vento, *A (sub)graph isomorphism algorithm for matching large graphs,* IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 26, pp. 1367-1372, 2004.

[10] A. Mishchenko, N. Een, R. Brayton, M. Case, P. Chauhan, and N. Sharma, *Semi-Canonical Form for Sequential AIGs,* Conference on Design, Automation and Test in Europe, pp. 797-802, 2013.

[11] R. M. Karp, *Reducibility among Combinatorial Problems,* 50 Years of Integer Programming 1958-2008, Springer, pp. 219-241, 2010.

[12] J. R. Ullmann, *An Algorithm for Subgraph Isomorphism,* Journal of the ACM, Vol. 23, pp. 31-42, 1976.

[13] R. C. Read and D. G. Corneil, *The Graph Isomorphism Disease,* Journal of Graph Theory, Vol. 1, pp. 339-363, 1977.

[14] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather, *SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm,* Design Automation Conference, pp. 31-37, 1993.

[15] V.E. Alekseev, *Polynomial algorithm for Finding the Largest Independent Sets in Graphs Without Forks,* Elsevier Journal of Discrete Applied Mathematics, pp. 316, 2004.

[16] E. Tomitaa, A. Tanakaa, and H. Takahashia, *The worst-case time complexity for generating all maximal cliques and computational experiments,* Elsevier Journal of Theoretical Computer Science, pp. 2842, 2006.

[17] Sh. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa *A new algorithm for generating all the maximal independent sets,* SIAM Journal of Computing, pp. 505-517, 1977.