DECREASING TRANSISTOR COUNT USING AN EDGES SHARING TECHNIQUE IN A GRAPH STRUCTURE

Vinicius N. Possani, Renato S. de Souza, Julio S. Domingues Jr.,
Luciano V. Agostini, Felipe S. Marques, Leomar S. da Rosa Jr.
{vnpossani, rsdsouza, jsdomingues, agostini, felipem, leomarjr}@inf.ufpel.edu.br

Group of Architectures and Integrated Circuits – GACI
Federal University of Pelotas – UFPel
Pelotas – Brazil

ABSTRACT

Increasingly, in VLSI designs, the integrated circuits have higher density of transistors on the small physical area, power consumption reduced and greater performance. An important factor that has contributed for this is the representation of logic functions with a reduced number of transistors. Thus, we sought an alternative solution to common methods, such as factorization, to generate optimized transistors networks. This paper presents a graph-based structure to represent a transistor network and a technique to reduce the number of transistors by edges sharing. Our method can achieve non-series-parallel arrangements while methods based in factorization can only derive series-parallel arrangements, which may not be the best solution. Thus, when applied to the set of 4 input p-class logic functions, our method has advantages if compared to the good-factor algorithm implemented in SIS software. Moreover, using other arbitrary logical functions our algorithm can achieve results as good as those generated by techniques based in BDD methods.

1. INTRODUCTION

The micro electronics industry has brought great advances in last years, no doubt, designing digital circuits VLSI becomes an increasingly task of extreme complexity and high cost of resources and time. In this context, aid tools are applied to support these projects, contributing to the designers manipulate more transistors and decreasing the development cycle. Therefore, the automatically generation of transistor networks makes simple some arduous tasks. Moreover, it also reduces the aggregate cost to the final product.

This paper proposes an edge sharing method, on a graph structure, to generate optimized transistor networks. In our approach, the input Boolean expression is translated into a graph that is later optimized through edges sharing. Nowadays, alternative methods which are available in the literature has been study and applied in this context. They are based on graph optimizations, were each edge in the graph keeps an association with a transistor in the network. The main idea is try to minimize the edges in an existent graph [1] or to compose a new graph with a reduced number of edges [2]. These alternative methods are used because the common technique to optimize a transistor network is based on factorization [3-4] and this may not be an optimum solution [5]. In factorization method an input Boolean expression is manipulated in order to reduce the number of literals that compose the expression. Subsequently, this optimized expression is translated in a transistor network composed by a reduced number of switches. In this sense, our sharing method intent derives non-series-parallel arrangements in order to deliver better results than the common technique.

2. EDGES SHARING METHOD

The edges sharing method considers as input a sum-of-products (SOP) expression. In order to translate the expression to a graph, a parser is needed. The parser will deliver one vector of literals for each product storing these vectors in a list. Afterward, it is started the assembly of the graph by removing vectors one at a time from this list and creating an edge in the graph for each literal found in the vector. As an example we will use the Exp. (1) which represents a 'XOR' with 4 inputs. Figure 1.a shows the graph obtained of this expression.

!A*!B*!C*D + !A*!B*C*!D + !A*B*!C*!D + !A*B*C*D +
A*!B*!C*!D + A*!B*C*D + A*B*!C*D + A*B*C*!D (Exp.1)

In the sequence, all paths in the graph are traversed in order to recognize identical edges (edges that represent same literals and may have at least one vertex in common). If this condition is verified in the graph, then the identical edges are shared. This procedure consists in keeping only one of these identical edges, eliminating the remaining edges and merge the vertices which connect these. The vertices that will be merged are detached with the circumferences without fill in the figures below. This is exemplified in Figure 1.b where the edge '!A' was shared and the vertices 1, 5, 8 and 11 were merged. Now the edge 'A' will be shared generating the graph shown in Figure 1.c. So, in this moment the vertices 8 and 17, one at a time, are considered the new starting point of the optimization process, where the algorithm sought identical edges between these two vertices and the vertex 4. This way the edge '!B' connected to the vertex 8 will

be shared and in sequence this occurs with edge ’B’. Afterward that, the same process is applied to the edges ‘!B’ and ‘B’ attached to the vertex 17. This is demonstrated in Figure 1.d.

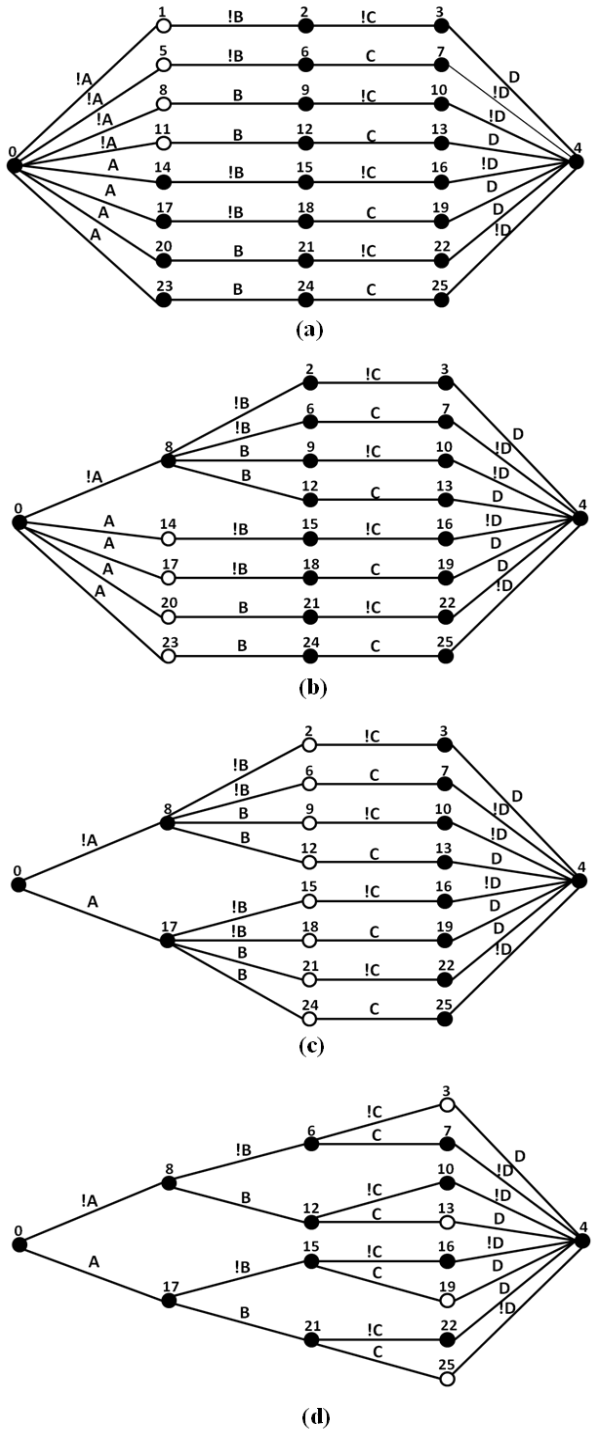

Fig. 1 – Steps of the sharing method, traversing the graph from the vertex 0 to the vertex 4.

Considering the Figure 1.d, departing from the vertices 6, 12, 15 and 21, one at a time, and traversing the graph toward the vertex 4, it is not possible perform new optimizations because identical edges are not found between these vertices and the vertex 4. Then the graph is traversed from the vertex 4 to the vertex 0. Thus, the edges ‘D’ are identified and were shared as the Figure 2.a. demonstrates. In the next step the edges ‘!D’ will be shared resulting in the graph of the Figure 2.b.

Now, consider the vertices 10 and 19 as new start points of the sharing method. There are two edges ‘!C’ connected on the vertex 10, as Figure 2.b shows. Then it is possible to remove the edge ‘!C’ attached to the vertices 10 and 12 merging the vertices 12 and 15. In this case, the merging of the vertices 12 and 15 will derive two edges ‘C’ between the vertices 19 and 15. When this is detected, just one of these edges remains in the graph. Figure 2.c shows this state of the graph. This process is applied again, but this time to the edges ‘C’ that are connected to the vertex 10, merging the vertices 6 and 21. This will derives two edges ‘!C’ between the vertices 19 and 21, one of this edges will be removed resulting in the final graph illustrated by Figure 2.d.

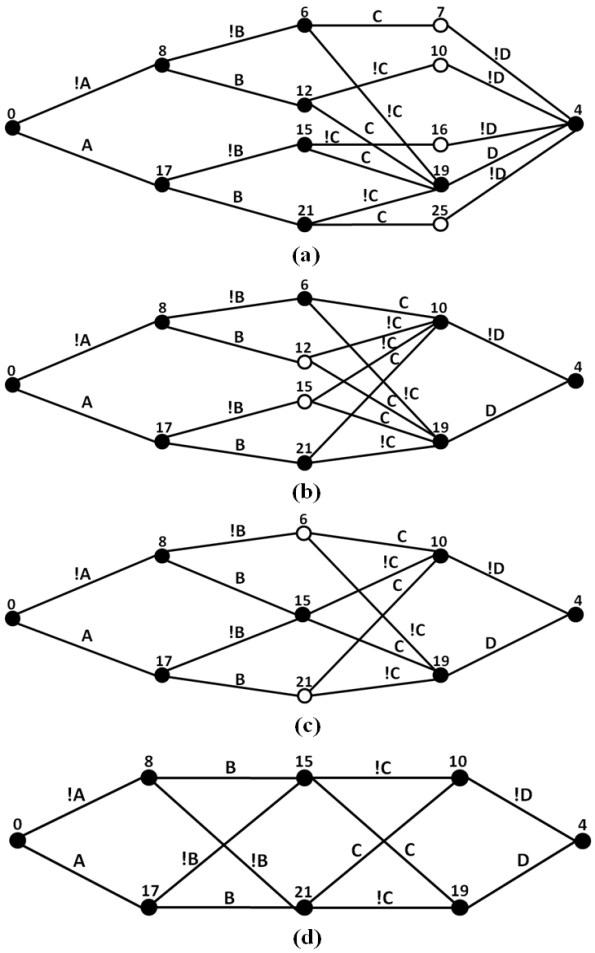

Fig. 2 – Steps of the sharing method, traversing the graph from the vertex 4 to the vertex 0.

Afterward, starting from the vertex 19, it is not possible to perform other optimizations. Thus, the optimization process ends. If any of these processes generates an invalid path, a recovery routine is invoked and the process is reversed. In the next session this procedure will be explained.

3. LOGICAL EQUIVALENCE CHECK

To guarantee that the optimized transistor network will be the faithful representation of the original expression, a validation procedure is applied making sure that all products described in the SOP are present in the resultant graph and sure that sneak-paths (forbidden paths) are not introduced on the network. Thus, it is necessary to validate all paths of the graph each time an edge is shared.

This procedure consists in comparing each path with the products that compose the original expression. If a path that does not belong to SOP has been introduced, a routine checks if this path is sensitized or not sensitized. When thinking in a transistor network, a path cannot be sensitized if it contains both polarities, for example 'A' and '!A'. In order words, this path is not a valid path. If the new path introduced is not sensitized it is accepted, since it does not change the logical behavior of the circuit. Otherwise, the graph needs to be restored to a step before the optimization that generated the new path, and this optimization is discarded. For this, a restore routine is invoked. This routine is responsible for recovering the edges and vertices that were eliminated from the graph and reconnect them.

Notice that all original products of the Exp. (1) are present in the graph of the Figure 2.d. However, by sharing edges, some new paths were also introduced. All these paths are allowed because are paths that cannot be sensitized.

4. EXPERIMENTAL RESULTS

The proposed method has been implemented in Java language using the NetBeans IDE. A tool containing the core algorithms and the graphics interface was developed. The graphics interface uses the Prefuse library [10]. Figure 3 illustrates it for the example from Expression 1. Also, the tool presents a Spice netlist output module that is capable to print Spice files. This feature permits the integration of the developed tool with other commercial or academic ones, making possible its application in a real design flow. For example, the obtained Spice netlist can be directly applied to electrical simulators.

Another interesting fact is that the proposed approach may derive Wheatstone bridge networks like methods proposed by [1] and [2]. The example illustrated in Figure 3 presents some bridge configuration. It is a benefit over optimization approaches based on factorization that can only derive series-parallel arrangements.

To describe our edges sharing method we used the Exp. (1), referring to a 'XOR' with four inputs. The achieved network was compared to the result obtained by others techniques described in [5], as BDD, OpBDD, LBBDD, and to the good-factor algorithm from SIS software. Our method reaches the same result like the BDD, OpBDD and LBBDD methods, with 12 transistors, overcoming the SIS with 16 transistors. Figure 3 shows the graphical interface of the Soptimizer, which illustrates a logic plain for the 'XOR' with four inputs after optimization.

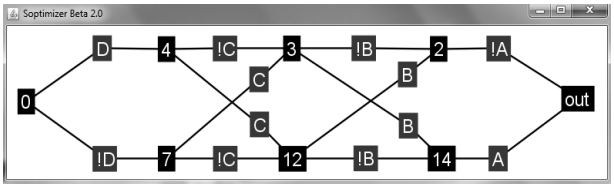

Fig. 3 – Graphical interface of Soptimizer showing the optimized XOR with 4 inputs.

In order to evaluate the proposed approach, 10 Boolean functions with 7 input variables were randomly chosen. They were introduced in SIS software [11] and extracted in their SOP form. Table 1 presents the obtained results. The total numbers of literals for the SOP form are described in column "# literals SOP form". In this analysis the expressions were factorized using the quick-factor algorithm from SIS. The results are shown in column "# literals SIS". Results obtained using the proposed approach are shown in column "# edges Soptimizer". The obtained gains over the SIS software are shown in column "% of gain".

Table 1 – Results for 10 Boolean Random Functions with 7 input.

Function	# literals SOP form	# literals SIS	# edges Soptimizer	% of gain
F1	133	76	61	19,73
F2	92	56	44	21,42
F3	78	48	39	18,75
F4	150	86	67	22,09
F5	119	68	58	14,70
F6	71	39	36	7,69
F7	170	99	80	19,19
F8	135	85	64	24,70
F9	111	70	56	20,00
F10	97	56	46	17,85

In all cases the proposed approach achieved smaller results. Analyzing the obtained networks we identified several bridge configurations in the arrangements delivered by the tool. On the other hand, SIS software delivers optimized expressions composed by 'AND' and 'OR' operators. In this case only series-parallel networks can be implemented, representing an overhead in terms of area (transistor count).

Finally, the set of 4 input p-class logic functions was used as benchmark to evaluate our proposed algorithm. This set is composed by 3982 logic functions. Each logic function was applied to SIS software as well as to our solution. When running in SIS, the two available algorithms were used, the quick-factor and the good-factor. However, our proposed method was able to deliver better solutions, reducing the total number of switches in the networks as Tab. 2 illustrates.

Table 2 – Total transistor count for the set of 4 input p-class logic functions.

	Total transistor count
Our solution	34990
Good-factory	37723
Quick-factory	38341

From the total of 3982 logic functions, our algorithm was able to deliver 1900 networks with less transistor count if comparing to the good-factor solution, 2033 networks with exact same transistor count, and only 49 networks with a smaller increase. Figure 4 shows the amount of logic functions were our algorithm loses and wins if comparing to the good-factor solution from SIS. As it is possible to see, our approach is capable to reduce up to 5 transistors in some networks. On the other hand, the good-factor achieves some smaller transistor networks. On these 49 cases, our algorithm was not able to generate bridge configuration. The generated networks are purely series-parallel solution.

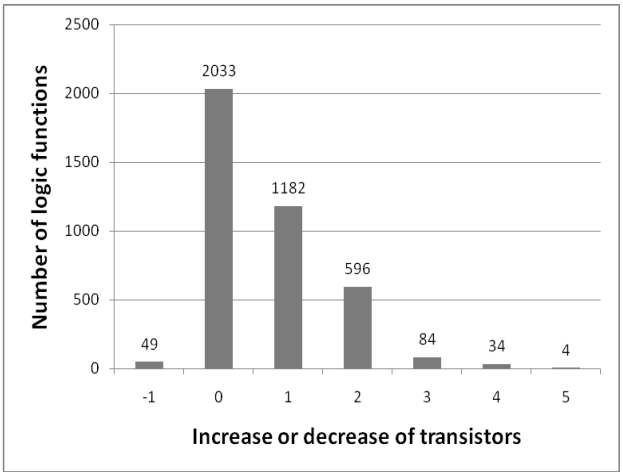

Fig. 4 – Number of functions with an increase or decrease in the transistor count when comparing with good-factor algorithm.

5. CONCLUSIONS AND FUTURE WORKS

This paper presented an edges sharing method to derive optimized transistor networks. The algorithm was implemented in Java language and the graphical interface uses the Prefuse library to show the optimized network.
The tool presents a Spice netlist output module that is capable to print Spice files. Moreover our approach can derive Wheatstone bridge networks. This arrangements can represents a logic function with a reduced number of transistors.

To describe our algorithm step by step, we use an 'XOR' with 4 inputs. This optimized network presents the same result of the methods in [5] with 12 transistors surpassing the SIS solution with 16 transistors. Also, 10 Boolean expressions with 7 input variables were randomly chosen to be used as benchmark. The results demonstrated that the proposed approach can delivery networks with a transistor reduction up to 24.70% if compared to the quick-factor algorithm from the SIS software.

Nevertheless, when using the set of 4 input p-class logic functions, our solution is able to perform a considerable reduction of the total transistor count. Moreover, it is capable to deliver 1900 networks with less transistor count, if comparing to the good-factor solution, reduce up to 5 transistors in some networks. The optimized transistor networks generate by our approach are validated ensuring the logical behavioral of the network. As future work we intend to evaluate the complexity of the algorithm. Also, we intend to compare the proposed solution with the method described in [1-2].

6. REFERENCES

[1] J. Zhu et al. On the Optimization of MOS Circuits. IEEE Transactions on Circuits and Systems: Fundamental Theory and Applications. (1993), 412-422.

[2] D. Kagaris et al. A Methodology for Transistor-Efficient Supergate Design. IEEE Transactions On Very Large Scale Integration (VLSI) Systems. (2007), 488-492.

[3] Brayton, R. K. Factoring logic functions. IBM J. Res. Dev. 31, 2. (1987), 187-198.

[4] Mintz, A. and Golumbic, M. C. Factoring boolean functions using graph partitioning. Discrete Appl. Math. 149, 1-3. (2005), 131-153.

[5] Da Rosa Jr, L. S. Automatic Generation and Evaluation of Transistor Networks in Different Logic Styles. PhD Thesis PGMicro/UFRGS, Porto Alegre, Brazil. (2008), 147 p.

[6] Golumbic, M. C.; Mintz, A.; Rotics, U. An improvement on the complexity of factoring read-once Boolean functions. Discrete Appl. Math. Vol. 156, n. 10. (2008), 1633-1636.

[7] Yoshida, H.; Ikeda, M.; Asada, K. Exact Minimum Logic Factoring via Quantified Boolean Satisfiability. International Conference on Electronics, Circuits, and Systems. (2006), 1065-1068.

[8] Poli, R. E. B.; Schneider, F. R.; Ribas R. P.; REIS, A. I. Unified Theory to Build Cell-Level Transistor Networks from BDDs. Symposium on Integrated Circuits and Systems Design. (2003), 199–204.

[9] Da Rosa Jr. L. S.; Marques, F. S.; Schneider, F.; Ribas, R. P.; Reis, A. I. A Comparative Study of CMOS Gates with Minimum Transistor Stacks. Symposium on Integrated Circuits and Systems Design. (2007), 93–98

[10] Prefuse.org. The Prefuse Visualization Toolkit. [Online] Avaliable: http://prefuse.org/ [Acessed: Oct. 12, 2010].

[11] Sentovich, E.; Singh, K., Lavagno; L., Moon; C., Murgai, R.; Saldanha, A., Savoj; H., Stephan, P.; Brayton, R.; and Sangiovanni-Vincentelli, A. SIS: A system for sequential circuit synthesis. Tech. Rep. UCB/ERL M92/41. UC Berkeley.