Extraction of Boolean Expressions from Transistor Networks

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

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

ABSTRACT

This paper presents a method to generate Boolean expressions from graphs that represent transistor networks. It uses graph compaction techniques to identify whether a transistor network has only series-parallel associations or also has bridge connections. The algorithm was implemented in Java language and integrated to the *Soptimizer* tool, that is an academic tool to apply optimizations in Boolean functions based on graphs. It has been validated using the set of 4-input functions of the pClass library set. Concerning series-parallel networks, all achieved results are compatible to the SIS tool, showing that the proposed algorithm is able to generate Boolean expressions from transistor networks. The SIS tool is the use reference as benchmark.

1. INTRODUCTION

Nowadays, most of the microelectronics market is mainly composed of Very Large Scale Integration (VLSI) circuits. These electronic devices can have billions of transistors in a single die. Due to the complexity of these circuits, Electronic Design Automation (EDA) tools play an important role in the design time. It reduces the time-to-market. Some of these tools work at the logic synthesis level, minimizing logic expressions through factorization procedures [1][2][3] in order to obtain equations with fewer literals. In factored form can be used to derive CMOS transistor networks. In this case, each literal of an equation is equivalent to a transistor. Therefore, when reducing the number of literals, the resultant network has fewer transistors. It leads to area saving and more efficiency on power consumption.

In a previous work, we have proposed a method to generate transistor networks from sum-of-products. This method uses factorization techniques to reduce the number of transistors needed to implement a given Boolean function. Each transistor network is represented by a graph. The implementation of this methodology resulted in the *Soptimizer* tool. The *Soptimizer* only works at the graph level, and it is not able to generate a Boolean expression that symbolizes the Boolean function represented by a graph. In order to close this gap, this paper presents an algorithm able to extract expressions from these graphs. The method proposed in *Soptimizer* has proven to be a good solution, since it is able to generate bridge networks with fewer transistors when compared to equivalent series-parallel networks.

The remaining of this paper is organized as follows. First, some basic concepts are reviewed in section 2. In section 3, we describe the proposed algorithm. Some results are shown in section 4. Finally, conclusions are presented in section 5.

2. BACKGROUND

For better understanding of this paper, some related concepts are reviewed on this section. **Sum-of-products** (SOPs) are canonical forms to represent Boolean functions through expressions. These expressions are not optimal concerning the number of literals. **Factorization** [1][2][3] is an optimization technique that can be applied to Boolean expressions (such as SOPs) aiming the minimization of some criteria. Usually, it aims the reduction on the number of literals in a given expression. There are algebraic and Boolean methods for factorization. Due to the algorithm complexity, algebraic are faster than Boolean methods. However, Boolean factorization can achieve better results.

Boolean functions can be represented in different ways. Graphs can be used to represent them. A **graph** is an ordered pair $G = (V, E)$ comprising a set V of **vertices** or **nodes** together with a set E of **edges** or **lines**, which are 2-element subsets of V. A **directed graph** is an ordered pair $D = (V, A)$ with V a set vertices, and A a set of ordered pairs of vertices, called **arcs**, **directed edges**, or **arrows**. An arc $a = (x, y)$ is considered to be directed from x to y; y is called the head and x is called the tail of the arc; y is said to be a **direct successor** of x, and x is said to be a **direct predecessor** of y. If a path leads from x to y, then y is said to be a **successor** of x and **reachable** from x, and x is said to be a **predecessor** of y.

Different kinds of data structures can be used to store graphs in computer systems. The **adjacency matrix** is one of them. This is an n by n matrix A, where n is the number of vertices in the graph. If there is an edge from a vertex x to a vertex y, then the element $a_{x,y}$ is 1 (or in general the number of xy edges), otherwise it is 0. In computing, this matrix makes it easy to find subgraphs, and to reverse a directed graph.

Besides Boolean function representation, graphs can also be used to represent transistor networks. In this case, each edge represents a given transistor and the vertices are points of connections among transistors.

Usually, CMOS transistor networks are constructed through series-parallel associations. In this case, each literal of a Boolean expression becomes an edge (transistor) in the graph representation. Therefore, the fewer literals in an expression the fewer transistors are needed to implement a Boolean function.

3. THE PROPOSED ALGORITHM

This section describes an algorithm to extract Boolean expressions from graphs that represent transistor networks. The proposed method is integrated with the *Soptimizer* tool. This tool factorizes SOPs (represented by graphs) using algebraic techniques applied to graphs. It reduces the number of edges required to represent a given Boolean function. As result, the *Soptimizer* tool presents a minimized graph.

In order to allow integration with other tools and other validation procedures, a Boolean expression is desired. The generation of a Boolean expression requires successive traversals of a graph from a terminal node to another. The algorithm performs these traversals applying graph compaction over the edges. On each traversal, every edge associated in series to an adjacent edge is "compacted" to a single edge. If there is no more series edges to be compacted, then the algorithm start to traverse the graph aiming the compaction of edges associated in parallel. When there are no more edges in parallel to be compacted, the algorithm is started again looking for series compaction. This iterative process stops when there is no more possibility of series and parallel compaction.

The graph compression steps are demonstrated in Fig.1. The initial graph is shown in Fig.1.a. The process begins by searching adjacent edges connected in series in sub-graphs. In this example, the first sets of edges to be compacted are connected to the terminal node *T0*. For instance, the edges "*!F"*, "*!E"* and "*!C"* are associated in series, and can be compacted to a single edge.
Therefore, the sub-graph composed by these edges is replaced by a new edge that actually is a list of edges. In this case, this new edge is referred as the product "*!F.!E.!C*". In a similar way, the edges "*F.E.C*" and "*D.B.A*" are created. This process is depicted in Fig.1.b.

After perform the first iteration of series compaction, the algorithm looks for parallel associations. In Fig.1.b, there are two sub-graphs in parallel. As shown in Fig.1.c, the edges "*!F.!E.!C*" and "*F.E.C*" can be compacted in a single edge that is referred as "*!F.!E.!C + F.E.C*". At this point, there are no more edges in parallel. The algorithm is restarted in order to look for edges associated in series. Hence, the graph in Fig.1.c can be reduced to the one in Fig.1.d. The final graph has a single edge that represents the Boolean function expressed by "*((!F.!E.!C+F.E.C).D.B.A)*".

Using graph compaction it is possible to identify bridge connections. Accordingly to [3], the minimal number of transistors to implement a Boolean function cannot be achieved by purely series-parallel networks. As an alternative, "*bridge*" transistor networks can be used. In this case, bridge connections are compacted through the compaction algorithm. Therefore, the final graph will have at least five edges on its construction (the minimal construction of a bridge network).

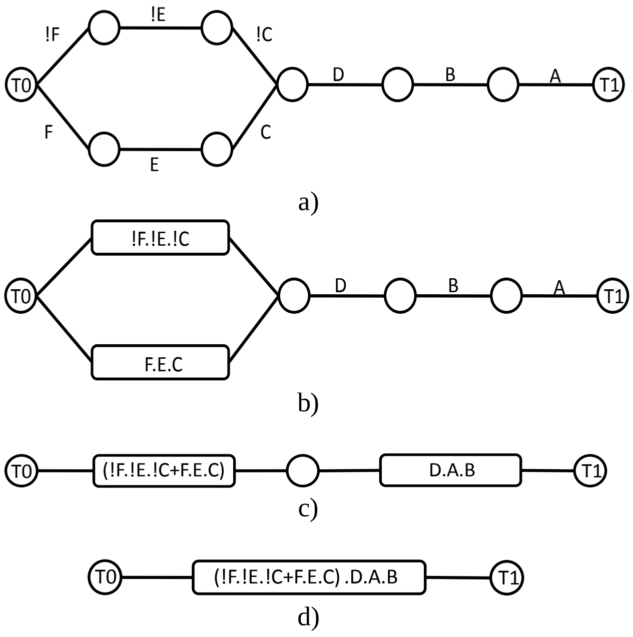


Figure1: Steps de Algorithm

The main objective of our method is to extract a Boolean expression from a graph. When a graph can be compacted into a single edge (meaning that there are only series-parallel connections), the Boolean expression is naturally achieved. That is not the case of bridge networks that requires a more sophisticated algorithm to construct the expression. An example of bridge network compaction can be seen in Fig.2. and Fig.3.

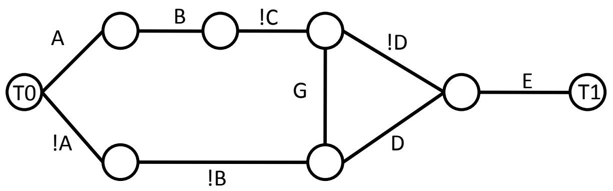


Figure 2: Initial Graph of Type Bridge.

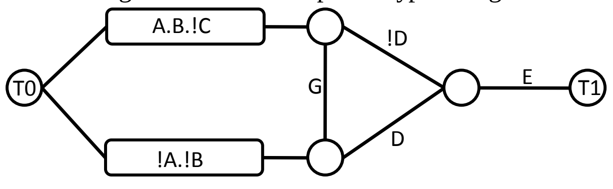
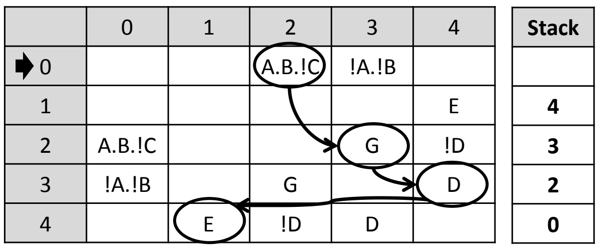
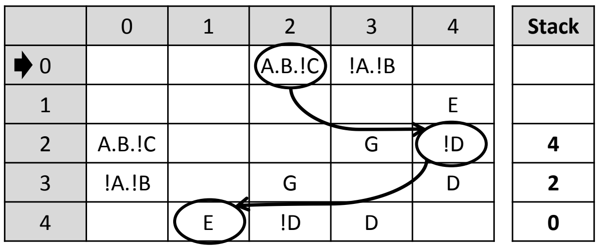


Figure 3: Graph Type Bridge Compacted.

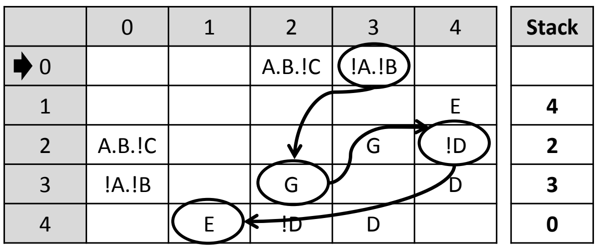
Considering the graph compaction of a bridge network, it is necessary to roam it in order to generate a Boolean expression that represents the whole graph. This final procedure is done using an adjacency matrix to represent the compacted graph. The expression can be extracted applying a depth-first-search algorithm on the adjacency matrix. The search starts on a terminal node and ends on another. Fig.4 shows the searches on the adjacency matrix that represents the compacted graph of Fig.3.



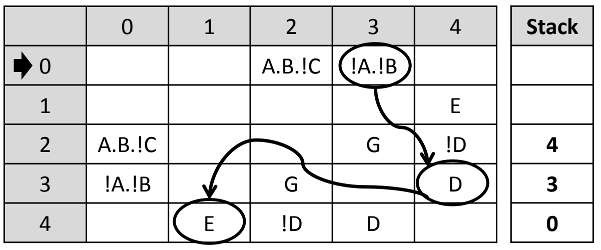
a)



b)



c)



d)

Figure 4: Paths generated from the graph traversal.

The depth-first-search starts on the terminal node "T0", i.e., on the index zero of the adjacency matrix. We use a stack to keep a traveled path. This stack stores the indexes of the adjacency matrix which represent nodes of a graph. Starting from the row zero, the algorithm looks for the first that represent a graph edge. At this point, column '2', which represents the edge "A.B.!C", is the first choice. Therefore, the index '2' is added into the stack. Through a recursive routine, the row '2' is the next point for search. In this row, the first column that indicates an edge is the column '0'. However, this row was already visited. This way, the next edge to be crossed to is the edge "G". Therefore, the next row is the one with index '3'. This process is repeated through the interconnected vertices, and it stops when the other terminal vertex is reached. In the examples of fig. 4, the index '4' represents the terminal node "T0". At the end, there will be a stack representing a path through both terminals.

Each constructed path symbolizes an expression which is a product of the edges expressions. In the example of Fig. 4, there are four resulting paths. A sum of the path products results in a Boolean expression that represents whole transistor network. Some products in a path are equivalent to products of other paths. Hence, associative techniques can be applied to reduce the number of literals of the final expression. In this example, the result of this process is the equation below:

$$A.B.!C.(!D.E+G.D.E)+!A.!B.(G.!D.E+D.E)$$

The Prefuse tool kit [4] was strongly used to debug the *Soptimizer* tool. This library provides a graphical framework for graph visualization. Later, we decided to provide a visual output to the user. An example of visual output is shown in Fig.5.

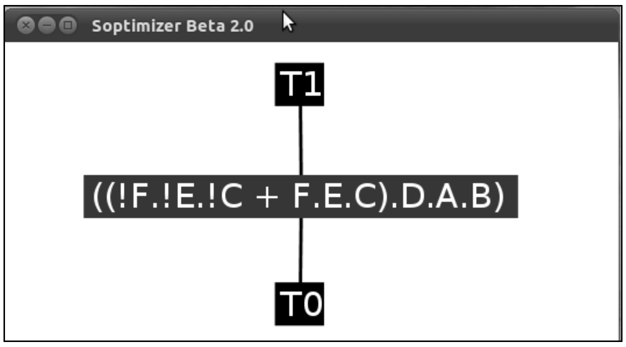


Figure 5: *Soptimizer* graphical interface.

4. EXPERIMENTAL RESULTS

Since the *Soptimizer* tool is implemented in Java language, the proposed algorithm was implemented through a set of Java classes that are already integrated into the tool and the Prefuse tool kit, which is used to visualize all generated graphs.

In order to validate the expressions generated by our algorithm, we have run a set of experiments on a computer with a Core2Duo processor and 4GB of RAM, running Ubuntu 10.10 - 64-bit. The first experiment validated 10 functions randomly chosen from the complete set of the 7-input *pClass* library. The second experiment has validated all 3,982 functions of the 4-input pClass functions library as benchmark.

All expressions were factorized through the *Soptimizer* tool and the resultant expressions were generated through the proposed algorithm in approximately 8 minutes. Since the expressions generated by our algorithm are equivalent to the input expressions, we assume that our algorithm is correct.

Notice that it also validates the *Soptimizer* tool results. This tool manipulates a graph constructed from an expression. Since the expressions generated from graphs resultant from the *Soptimizer* are equivalent to the original expressions, this means that the methods implemented by the tool are also correct.

Tab.1 presents a comparison of the SIS [5] factorization procedure and our method associated to the *Soptimizer* tool, considering a sub set of 10 functions of the 4-input pClass library. The second column shows the number of literals of each SOP that represents each function. The third column shows the number of literals of the factored forms generated by the SIS tool. The fourth column presents the number literals achieved by the *Soptimizer*. The last one presents the number of literals achieved by our algorithm.

Tab.1 – Literals count of factored forms

Function	**# literals in the SOP**	**#literals SIS**	**#literals Soptimizer**	**#literals Soptimizer Plus**
87	14	11	10	10
100	7	6	6	6
126	12	9	9	9
231	8	7	6	9
393	13	10	8	8
594	11	9	8	8
1183	13	10	8	8
2284	16	13	12	12
3622	12	10	9	9
3635	22	15	13	36

In the second experiment, all 4-input functions from pClass library have been synthesized using the *Soptimizer* tool. Furthermore, we have computed the number of transistors (number of edges) from the resulting graphs and the number of literals from the expressions generated by our method. As expected, in 2,192 functions (from 3,982), the numbers are the same when the resulting graph represents a series-parallel transistor network. However, when the resulting graph is a bridge network, the expressions have a higher number since the Boolean equations are expressed through the operators AND and OR that denotes series and parallel associations, respectively.

Besides, we have also compared our algorithm to factorization method implemented on SIS using the same pClass library. Our algorithm has achieved better results in a set of 107 functions. In 2,011 others cases, the results are the same. In the other 1,864, the SIS tool achieved better results. Currently, the *Soptimizer* tool is not able to eliminate non-sensitizable paths in a transistor network. Even though the resultant graphs are Boolean equivalent to the original expressions, when the *Soptimizer* manipulates the initial graph, it may introduce false paths in the network. It is demonstrated in Fig.6. The path going through edges "A", "G" and "!A" is not sensitizable given that the variable "A" appears in both polarities. Our method generates the expression "*A.(G.!A+D) + !B(G.D+!A)*", while it could be reduced to "*A.D+!B(G.D+!A)*".

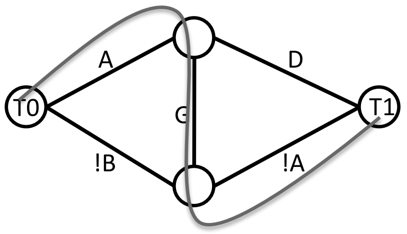


Figure 6: Example of a non-sensitizable path.

5. CONCLUSIONS AND FUTURE WORKS

This paper presented a method to extract Boolean expressions from graphs. This method fills a gap between *Soptimizer* and other tools that uses expression as input. Our algorithm is able to generate expression for series-parallel and bridge transistor networks represented by graphs. Boolean expressions do not have an operator to denote bridge connections. In this way, when expressed using the primitive operators, a lot of redundancies are added into the expression. Our algorithm is able to handle this problem, leading to optimal solutions. As future work we intent to eliminate non-sensitized paths of the transistor networks generated by the Soptimizer. It can be done by using Boolean methods during the graph manipulation. This way, we can reduce the transistor count and achieve smaller expressions in terms of literals.

6. REFERENCES

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

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

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

[4] PREFUSE.ORG. The Prefuse Visualization Toolkit. [Online] Avaliable: http://prefuse.org/ [Acessed: Mar. 25, 2010].

[5] SENTOVICH, E. et al. SIS: A system for sequential circuit synthesis. Tech.Rep. UCB/ERL M92/41. UC Berkeley, Berkeley. (1992).