Novel Methods for Efficient Realization of Logic Functions Using Switching Lattices

Levent Aksoy and Mustafa Altun

Abstract—Two-dimensional switching lattices including four-terminal switches are introduced as alternative structures to realize logic functions, aiming to outperform the designs consisting of one-dimensional two-terminal switches. Exact and approximate algorithms have been proposed for the problem of finding a switching lattice which implements a given logic function and has the minimum size, *i.e.*, a minimum number of switches. In this article, we present an approximate algorithm, called JANUS, that explores the search space in a dichotomic search manner. It iteratively checks if the target function can be realized using a given lattice candidate, which is formalized as a satisfiability (SAT) problem. As the lattice size and the number of literals and products in the given target function increase, the size of a SAT problem grows dramatically, increasing the run-time of a SAT solver. To handle the instances that JANUS cannot cope with, we introduce a divide and conquer method called MEDEA. It partitions the target function into smaller sub-functions, finds the realizations of these sub-functions on switching lattices using JANUS, and explores alternative realizations of these sub-functions which may reduce the size of the final lattice. Moreover, we describe the realization of multiple functions in a single lattice. Experimental results show that JANUS can find better solutions than the existing approximate algorithms, even than the exact algorithm which cannot determine a minimum solution in a given time limit. On the other hand, MEDEA can find better solutions on relatively large size instances using a little computational effort when compared to the previously proposed algorithms. Moreover, on instances that the existing methods cannot handle, MEDEA can easily find a solution which is significantly better than the available solutions.

Index Terms—emerging technologies, four-terminal switch, switching lattice, logic synthesis, satisfiability, binary search.

1 Introduction

As the Moore's law [1], *i.e.*, the number of transistors doubles in the given chip every two years, has been reaching its limit [2], researchers have been exploring new technologies and structures. Nanotechnology, which aims to build materials and devices on the scale of atoms and molecules, has been an emerging technology to tackle the limitations of the conventional CMOS technology [3]. Recent years have seen successful design of memory cores and programmable logic arrays and interconnects using nanotechnologies [4], [5]. Moreover, architectures and structures for the nano-electronic computation, realizing simple logic gates, such as NAND and NOR, and implementing complex logic circuits, such as adders and microprocessors, have been introduced [6], [7], [8], [9], [10].

As shown in [11], a four-terminal switch, developed especially for the cross-points of nanoarrays, can be used to realize logic functions. As illustrated in Fig. 1a, if its control input x has the value 0, all its terminals are disconnected (OFF). Otherwise, they are connected (ON). In a switching lattice, that is formed as a network of four-terminal switches, each switch is connected to its horizontal and vertical neighbors. A 3×3 switching network is shown in Fig. 1b where $x_1, \ldots, x_9$ denote the control inputs of switches. The lattice function, whose inputs are the control inputs of switches, evaluates to 1 if there is a path between the top and bottom plates of the lattice and is written as the sum of products of control inputs of switches in each path. In a lattice with

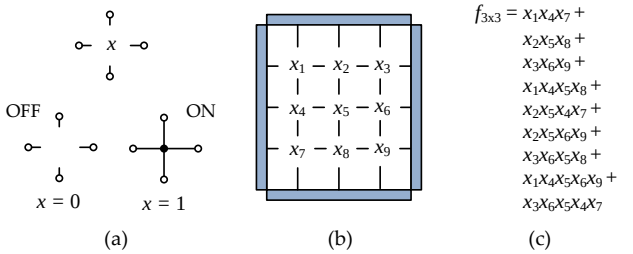


Fig. 1. (a) Four-terminal switch; (b) the 3×3 four-terminal switching network; (c) the 3×3 switching lattice function.

four-terminal switches, a path is a sequence of switches connected by taking horizontal and vertical moves. Fig. 1c shows the 3×3 lattice function $f_{3\times3}$. A lattice function is unique and does not include any redundant products, *e.g.*, a possible path $x_3x_2x_1x_4x_7$ in the 3×3 switching network is eliminated by the path $x_1x_4x_7$.

One of the main advantages of using switching lattices is its reconfigurability. As shown in this article, there exists a switching lattice that can be used to realize all logic functions with a certain number of variables. Its another advantage is to reduce the number of switches when compared to the conventional two-terminal switches, such as field-effect transistors, in a given design. As an example, consider $f(a,b,c,d) = \sum(2,3,4,8,9,12,14,15)$ which can be written as $f = \overline{a}\overline{b}c + abc + a\overline{b}\overline{c} + b\overline{c}\overline{d}$. Taking into account the most commonly used CMOS technology, its straightforward two-level realization using AND and OR gates, as shown in Fig. 2a, requires 42 CMOS transistors without counting the ones for the inverters of primary inputs. The number of CMOS transistors can be reduced further as follows: i) apply a state-of-art logic synthesis tool to the given logic function using a synthesis script; ii) map the

The authors are with the Emerging Circuits and Computation (ECC) Group, Department of Electronics and Communication Engineering, Istanbul Technical University, Maslak, 34469, Istanbul, Turkey (e-mail: aksoyl, altunmus@itu.edu.tr).

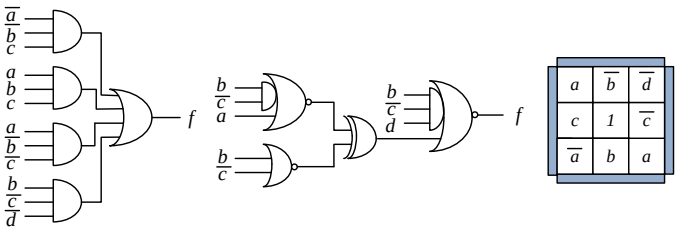


Fig. 2. Realizations of $f = \overline{a}\overline{b}c + abc + a\overline{b}\overline{c} + b\overline{c}\overline{d}$: (a) two-level; (b) considering further reduction in the number of CMOS transistors; (c) using the 3×3 lattice with four-terminal switches.

design into gates of a given library where the cost value of a gate is defined in terms of the number of CMOS transistors required to build the gate; iii) compute the design complexity in terms of the number of required transistors; and iv) repeat this process for a number of synthesis scripts and keep the design with the least complexity. In this work, we used *ABC* [12] as a logic synthesis tool, its 17 synthesis scripts, and a gate-library which is the extended version of the *mcnc.genlib* library. Fig. 2b presents the solution found for our example which requires 26 CMOS transistors without counting the ones for the inverter of a primary input. On the other hand, the realization of f using the 3×3 lattice[1], shown in Fig. 2c, needs 9 four-terminal switches.

It is shown in [13] that a four-terminal switch can be developed using the conventional CMOS technology and can be used to form a switching lattice. Fig. 3a and b present two CMOS-compatible devices introduced for the development of the four-terminal switch whose gate structure is square and cross, respectively. In this figure, T1, T2, T3, and T4 stand for the four terminals. Further details on the technology development can be found in [13]. It is confirmed through technology simulations that these devices behave as a four-terminal switch. During the fabrication process in implementation of a four-terminal switch, it is observed that the compact structure of a lattice, shown in Fig. 3c, has a potential to yield significant savings in area. This is mainly because the excessive area due to the placement and routing of transistors and gates [14] does not occur in the design of a switching lattice and the realization of a logic function using a lattice generally needs less number of switches than that in the conventional CMOS realization as shown in Fig 2.

In recent years, many algorithms have been introduced to realize a logic function on a switching lattice using the fewest number of switches [15], [16], [17], [18], [19], [20]. However, while the exact algorithm [15] can only handle relatively small instances, the approximate algorithms [16], [17], [18], [19] can find solutions that are far away from the minimum. Hence, in this article, we present the approximate algorithm of [20], called JANUS, that can find better solutions using less computational effort when compared to the existing approximate algorithms. It improves the initial boundaries of the search space significantly and uses an efficient SAT formulation for the problem of finding if a given target function can be realized using a given switching

1. Keeping the same order in the products and variables of $f_{3\times3}$ in Fig. 1c, the function realized by the lattice can be given as $f = ac\overline{a} + \overline{b}1b + \overline{d}\overline{c}a + ac1b + \overline{b}1c\overline{a} + \overline{b}1\overline{c}a + \overline{d}\overline{c}1b + ac1\overline{c}a + \overline{d}\overline{c}1c\overline{a}$. After the application of Boolean algebra laws, it can be given as $f = a\overline{c}\overline{d} + abc + \overline{a}\overline{b}c + a\overline{b}\overline{c} + b\overline{c}\overline{d}$. Note that the $a\overline{c}\overline{d}$ product can also be eliminated by the consensus law.

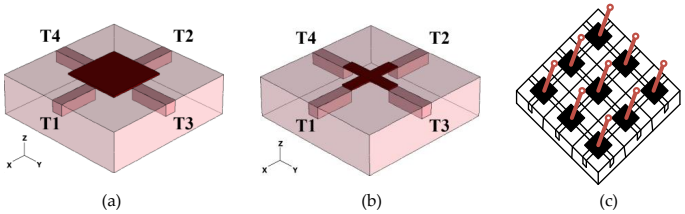


Fig. 3. (a) Square-shaped four-terminal switch; (b) cross-shaped four-terminal switch; (c) illustration of the 3×3 lattice including four-terminal switches where the vertical wires denote the control inputs.

lattice. However, as the number of literals and products in a logic function increases, the SAT problem complexity goes beyond the reach of state-of-art SAT algorithms. In order to handle such complex instances that JANUS finds hard to cope with, we introduce a divide and conquer algorithm called MEDEA. It is observed that MEDEA can find solutions using significantly less time than JANUS and its solutions are better than those of the previously proposed approximate algorithms and close to those of JANUS on logic functions with a small number of products. It is shown on the logic functions, which the existing algorithms cannot handle, that the solutions of MEDEA have significantly less number of switches than the best solutions found so far. Finally, in this article, we present an efficient way of realizing multiple functions in a single lattice using the proposed methods.

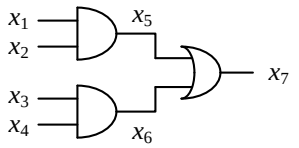
The rest of this article is organized as follows: Section 2 presents the background concepts, problem definitions, and related work. The proposed algorithms are introduced and the realization of multiple functions in a single lattice is described in Section 3. Experimental results are shown in Section 4 and finally, Section 5 concludes the article.

2 Background

2.1 Preliminaries

A *Boolean logic function*, $f : \mathcal{B}^r \rightarrow \mathcal{B}$, over r variables $x_1, \ldots, x_r$ maps each truth assignment in $\mathcal{B}^r$ to 0 or 1. The logic function f in *sum of products* (SOP) form on r variables is a disjunction of s products $p_1, \ldots, p_s$, where a *product* $p_i = l_1 \cdot l_2 \cdot \ldots \cdot l_j$, $i \leq s$ and $j \leq r$, is a conjunction of literals. A *literal* l_j, $j \leq r$, is either a variable x_j or its complement $\overline{x_j}$. A product is an *implicant* if and only if it evaluates f to 1 and it is a *prime implicant* if it is an implicant and there exist no other implicants whose literals are subset of its literals. In an *irredundant SOP* (ISOP) form of f, every product is a prime implicant and no product can be deleted without changing f. The *degree* of f is the maximum number of literals in the products of f. The dual of f can be computed as $f^D(x_1, \ldots, x_r) = \overline{f}(\overline{x_1}, \ldots, \overline{x_r})$ and can be found by interchanging the AND and OR operations and the constants 0 and 1 as well. The Shannon expansion of f is given as $f = \overline{x_i} f_{\overline{x_i}} + x_i f_{x_i}$, $1 \leq i \leq r$, where $f_{\overline{x_i}}$ and f_{x_i} stand for the negative and positive co-factors of f which are computed by replacing x_i with logic 0 and 1 in f, respectively and include maximum $r - 1$ variables.

A Boolean function φ in *product of sums* (POS) form on r variables is a conjunction of t clauses $c_1, \ldots, c_t$, where a *clause* $c_i = l_1 + l_2 + \ldots + l_j$, $i \leq t$ and $j \leq r$, is a disjunction of literals. If a literal of a clause is set to 1, the clause is satisfied. If all literals of a clause are set to 0, the clause is unsatisfied. The *satisfiability* (SAT) problem is to find an assignment to



$$\varphi = (x_1+\overline{x_5})\cdot(x_2+\overline{x_5})\cdot(\overline{x_1}+\overline{x_2}+x_5)\cdot (x_3+\overline{x_6})\cdot(x_4+\overline{x_6})\cdot(\overline{x_3}+\overline{x_4}+x_6)\cdot (\overline{x_5}+x_7)\cdot(\overline{x_6}+x_7)\cdot(x_5+x_6+\overline{x_7})$$

Fig. 4. A combinational network and its POS formula.

the variables of a function φ in POS form that makes φ to be equal to 1 or to prove that φ is equal to 0. The SAT problem is NP-complete [21].

A *combinational circuit* is a directed acyclic graph with nodes and directed edges corresponding to logic gates and wires connecting the gates, respectively. The POS formula of a combinational circuit is the conjunction of POS formula of each gate which denotes the valid input-output assignments to the gate. The derivation of POS formulas of basic logic gates can be found in [22]. Fig. 4 shows a combinational circuit and its formula in the POS form.

The *0-1 integer linear programming* (ILP) problem is the minimization or the maximization of a linear objective function subject to a set of linear constraints and is generally defined as follows[2]:

$$\text{minimize} \quad \mathbf{w}^T \cdot \mathbf{y} \tag{1}$$

$$\text{subject to} \quad \mathbf{A} \cdot \mathbf{y} \geq \mathbf{b}, \quad \mathbf{y} \in \{0,1\}^k \tag{2}$$

In the objective function given in Eqn. 1, each weight w_i associated with each variable y_i is an integer value, where $1 \leq i \leq k$. In Eqn. 2, $\mathbf{A} \cdot \mathbf{y} \geq \mathbf{b}$ denotes the set of j linear constraints, where $\mathbf{b} \in \mathbb{Z}^j$ and $\mathbf{A} \in \mathbb{Z}^j \times \mathbb{Z}^k$.

2.2 Switching Lattices

2.2.1 Computing the Lattice Function

In a switching lattice, a *four-connected path* is a sequence of switches connected by taking horizontal and vertical moves and a lattice function includes all irredundant four-connected paths between the top and bottom plates. An *eight-connected path* is generated by taking diagonal moves in addition and the dual of a lattice function consists of all irredundant eight-connected paths between the left and right plates [11]. For example, the dual of the lattice function $f_{3\times3}$ given in Fig. 1c has 17 products all with three variables[3]. Thus, finding a realization of a target function on an $m \times n$ switching lattice considering the four-connected paths between the top and bottom plates can also be done by finding a realization of the dual of target function on the same $m \times n$ lattice considering the eight-connected paths between the left and right plates. This is because taking the dual of a logic function twice leads to the function itself.

Table 1 presents the number of products in the $m \times n$ lattice function and its dual at the top and bottom of each entry, respectively, where $2 \leq m, n \leq 8$. Similarly, Table 2 shows the degree of the $m \times n$ lattice function and its dual at the top and bottom of each entry, respectively.

2. The maximization objective can be easily converted to a minimization objective by negating the objective function. Less-than-or-equal and equality constraints are handled by the equivalences, $\mathbf{A} \cdot \mathbf{y} \leq \mathbf{b} \Leftrightarrow -\mathbf{A} \cdot \mathbf{y} \geq -\mathbf{b}$ and $\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \Leftrightarrow (\mathbf{A} \cdot \mathbf{y} \geq \mathbf{b}) \wedge (\mathbf{A} \cdot \mathbf{y} \leq \mathbf{b})$, respectively.

3. $f^D_{3\times3} = x_1x_2x_3+x_1x_2x_6+x_1x_5x_3+x_1x_5x_6+x_1x_5x_9+x_4x_2x_3+x_4x_2x_6+x_4x_5x_3+x_4x_5x_6+x_4x_5x_9+x_4x_8x_6+x_4x_8x_9+x_7x_5x_3+x_7x_5x_6+x_7x_5x_9+x_7x_8x_6+x_7x_8x_9$.

TABLE 1
Number of products in the $m \times n$ lattice function and its dual.

m/n	2	3	4	5	6	7	8
2	2 4	3 8	4 16	5 32	6 64	7 128	8 256
3	4 7	9 17	16 41	25 99	36 239	49 577	64 1393
4	6 10	17 28	36 78	67 216	118 600	203 1666	344 4626
5	10 13	37 41	94 139	205 453	436 1497	957 4981	2146 16539
6	16 16	77 56	236 250	621 1018	1668 4286	4883 18730	14880 81192
7	26 19	163 73	602 461	1905 2439	6562 13833	26317 86963	110838 539537
8	42 22	343 92	1528 872	5835 6004	25686 45788	139231 421182	797048 3779226

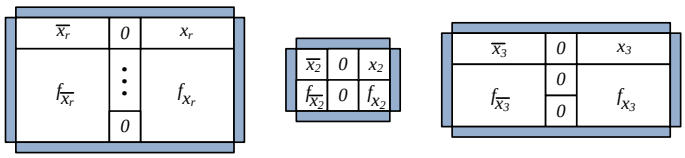
TABLE 2
Degree of the $m \times n$ lattice function and its dual.

m/n	2	3	4	5	6	7	8
2	2 2	2 3	2 4	2 5	2 6	2 7	2 8
3	4 2	5 3	6 4	7 5	8 6	9 7	10 8
4	5 2	6 4	7 6	10 7	11 9	12 11	13 12
5	7 2	9 5	11 7	13 9	16 12	18 15	20 18
6	8 2	10 6	12 8	16 11	19 14	22 17	24 20
7	10 2	13 7	16 9	19 13	23 15	26 19	29 22
8	11 2	14 8	17 10	22 15	26 17	30 22	33 26

Observe from these tables that as the number of rows and columns increases, the number of products and degree of lattice functions increase dramatically, pointing out the lattices that can be used to realize a rich variety of logic functions. Note that a lattice function may have fewer or more products than its dual, *e.g.*, 2×4 and 8×4 lattices as shown in Table 1. On the other hand, the degree of a lattice function is greater than that of the dual of a lattice function, except when m is equal to 2. For the lattices with the same size, there exists a wide range of functions with different number of products and degrees. As an example, while $f_{3\times8}$ includes 64 products with a degree of 10, $f_{6\times4}$ has 236 products with a degree of 12. This is also true for the lattices with sizes very close to each other. For example, while $f_{5\times7}$ has 957 products with a degree of 18, $f_{6\times6}$ contains 1668 products with a degree of 19. We note that not only the degree and the number of products, but also the number of literals in each product is important while realizing a logic function on a switching lattice.

2.2.2 Reconfigurable Switching Lattices

Similar to the look-up tables in field programmable gate arrays [23], a switching lattice can be used as a reconfigurable block which can realize all logic functions with r variables. An upper-bound on the size of such a switching lattice can be determined based on the Shannon expansion in a recursive manner. Fig. 5a presents the realization of the Shannon expansion of a logic function, $f = \overline{x_i} f_{\overline{x_i}} + x_i f_{x_i}$, using a switching lattice when i is r. Since any logic function including 1 variable requires a 1×1 lattice, any logic function with 2 variables can be realized using the 2×3 lattice as shown in Fig 5b. Note that the co-factors $f_{\overline{x_2}}$ and f_{x_2} include maximum one variable, *i.e.*, x_1. Thus, any logic function with 3 variables can be realized using a 3×7 lattice as shown in Fig 5c where the co-factors $f_{\overline{x_3}}$ and f_{x_3}, which



(a) (b) (c)

Fig. 5. (a) Realization of $f = \overline{x_r} f_{\overline{x_r}} + x_r f_{x_r}$ using a switching lattice; lattices which can be used to realize any logic function including r variables: (b) 2×3 lattice when r is 2; (c) 3×7 lattice when r is 3.

include maximum two variables, are realized as given in Fig 5b. Hence, to realize any logic function with r variables, an $r \times 2^r - 1$ lattice can be used. Observe that the size of such a lattice grows exponentially as r increases.

On the other hand, by trying all possible logic functions, the switching lattice with the minimum size that can be used to realize any logic function with 2, 3, and 4 variables is found to be the 2×2, 3×3, and 3×6 lattice, respectively. Hence, using the Shannon expansion in a recursive manner, the size of the lattice, which can be used to realize any logic function with r variables when $r \geq 5$, can be formulated as $(r-1) \times (6 \cdot 2^{r-5} + 2^{r-2} - 1)$. For example, any logic function with 5 and 6 variables can be realized using the 4×13 and 5×27 lattice using the new formula rather than the 5×31 and 6×63 lattice, respectively.

2.3 Problem Definitions

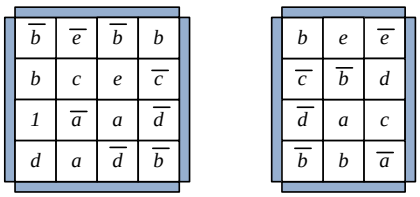
Realization of a logic function using a switching lattice can be obtained by simply mapping the appropriate literals of this target function and/or constant values (0 and 1) to the control inputs of switches. The *lattice mapping* (LM) problem, is defined as: given a target function f and an $m \times n$ lattice function $f_{m \times n}$, find the appropriate assignments to the lattice variables such that the target function f can be realized on the $m \times n$ lattice or prove that there exists no such assignment. The LM problem was shown to be *NP-complete* in [15]. As an example, consider $f(a,b,c,d,e) = \sum(1,6,7,14,17,21,24,25,30)$ which can be written as $f = ab\overline{c}\overline{d} + \overline{a}bcd + bcd\overline{e} + \overline{b}\overline{c}de + a\overline{b}de$. It can be realized using the 4×4 lattice as shown in Fig. 6a and the 5×3 lattice, but cannot be realized using the 3×3 lattice.

In the realization of a logic function using a switching lattice, the design complexity is determined as the number of switches, *i.e.*, the lattice size. Thus, the *lattice synthesis* (LS) problem, is defined as: given the target function f, find an $m \times n$ lattice such that there exists an appropriate assignment to the lattice variables, realizing f, and m times n is minimum. Returning to our example, Fig. 6b presents the realization of the target function on a lattice with the minimum size of 4×3.

2.4 Related Work

Over the years, logic structures and arrays, that provide regularity, reconfigurability, and ease of design, have been introduced to realize logic functions [24], [25], [26]. A structure similar to the switching lattices can be found in [26] where a rectangular logic array is composed of cells, each connecting to its neighbors. Each cell, which is a three-input and two-output circuit, connects the up and right cell to the down and left one based on the input values.

In order to realize logic functions using switching lattices, the recursive method of [27] maps logic functions



(a) (b)

Fig. 6. Realizations of $f = ab\overline{c}\overline{d} + \overline{a}\overline{b}cd + bcd\overline{e} + \overline{b}\overline{c}\overline{d}e + a\overline{b}\overline{d}e$ using switching lattices: (a) 4×4; (b) 4×3.

onto nanowire based crossbar architectures with different topologies rather than using a regular switching lattice structure. In [11], an upper bound on the lattice size is computed by finding common literals in the products of the target function and its dual and the lower bound on the lattice size is obtained based on the minimum degrees of the target function and its dual. The exact algorithm of [15] explores the search space using a binary search technique in between the lower and upper bounds computed in [11]. During this search, for each possible lattice, an LM problem is generated. The LM problem is encoded as a quantified Boolean formula (QBF) problem, the QBF constraints are converted to SAT clauses, and a solution is found using a SAT solver. The approximate method of [15] restricts this exact QBF formulation by allowing the paths to include only the literals in the given products, reducing the size of SAT problems. However, since the approximate method may yield a non-optimal solution, it may solve more LM problems than the exact method. The technique of [17] synthesizes the D-reducible form of a target function, which is composed of two small sub-functions, on a switching lattice. In [17], these sub-functions are synthesized using the algorithm of [15] and then, their solutions are merged into a single lattice. Note that not every logic function can be represented in the D-reducible form. Similarly, the methods of [16], [18] exploit the p-circuits and autosymmetric form of a target function, respectively and use the algorithms of [11], [15] to find a solution on the decomposed sub-functions. In [18], the target function is synthesized with multiple lattices sharing the common ones, but adding extra logic gates which may not be desirable due to the wires between these gates and lattice control inputs. The method of [19] determines a number of promising lattice candidates and uses a method of [15] to find if one of these lattices leads to a solution.

3 Proposed Algorithms

In this section, we initially describe the approximate algorithm and then, the divide and conquer method, both developed for the realization of a single logic function using a switching lattice. Finally, we present the method developed for the realization of multiple functions on a single lattice.

3.1 JANUS: An Approximate Algorithm

JANUS takes the target function f as an input and finds its implementation on a switching lattice as described below:

1) Determine the *lower bound* (*lb*) and *upper bound* (*ub*) of the LS problem in terms of the number of switches.
2) If $lb = ub$, return the solution found while computing *ub*.

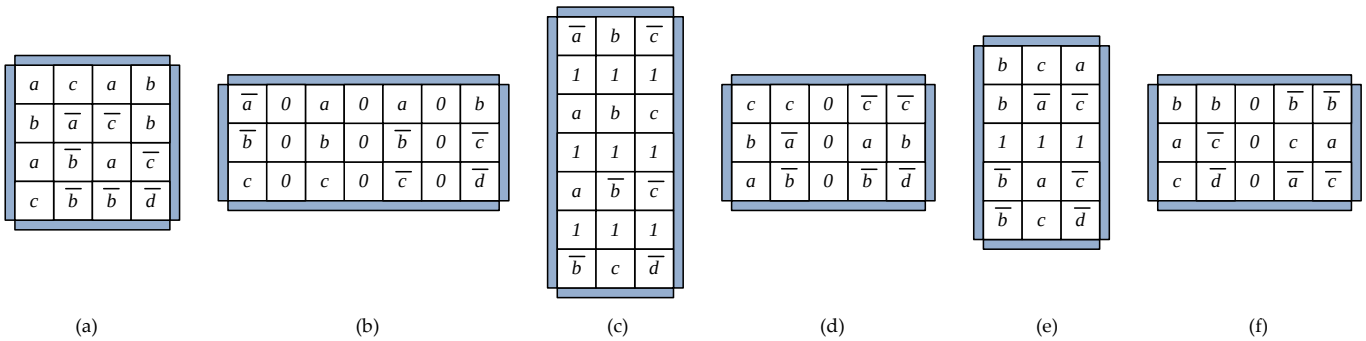


Fig. 7. Computing the upper bound of $f = \bar{a}\bar{b}c + abc + a\bar{b}\bar{c} + b\bar{c}\bar{d}$ using different methods: (a) DS; (b) PS; (c) DPS; (d) IPS; (e) IPDS; (f) DS.

3) Determine the *middle point* as $mp = \lfloor (lb + ub)/2 \rfloor$ and generate a set of lattice candidates C.
4) For each lattice candidate in C, generate the related LM problem and check if f can be realized using the lattice candidate. If there exists a solution to the LM problem, set ub to mp and go to Step 6.
5) If there are no solutions for all lattice candidates in C, set lb to $mp + 1$.
6) If $lb < ub$, go to Step 3. Otherwise, return the solution.

In following, we introduce the methods that compute the initial lower and upper bounds of the LS problem (Step 1), describe how the lattice candidates are generated (Step 3), present the encoding of the LM problem as a SAT problem (Step 4), and analyze the SAT problem complexity.

3.1.1 Computing the Initial Lower and Upper Bounds

The computation of the initial lower bound of the LS problem takes into account the products of the lattice and target functions and their duals and is described as follows:

1) Find the ISOP forms of the target function and its dual, both including a minimum number of products obtained using a logic minimization tool.
2) Set the lattice size ls equal to 1.
3) Obtain all possible lattice candidates with the size ls and apply the *structural check* procedure to each lattice candidate. If the structural check is passed, return the lower bound computed as ls. Otherwise, try another lattice candidate.
4) If there exist no lattices with the lattice size ls that pass the structural check, increase ls by 1 and go to Step 3.

In the structural check procedure, we check for each product of the target function with j literals if the lattice candidate function has a different product with a number of literals greater than or equal to j. If it is so, similarly, we check if each product of the dual of target function is covered by a different product of the dual of lattice candidate function. In this work, we use *espresso* [28] as a logic minimization tool. Consider our example in Fig. 2 and assume that ls is equal to 8. Note that the dual of the target function $f = \bar{a}\bar{b}c + abc + a\bar{b}\bar{c} + b\bar{c}\bar{d}$ is $f^D = \bar{a}b\bar{c} + abc + a\bar{b}\bar{c} + \bar{b}c\bar{d}$. For the lattice size ls equal to 8, there exist four possible lattices, *i.e.*, 1×8, 2×4, 4×2, and 8×1. There are eight products with only one variable in $f_{1\times8} = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$, $f_{2\times4} = x_1x_5 + x_2x_6 + x_3x_7 + x_4x_8$ has four products each including only two variables, and similarly, $f^D_{4\times2} = x_1x_2 + x_1x_4 + x_3x_2 + x_3x_4 + x_3x_6 + x_5x_4 + x_5x_6 + x_5x_8 + x_7x_6 + x_7x_8$ has ten products each including only two variables, and finally, there exists only one product in $f_{8\times1} = x_1x_2x_3x_4x_5x_6x_7x_8$. Hence, the structural check confirms that all these lattices cannot be used to realize the target function. Thus, ls is increased to 9 and it is found that all products of the target function and its dual are covered by the products of the 3×3 lattice function and its dual, respectively. Hence, the initial lower bound is determined as 9. Similarly, for our example in Fig. 6, where the dual of the target function $f = ab\bar{c}\bar{d} + \bar{a}\bar{b}cd + bcd\bar{e} + \bar{b}\bar{c}\bar{d}e + a\bar{b}\bar{d}e$ is $f^D = c\bar{d} + a\bar{c}d + a\bar{b}d + \bar{b}\bar{c}d + \bar{a}b\bar{d} + bde + \bar{b}\bar{c}\bar{e}$, the initial lower bound is computed as 9.

There exist three efficient methods used to find an initial upper bound. The dual production (DP) [11] method realizes a target function using an $u \times v$ lattice, where v and u are the number of products in the target function and its dual, respectively. In the product separation (PS) method [15], the v products of a target function are placed on the columns of a lattice each separated by a column full of zeros, filling the unspecified entries of the lattice with constant 1. Thus, a solution with a $\delta \times (2v - 1)$ lattice is found where δ is the degree of the target function. In the dual product separation (DPS) method [19], the u products of the dual of target function are placed on the rows of a lattice separated by a row full of ones, filling the unspecified entries of the lattice with constant 0. Thus, a solution with a $(2u - 1) \times \gamma$ lattice is found where γ is the degree of the dual of target function.

For our example in Fig. 2, as shown in Fig. 7a-c, the DP, PS, and DPS methods find a solution with the 4×4, 3×7, and 7×3 lattice, respectively. For our example in Fig. 6, we note that the solution of the DP, PS, and DPS methods includes the 7×5, 4×9, and 13×3 lattice, respectively.

However, the PS and DPS methods can be modified to decrease the required number of switches by reducing the number of isolation columns and rows between the products, *i.e.*, the ones full of constant 0 and 1, respectively. We developed the improved version of the PS method, called IPS, as follows:

1) Find the products with two literals, put each of them side by side by placing one literal on the δ^{th} row and the other on the other rows of that column, and finally, add an isolation column if there are products with more than 2 literals.
2) For each product, p_i, find another product, p_j, both including more than 2 literals, so that the pair function $f_p = p_i + p_j$ can be realized using a $\delta \times 2$ lattice, *i.e.*, without using an isolation column. Such a product is found if the number of products in the dual of the pair function, f^D_p, is less than or equal to δ. If such a product, p_j, exists, add the realization of the pair

function into the lattice. Otherwise, add p_i into the lattice. If there exists other product(s) with more than 2 literals to consider, add an isolation column.

3) Find the products with a single literal and if there exist isolation columns, for each product, place its literal on every row of an isolation column full of constant 0. If the number of isolation columns is less than the number of such products, put each remaining product side by side by placing its literal on every row of that column.

The improved version of the DPS method, called IDPS, can be developed similarly. It consists of the second step of the IPS method, where the products including more than 1 literal are considered, and its third step.

For our example in Fig. 2, as shown in Fig. 7d and e, the IPS and IDPS methods find a solution with the 3×5 lattice and the 5×3 lattice, respectively. For our example in Fig. 6, we note that the solution of the IPS and IDPS methods includes the 4×7 and 10×3 lattice, respectively.

The run-time complexity of these methods is bounded by the complexity of finding the dual of a logic function which can be obtained using a little computational effort on functions even with a large number of products by the state-of-art logic minimization tools. For example, the dual of a logic function with 14 variables, 90 products (the largest in our experiments), and a degree value of 9, is found less than a second using *espresso*. However, we observed in our experiments that the solutions of these methods can be far away from the minimum on the logic functions including more than 10 products. Hence, a divide and synthesize (DS) method, which is based on our divide and conquer method described in Section 3.2, is developed. The DS method consists of three main steps described as follows:

1) Partition the products in the target function f into two sub-functions f_1 and f_2 such that $f = f_1 + f_2$, where f_1 and f_2 have a number of products close to each other and a minimum number of literals.
2) Apply JANUS to these sub-functions and add its solutions into a lattice using a single isolation column.
3) For each sub-function, explore alternative realizations with a small number of rows and columns.

The partitioning of products into two sub-functions in the first step is formulated as a 0-1 ILP problem and solved using a 0-1 ILP solver. To explore alternative realizations of sub-functions in the third step, we initially compute the size and number of rows of the lattice found at the second step, denoted as the best cost bc and best row br, respectively. Then, we apply the following procedure when $br \geq 3$.

1) For each solution of a sub-function with an $m \times n$ lattice, where $m \geq br > 3$, check if a $(br-1) \times k$ lattice, where $k > n$, can be used to synthesize this sub-function. Note that k initially set to n is incremented by 1 till the bc value is exceeded or a solution is found.
2) For each solution of a sub-function with an $m \times n$ lattice, where $m < br - 1$, check if this sub-function can be realized using an $(br-1) \times k$ lattice, where $k < n$. Note that k initially set to n is decremented by 1 till there exists no solution.
3) If a solution with a size less than bc is found, update the lattice and its cost.

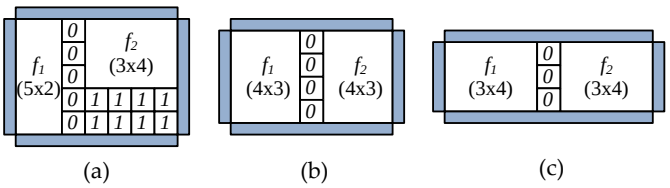


Fig. 8. An illustrative example for the realization of $f = f_1 + f_2$: (a) the 5×7 lattice; (b) the 4×7 lattice; (c) the 3×9 lattice.

4) If $br > 3$, decrease br by 1 and go to Step 1. Otherwise, return the lattice.

As an illustrative example, assume that after the target function is partitioned into two sub-functions f_1 and f_2, the solutions of JANUS on these sub-functions include the 5×2 and 3×4 lattices, respectively. As shown in Fig. 8a, the lattice, that realizes the target function $f = f_1 + f_2$, is 5×7, where bc and br are 35 and 5, respectively. If both f_1 and f_2 are realized using the 4×3 lattice, f can be realized using the 4×7 lattice as shown in Fig. 8b. Furthermore, if f_1 is realized using the 3×4 lattice, then f can be realized using the 3×9 lattice as shown in Fig. 8c.

For our example in Fig. 2, the DS method finds a solution using the 3×5 lattice as shown in Fig. 7f. For our example in Fig. 6, we note that the DS method finds a solution using the 4×5 lattice.

In JANUS, the initial upper bound is computed as the minimum of solutions of all these methods. Thus, the upper bounds on our examples in Fig. 2 and 6 are computed as 15 and 20, respectively.

Observe that the methods used to compute the initial lower and upper bounds of the search space consider a single ISOP form of a logic function. However, a logic function may have a number of ISOP forms with different products. Thus, better initial lower and upper bounds can be found when all ISOP forms are considered, but increasing the computational effort.

3.1.2 Generation of Lattice Candidates

While exploring the search space of the LS problem in a binary search manner, the middle point mp is computed as $\lfloor (lb+ub)/2 \rfloor$ where lb and ub stand for the lower and upper bound of the LS problem, respectively. As done in [15], the set of lattice candidates C in this case is found as follows:

$$C = \begin{cases} (m,n) \mid & m \times n \leq mp \\ & (m+1) \times n > mp \\ & m \times (n+1) > mp \\ & \forall (m',n') \notin F, m' \geq m \; and \; n' \geq n \end{cases}$$

where F is a failed set which includes the row and column of lattices on which the given target function cannot be realized as described in the next subsection. This is based on the fact that if it is proved that a given target function cannot be realized using an $m' \times n'$ lattice, then any $m \times n$ lattice with $1 \leq m \leq m'$ and $1 \leq n \leq n'$ cannot be used to realize the target function [15].

For our example in Fig. 2, with the initial lower and upper bounds computed as 9 and 15 in Section 3.1.1, mp is determined as 12 and thus, the set C includes the 1×12, 2×6, 3×4, 4×3, 6×2, 12×1, 2×5, and 5×2 lattices.

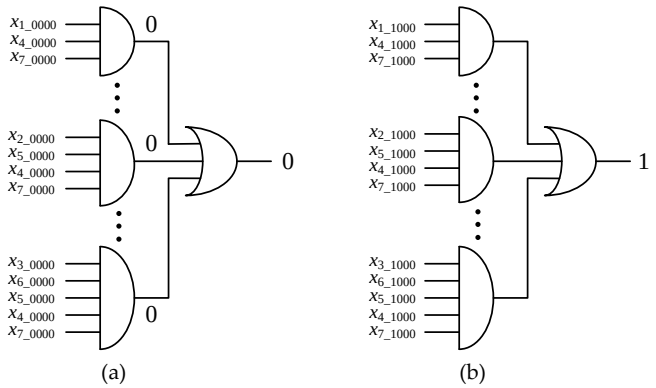


Fig. 9. The combinational circuits of $f_{3\times3}$ for $f = \overline{a}\,\overline{b}c + abc + a\overline{b}\,\overline{c} + b\overline{c}\,\overline{d}$: (a) when $abcd = 0000$ and f is low; (b) when $abcd = 1000$ and f is high.

3.1.3 Finding a Solution to the LM Problem

Given the function of a lattice candidate and the target function with a minimum number of products obtained using a logic minimization tool, both in ISOP form, initially, the structural check, described in Section 3.1.1, is performed. If the structural check is not passed, the row and column of the lattice are added into the failed set F. Otherwise, we check if there exists an assignment to the lattice function variables included in the *lattice variable* LV set from the *target literal* TL set, consisting of the target function literals and constants 0 and 1, such that every entry in the truth table of the target function is satisfied. The LM problem is encoded as a SAT problem in three steps as follows.

1) Generate variables of the LM problem and the necessary constraints related to these variables.
2) Generate constraints which ensure that the lattice function can be used to realize the target function.
3) Generate constraints which enable to reduce the computational effort on the SAT solver.

In the first step of the LM problem encoding, we generate the sets LV and TL and the mapping variables $lv_i\_tl_j$, where $lv_i \in LV$, $1 \leq i \leq |LV|$, $tl_j \in TL$, $1 \leq j \leq |TL|$, and $|A|$ denotes the cardinality of set A. The mapping variable $lv_i\_tl_j$ indicates that the lattice variable lv_i is assigned to an element of TL, tl_j, when this mapping variable is set to high. As an example, consider our target function $f = \overline{a}\,\overline{b}c + abc + a\overline{b}\,\overline{c} + b\overline{c}\,\overline{d}$ in Fig. 2 when the 3×3 lattice is used. Thus, $LV = \{x_1, x_2, \ldots, x_9\}$, $TL = \{a, \overline{a}, b, \overline{b}, c, \overline{c}, \overline{d}, 0, 1\}$, and for example, the mapping variable $x_1\_\overline{a}$ indicates that the lattice variable x_1 is assigned to $\overline{a}$, if $x_1\_\overline{a}$ is set to high. Then, we generate clauses, which confirm that each lattice variable is assigned to only one element in TL, as follows:

$$\prod_{i=1}^{|LV|} \sum_{j=1}^{|TL|} lv_i\_tl_j \quad \text{and} \quad \prod_{i=1}^{|LV|} \prod_{j=1}^{|TL|-1} \prod_{k=j+1}^{|TL|} \overline{lv_i\_tl_j} + \overline{lv_i\_tl_k}$$

where $\prod$ and $\sum$ denote AND and OR operators, respectively. The former clauses guarantee that for each lattice variable, at least one of the mapping variables should be high. The latter ones ensure that for each lattice variable, when one mapping variable is high, the other ones should be low. Note that $\overline{a}+\overline{b}$ is equal to both $a \Rightarrow \overline{b}$ and $b \Rightarrow \overline{a}$, where $\Rightarrow$ stands for the imply operator, indicating that if one of these variables is high, the other one should be low.

In the second step of the LM problem encoding, to satisfy the target function, for each truth table entry, we generate the combinational circuit corresponding to the lattice function and assign the target function value at this entry to the circuit output. The circuit inputs, *i.e.*, the lattice function variables, are associated with the truth table entry and denoted as lv_{i_tte}, where $1 \leq i \leq |LV|$ and tte is the truth table entry. We obtain the POS formula of the circuit as shown in Fig. 4 and simplify it based on the logic value at the circuit output. For our example, Fig. 9 presents the circuits generated for $abcd = 0000$ and $abcd = 1000$, where the target function is 0 and 1, respectively. For the sake of clarity, only three products of $f_{3\times3}$ are shown in this figure.

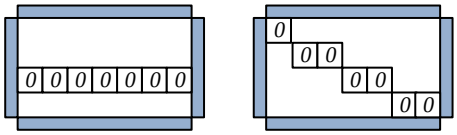


Fig. 10. Two assignments which set the lattice function to low.

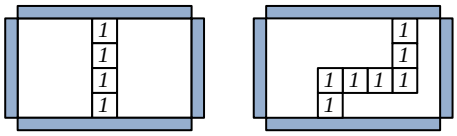


Fig. 11. Two paths between the top and bottom plates in case the target function is high.

Observe from Fig. 9a that when the target function is low for a truth table entry, the logic 0 at the OR gate output can be propagated to the outputs of AND gates and thus, the POS formula of the circuit can be reduced to the only ones that indicate the possible ways of setting each AND gate output to 0. For our example in Fig. 9a, the clause generated for the AND gate at the top is $\overline{x_{1\_0000}} + \overline{x_{4\_0000}} + \overline{x_{7\_0000}}$. Fig. 10 presents two cases showing how a logic function is set to a low value in a lattice. Observe that the control inputs having a low value actually block all paths that can carry a high value from the top plate to the bottom plate.

Observe from Fig. 9b that when the target function is high for a truth table entry, the clauses, which ensure that if an input of the OR gate is high, then the OR gate output should be high, can be removed from the POS formula. Note that, for this case, there should be at least one four-connected path in between the top and bottom plates, where the control inputs of associated switches are set to high. Such paths are illustrated in Fig. 11. Based on these paths, for the truth table entry where the target function is high, there are two facts described as follows: i) in each row of the switching lattice, there should be at least one switch whose control input has a high value; ii) in each two consecutive rows, there should be at least one situation that the control inputs of switches on the same column have a high value. We generate clauses for these facts to hold. These constraints add a problem-specific knowledge into the SAT problem, leading to a decrease in the run-time of the SAT solver.

To link the mapping variables to the circuit inputs, for each mapping variable, we generate clauses which ensure that when a mapping variable is set to high, the associated circuit input, *i.e.*, a lattice variable, is set to a value determined by the truth table entry. For our example, when $abcd = 0000$, the constraints, such as $x_1\_a \Rightarrow \overline{x_{1\_0000}}$ and $x_3\_\overline{b} \Rightarrow x_{3\_0000}$, ensure that the circuit input has the corresponding value when a lattice variable is assigned to a target literal. When a lattice variable is assigned to a constant value 0 or 1, the related circuit input is set to that value.

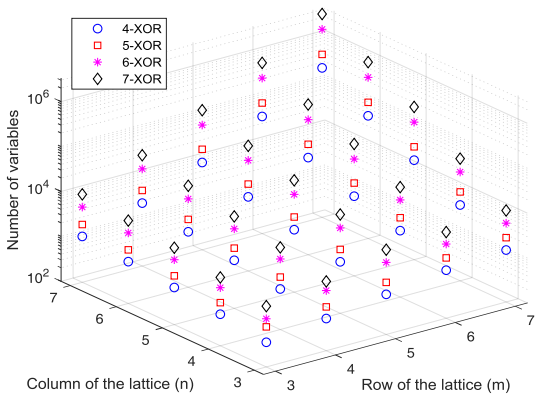
In the third step of the LM problem encoding, if the degree of the target function, denoted as δ, is equal to that of the lattice function, for each product with δ literals in the target function, we generate clauses indicating that at least one of the products with δ variables in the lattice function should be used to realize this product. Consider the realization of $f = \bar{b}\bar{c}\bar{d} + abcde$ on the 3×3 lattice, where δ is 5. It is obvious that the product $x_1x_4x_5x_6x_9$ or $x_3x_6x_5x_4x_7$ of $f_{3\times3}$ should realize $abcde$. Among many possibilities, this can be achieved by setting the mapping variables x_1_a, x_4_b, x_5_c, x_6_d, and x_9_e to high. Moreover, it was noticed that realizing products with a large number of literals in the lattice is a hard task. Hence, if a product of a target function has more than 5 literals (determined empirically), we also generate clauses which ensure that this product is realized by at least one product with more than 5 variables in the lattice function.

Thus, a SAT problem, that formalizes the LM problem, is generated based on the target and lattice functions. We also consider the realization of the dual of target function using the dual of given lattice function and generate another SAT problem using a formulation similar to the one given above. This is due to the fact that the dual of lattice function may have a smaller number of products than that of the lattice function as shown in Table 1 and the dual of target function may have a smaller number of truth table entries where the target function is high, yielding a SAT problem with a small number of variables and clauses. After generating the alternative SAT problem, we choose the one that has the least complexity computed as the number of variables times the number of clauses and then, solve it using a SAT solver. Since it is easier for the SAT solver to find a solution if it exists than to prove that there is no solution, we set a time limit as 1200 seconds, determined empirically. Thus, if the SAT solver finds a solution in the given time limit, the assignment to the lattice variables is obtained by the mapping variables set to high. Otherwise, it is accepted that the target function cannot be realized using the given lattice. If it is proven that the given lattice cannot be used to realize the target function in the given time limit, the row and column of the lattice are added into the failed set F.

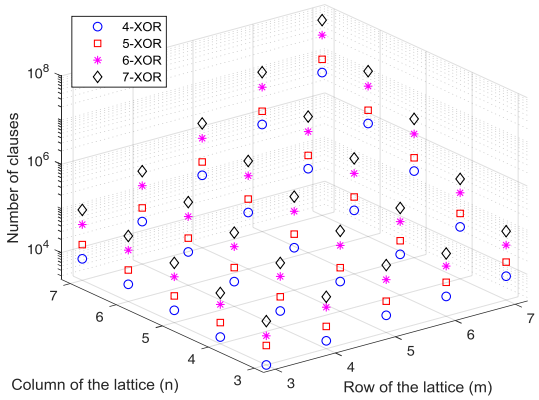
3.1.4 SAT Problem Complexity

In order to show the increase in the complexity of the SAT problem generated by JANUS as the number of literals and products in a target function and the lattice size increase, we consider the logic function of an r-input XOR gate, denoted as r-XOR. Note that r-XOR includes all possible $2r$ literals and consists of $2^{(r-1)}$ products, each having r literals. Fig. 12 presents the number of variables and clauses (in the logarithmic scale) of the SAT problems generated for r-XOR on $m \times n$ lattices where r varies in between 4 and 7 and m and n range in between 3 and 7.

Observe from Fig. 12 that the complexity of the SAT problem increases dramatically as the number of inputs in the XOR logic function, and consequently, the number of products, increase. For example, the SAT problem, which is generated to check if 6-XOR (7-XOR) can be realized using the 7×7 lattice, includes 1,532,826 (3,355,090) variables and 30,031,338 (65,576,998) clauses. It is important to note that the SAT problem complexity may reach to a point



(a)



(b)

Fig. 12. Complexity of SAT problems on XOR functions: (a) number of variables; (b) number of clauses.

that is beyond the capabilities of state-of-art SAT solvers. Moreover, the SAT problem complexity increases as the lattice size increases because the number of products and degree of the lattice function increase as shown in Tables 1 and 2. For example, for 7-XOR to be realized using the 6×6 (7×6) lattice, the SAT problem has 220,866 (844,372) variables and 3,016,945 (13,808,368) clauses. This analysis clearly indicates that the performance of JANUS depends heavily on the number of literals and products of the target function and lattice size. This also points out the importance of improving the initial upper bound because JANUS may need to solve large size SAT problems otherwise.

Although there are logic functions with a small number of products and literals that JANUS can handle easily, there are still complex instances that it may find them hard to solve as shown in Fig. 12. Hence, an algorithm, that can easily cope with such logic functions, is needed.

3.2 MEDEA: A Divide and Conquer Algorithm

The divide and conquer method, called MEDEA, aims to realize complex logic functions using a little computational effort. Its main steps are given as follows:

1) Recursively partition a large number of products in a single function into sub-functions with a small number of products such that they can be handled by JANUS.
2) Find the realizations of these sub-functions using JANUS and merge these lattices into a single lattice.
3) Explore alternative realizations of these sub-functions such that the final lattice requires a small number of switches.

In the first step, the logic function and also, the sub-functions to be generated are recursively partitioned into two sub-functions if the difference between the upper and lower bounds of the function, denoted as $dulb$, is greater than or equal to 31. We note that the lower and upper bounds of a function are computed as described in Section 3.1.1, except the DS method is not used while computing the upper bound. As an example, assume that a target function f is initially partitioned into sub-functions as $f = f_1 + f_2$. Then, the sub-function f_2, denoted as g, which has a $dulb$ value greater than or equal to 31, is decomposed as $g = g_1 + g_2$. Finally, the sub-function g_2, denoted as h, which has a $dulb$ value greater than or equal to 31, is divided into sub-functions as $h = h_1 + h_2$. Thus, the target function is written as $f = f_1 + g_1 + h_1 + h_2$, where the sub-functions f_1, g_1, h_1, and h_2 have a $dulb$ value less than 31. While determining the $dulb$ value, we considered two main criteria. First, the sub-functions having the determined $dulb$ value should lead to SAT problems which can be solved easily using the state-of-art SAT solvers, and thus, they can be easily handled by JANUS. Second, these sub-functions should yield a final lattice with a small size. However, these criteria conflict with each other as also shown in our experiments. It is observed that a large (small) $dulb$ value leads to a small (large) number of sub-functions with a large (small) number of products whose realizations can be found using a great (little) computational effort and which are merged into a small (large) size single lattice. Thus, the $dulb$ value is determined to be 31 based on experiments, meeting these two criteria adequately.

In its second step, as done in the DS method, JANUS is used to find the realizations of these sub-functions on lattices. These lattices are added into a single lattice, separating each one of them by an isolation column and filling the unspecified entries by constant 1 as shown in Fig. 8a.

In its third step, as done in the DS method, the possible realizations of each lattice with a small number of rows and columns are explored and the one that can reduce the final lattice size is chosen to replace the current one.

We note that the run-time limit for the SAT solver used in the second and third steps of MEDEA is set to 300 seconds to reduce the computational effort.

3.3 Realization of Multiple Functions

The proposed algorithms, which realize a single logic function using a switching lattice, can be used to realize multiple functions on a single lattice as follows:

1) Find the realization of each logic function using one of the proposed algorithms.
2) Merge these realizations into a single lattice.
3) Find alternative realizations of these functions that can reduce the final lattice size.

Based on the algorithm used to find the realization of each function on a switching lattice, *i.e.*, JANUS or MEDEA, the developed algorithms are called as JANUS-MF and MEDEA-MF, respectively.

As an example, consider the functions of a full adder, where $c_{out}(a, b, c_{in}) = \sum(3, 5, 6, 7)$ which can be written as $c_{out} = ab + ac_{in} + bc_{in}$ and $s(a, b, c_{in}) = \sum(1, 2, 4, 7)$ which can be written as $s = \overline{a}\overline{b}c_{in} + \overline{a}b\overline{c_{in}} + a\overline{b}\overline{c_{in}} + abc_{in}$.

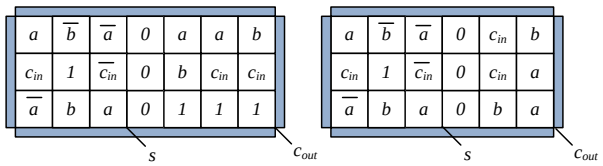


Fig. 13. Realizations of a full adder in a single lattice: (a) 3×7 (b) 3×6.

Fig. 13a presents the 3×7 switching lattice which is formed by merging the realizations of these functions that are found using JANUS. However, in its third step, JANUS-MF finds that the c_{out} function can also be realized using the 3×2 lattice which reduces the final lattice size to 18 as shown in Fig. 13b. Note that the associated outputs can be pinned out at the switch-level as illustrated in Fig. 13.

4 Experimental Results

In this section, we present the results of JANUS, MEDEA, and the methods of [15], [16], [19]. Note that JANUS and MEDEA, developed in Perl, use *espresso* [28] as a logic minimization tool to find the ISOP forms of logic functions and their duals, *glucose4.1* [29] to solve a SAT problem, and *SCIP2.1* [30] to solve a 0-1 ILP problem. The proposed algorithms can be found at https://github.com/leventaksoy/Lattices. We used the updated version of the exact method of [15] where an issue, that may cause the method to miss some paths in a switching lattice, was fixed [19]. The results of the method [16] were obtained using a developed tool that can find the lattice realizations of sub-functions decomposed by the algorithm of [31] and can merge these realizations into a single lattice. Note that the decomposed sub-functions of logic functions used in our experiments were provided by Luca Frontini and were obtained in less than a second. As done in [16], in the developed tool, the time limit for the SAT solver was set to 600 seconds and for the sub-functions whose solutions could not be found by the exact algorithm of [15], the DP method [11] was used. All the algorithms were run on an Intel Xeon CPU at 2.40GHz with 28 cores and 128GB RAM with the CPU time limit of 6 hours.

In order to show the complexity of generated SAT problems and the performance of the SAT solver on these problems, two moderate logic functions, *i.e.*, b12_06 and mp2d_02, were used. Note that the number of inputs, prime implicants, and the degree of the b12_06 (mp2d_02) function is 9 (11), 9 (10), and 6 (4), respectively. Tables 3 and 4 present the results on the SAT problem complexity in terms of the number of variables and clauses where *decision* and *CPU* are the answer and run-time of the SAT solver in seconds, respectively. Note that while the decision *sat*(*unsat*) denotes that the target function can(not) be realized using the given lattice, *undet* indicates that the SAT solver cannot make a decision in the given time limit. The lattices given in these tables are the ones in between the lower and upper bounds computed as described in Section 3.1.1, each passing the structural check as mentioned in Section 3.1.3. Note also that the lattices given in italic indicate that the complexity of the SAT problem generated based on the dual of target and lattice functions is less than that of the SAT problem generated based on the target and lattice functions.

Observe from Tables 3 and 4 that the complexity of the SAT problem differs on the lattices with the same size, *e.g.*,

TABLE 3
Complexity of SAT problems on the b12_06 instance.

lattice	#variables	#clauses	decision	CPU
3x5	10656	150025	unsat	0.4
5x3	11840	168151	unsat	1.2
4x4	12240	171962	unsat	8.5
3x6	13272	186363	unsat	0.5
6x3	16876	245351	unsat	6.5
4x5	17056	239345	unsat	146.0
5x4	19360	276153	sat	2.4
3x7	16040	225047	unsat	1.0
7x3	25580	391541	unsat	30.9

TABLE 4
Complexity of SAT problems on the mp2d_02 instance.

lattice	#variables	#clauses	decision	time
4x7	81308	4508304	sat	861.1
3x10	306120	2607947	sat	27.6
5x6	83822	4213051	sat	901.0
6x5	77142	3178046	undet	1200.0
10x3	64528	1276710	unsat	1.2
4x8	136722	11139857	sat	526.4
8x4	79596	2935943	unsat	78.0

the SAT problems generated for the 3×10, 5×6, 6×5, and 10×3 lattices on the mp2d_02 instance. This is mainly due to different number of products and degree of the lattice function as shown in Tables 1 and 2. Moreover, having a SAT problem with a small complexity does not always mean that it will be solved using a little computational effort, *e.g.*, the SAT problems generated for the 4×5 and 7×3 lattices on the b12_06 instance. This is related to the number of products and the number of literals in products of both target and lattice functions. Furthermore, observe that the complexity of SAT problems generated for the mp2d_02 instance is larger than that of SAT problems generated for the b12_06 instance. This is mainly because the number of inputs of the mp2d_02 instance and the sizes of lattices checked for this function are larger than those in the b12_02 instance.

In order to compare the performance of algorithms, we used 48 instances presented in [20]. Table 5 shows the results of algorithms where *lb* stands for the lower bound found as described in Section 3.1.1, *oub* is the old upper bound computed based on the DP, PS, and DPS methods [19], *nub* is the new upper bound found considering also the solutions of the IPS, IDPS, and DS methods, and *lubt* denotes the time required for the computation of the initial lower and upper bounds in seconds. Finally, *sol* and *CPU* denote the solution and run-time of algorithms in seconds, respectively.

Observe from Table 5 that the use of new methods introduced for finding an upper bound improves the existing upper bound of [19] by 42.8% on average using a little computational effort, reducing the search space of the LS problem significantly. Note that while the DP, PS, and DPS methods find a smaller upper bound on only one instance than other methods, the proposed IPS, IDPS, and DS methods lead to better upper bounds on 39 instances than other methods. Observe also that the new upper bound can be better than the solutions of existing methods proposed for the LS problem, *e.g.*, 5xp1_3.

Observe from Table 5 that JANUS can find better solutions in terms of lattice size than the exact algorithm of [15], *e.g.*, ex5_15, ex5_17, and ex5_24. This is simply because it explores a small search space due to the improved upper bounds and it encodes the LM problem as a SAT problem efficiently. Also, JANUS can find solutions with the same size as the exact algorithm, but using less computational effort, *e.g.*, ex5_23, mp2d_02, and mp2d_04. Although JANUS does not consider all ISOP forms of a logic function while finding the initial lower and upper bounds of the LS problem as described in Section 3.1.1, its solutions are never worse than the exact ones. Furthermore, the solutions of JANUS are better or equal to those found by the existing algorithms, having the smallest lattice size on average. Moreover, MEDEA can find solutions using a little computational effort with respect to other algorithms except the method of [16], *e.g.*, 5xp1_3, ex5_23, and mp2d_01, and its solutions are better than those of the existing approximate algorithms and close to those of the exact algorithm and JANUS on average. On the other hand, the strict rules on the realization of a product in the approximate method of [15] yield the worst solutions on instances ex5_15, ex5_17, and ex5_23. The solutions of the method [19] may be far away from the optimal, *e.g.*, 5xp1_3, ex5_24, and mp2d_01, since it does not consider all possible lattice candidates. The method of [16] finds solutions using the least computational effort on average, but its solutions are worse than those of other algorithms on average.

Tables 6 and 7 respectively present the ratio of CPU times required by the tools and methods used in JANUS and MEDEA over their total run-time given in percentage on instances in Table 5. Observe from these tables that while the run-time of JANUS is dominated by solving LM problems formulated as SAT problems, the run-time of MEDEA is dominated by finding lattice realizations using JANUS.

In order to explore the limitations of algorithms [15], [19] and JANUS, we used 12 logic functions taken from the LGS91 benchmark [32]. Table 8 presents the results of algorithms where *#in*, *#pi*, and δ denote the number of inputs, prime implicants, and degree of the target functions in ISOP form, respectively. Since the methods of [15], [19] cannot find a solution on these instances in the given time limit, *i.e.*, 6 hours, their CPU time is not listed to avoid repeated values.

We note that the initial upper bound of the LS problem for each instance given in Table 8 is found by the DS method, leading to a 73.6% reduction on average when compared to the old upper bound computed by the DP, PS, and DPS methods. Observe that the found upper bound is better than the solutions of the algorithms [15], [16], [19] on average. However, the DS method cannot find the upper bound in the given time limit on two instances, *i.e.*, sao2_01 and sao2_03.

Observe from Table 8 that the algorithms [15], [19] and JANUS find these instances hard to solve. While all solutions of the method [19] are equal to the initial upper bound, the methods of [15] can only improve the initial upper bound value on the inc_03 and rd53_01 instances. There is only one function, *i.e.*, inc_03, that JANUS can find a solution in a given time limit. Observe that JANUS cannot improve the initial upper bound of any instance. However, it can find significantly better solutions than the algorithms [15], [19] which is due to the DS method used to find an initial upper bound. On the other hand, the method of [16] can find better solutions than the algorithms of [15], [19], *i.e.*, all instances except apex4_18 and sao2_01. This is because while the decomposed sub-functions can be solved by the exact method of [15], the whole functions are hard to solved by the exact method. However, the solutions of [16] are worse than those of JANUS and MEDEA on average. Moreover, the solutions of MEDEA are obtained using the least computational effort on

TABLE 5
Summary of initial lower and upper bounds and results of algorithms on moderate single functions.

Instance	Lower and Upper Bounds				[16]		[19]		Approximate [15]		Exact [15]		JANUS		MEDEA	
	lb	oub	nub	lubt	sol	CPU	sol	CPU	sol	CPU	sol	CPU	sol	CPU	sol	CPU
5xp1_1	16	105	32	4.1	5x10	4.2	5x5	501.2	6x5	21600	5x5	21600	4x6	2023.2	4x8	2.2
5xp1_3	15	135	40	57.3	4x11	11.1	5x27	21600	11x4	21600	11x4	21600	4x9	19745.8	5x8	55.8
b12_00	9	24	20	0.2	4x3	0.8	4x3	0.3	4x3	0.6	4x3	2.1	4x3	0.3	4x3	0.3
b12_01	12	35	20	0.2	4x4	2.5	4x4	1.1	4x4	1.6	5x3	8.5	5x3	1.1	5x3	1.2
b12_02	12	42	24	0.8	5x8	3.4	4x4	5.7	5x4	3.7	4x4	35.4	4x4	4.1	4x4	3.4
b12_03	6	6	6	0.1	2x5	0.1	3x2	0.1	3x2	0.2	3x2	0.1	3x2	0.1	3x2	0.1
b12_06	15	44	24	4.3	5x10	22.6	5x4	23.8	5x4	4.6	5x4	139.3	5x4	23.8	5x4	23.8
b12_07	16	24	24	0.3	4x8	0.1	3x6	1.1	5x4	2.5	3x6	5.4	3x6	1.5	3x6	0.2
c17_01	6	6	6	0.1	2x5	0.1	3x2	0.1	3x2	0.2	3x2	0.1	3x2	0.1	3x2	0.1
clpl_00	12	16	15	0.2	4x7	0.2	3x4	0.4	3x4	0.3	3x4	1.3	3x4	0.3	3x4	0.2
clpl_03	16	36	24	0.6	6x9	6.6	3x6	19.6	3x6	2.3	3x6	200.0	3x6	84.9	3x6	20.6
clpl_04	15	25	18	0.3	5x8	1.1	3x5	5.0	3x5	1.3	3x5	25.3	3x5	1.3	3x5	1.0
dc1_00	9	16	15	0.2	4x4	0.2	3x3	0.1	3x3	0.4	3x3	0.4	3x3	0.2	3x3	0.2
dc1_02	12	16	15	0.2	3x5	0.1	3x4	0.1	3x4	0.3	4x3	0.2	4x3	0.3	4x3	0.2
dc1_03	9	20	18	0.2	4x5	0.1	4x3	0.2	4x3	0.4	4x3	0.5	4x3	0.3	4x3	0.3
ex5_06	16	32	24	0.3	3x10	0.2	3x6	1.2	3x7	12.0	3x6	7.2	3x6	2.1	3x6	0.5
ex5_07	24	40	27	0.7	3x13	0.2	4x6	19.7	3x9	332.2	4x6	473.2	3x8	2.5	4x6	29.3
ex5_08	20	21	21	0.2	3x9	1.5	3x7	0.0	3x7	9.3	3x7	51.2	3x7	7.2	3x7	0.2
ex5_09	24	40	30	12.3	3x11	6.1	4x6	5.7	3x8	108.2	4x6	454.6	3x8	17.6	3x8	12.4
ex5_10	16	21	21	0.2	3x9	0.2	3x6	0.7	3x6	1.4	3x6	3.8	3x6	0.5	4x5	0.2
ex5_12	15	25	20	0.2	4x11	3.8	3x5	1.8	3x5	1.7	3x5	13.7	3x5	12.6	3x5	15.6
ex5_13	24	36	27	0.9	3x13	0.1	3x8	10.0	4x6	57.6	4x6	190.2	3x8	2.8	4x6	4.0
ex5_14	16	16	16	0.2	3x11	0.1	2x8	0.9	2x8	1.2	2x8	6.7	2x8	0.2	2x8	0.3
ex5_15	20	72	33	3.1	4x13	2.2	4x7	48.5	6x12	21600	6x5	21600	3x8	2562.4	3x11	5.4
ex5_17	20	105	42	23.2	4x13	21.6	4x7	1425.6	10x6	21600	6x6	21600	3x9	4377.6	4x10	29.1
ex5_19	16	18	18	0.1	3x8	0.1	3x6	1.4	3x6	1.1	3x6	6.9	3x6	0.4	3x6	0.2
ex5_21	20	57	30	0.5	3x11	9.1	3x7	8.2	4x7	1364.6	3x7	280.9	3x7	790.8	3x7	411.0
ex5_22	16	33	21	0.2	3x8	1.8	3x6	1.3	3x6	2.0	3x6	8.4	3x6	1.2	3x6	2.4
ex5_23	24	92	36	39.0	4x11	13.2	4x8	2465.0	11x5	21600	3x9	15418.6	3x9	3726.4	3x12	29.9
ex5_24	20	105	33	7.0	5x14	29.3	15x7	21600	3x11	21600	4x7	21600	3x8	1638.8	3x13	2.6
ex5_25	20	40	27	0.3	4x10	2.0	3x7	16.4	3x7	6.4	3x7	79.4	3x7	152.7	3x7	7.1
ex5_26	20	57	30	0.7	3x13	7.8	3x7	12.9	3x9	384.5	3x7	238.5	3x7	36.3	3x7	43.8
ex5_27	20	77	27	1.3	4x11	7.8	4x6	58.1	3x8	1049.5	4x6	1561.3	3x8	1229.3	3x9	1.7
ex5_28	24	27	27	0.2	3x11	6.0	3x8	5.3	3x8	180.2	6x4	51.5	3x8	1.6	3x8	1.9
misex1_00	6	8	8	0.1	4x3	0.1	4x2	0.1	4x2	0.2	4x2	0.2	4x2	0.1	4x2	0.2
misex1_01	12	35	18	0.2	5x5	0.9	3x5	1.9	4x4	1.7	3x5	7.4	3x5	1.1	3x5	1.4
misex1_02	12	40	25	0.4	5x5	1.4	5x4	24.0	5x4	4.6	5x4	50.9	5x4	19.7	5x4	23.7
misex1_03	9	28	20	0.3	4x6	0.3	4x3	0.9	5x3	1.2	4x3	3.9	4x3	0.5	4x3	0.4
misex1_04	12	25	18	0.2	4x6	0.1	3x4	0.2	5x3	1.0	3x4	0.7	3x4	0.4	3x4	0.4
misex1_05	12	42	21	0.3	4x6	0.7	4x4	4.6	5x4	4.9	4x4	13.4	4x4	2.1	4x4	3.8
misex1_06	12	35	18	0.2	5x6	0.8	5x3	1.3	5x3	1.6	5x3	4.7	5x3	1.3	5x3	1.7
misex1_07	9	20	18	0.3	4x6	0.2	4x3	0.7	5x3	1.0	4x3	1.6	4x3	0.5	4x3	0.5
mp2d_01	24	48	30	4.3	4x11	0.9	5x7	28.7	4x7	291.3	3x9	6478.3	3x9	3257.3	4x8	0.3
mp2d_02	28	50	33	0.9	4x13	0.4	4x9	33.9	4x7	730.7	4x7	4580.7	4x7	948.9	3x10	0.3
mp2d_03	15	72	32	4.5	7x6	19.8	5x5	42.3	4x6	188.2	6x4	1322.7	4x6	271.2	4x8	5.5
mp2d_04	15	57	36	5.5	7x3	184.2	7x3	18.9	7x3	58.8	7x3	3043.1	7x3	286.8	7x3	299.4
mp2d_06	8	18	16	0.3	5x4	0.3	6x2	0.3	7x2	1.2	4x3	1.1	6x2	0.4	6x2	0.2
newtag_00	16	32	24	0.2	3x8	1.4	3x6	2.7	3x6	2.1	3x6	19.0	3x6	2.2	3x6	0.2
Average	15.5	41.1	23.5	3.7	32.4	7.9	22.7	1000.0	22.0	2800.4	18.9	2974.8	18.3	859.2	19.9	21.8

TABLE 6
Summary of percentage of CPU times of tools in run-time of JANUS.

Logic Minimization espresso [28]	SAT Problem Solving glucose [29]
0.03	99.16

TABLE 7
Summary of percentage of CPU times of tools in run-time of MEDEA.

Logic Minimization espresso [28]	Lattice Realization JANUS
0.8	97.35

average and they are better than those of all algorithms on average, except those of JANUS.

In order to show the tradeoff between the solution quality and run-time in MEDEA, on the instances given in Table 8, the *dulb* value, which is used while partitioning a logic function into two sub-functions, is changed from 11 to 51, in step of 10. These results are shown in Table 9 where #*sf* denotes the number of generated sub-functions.

Observe from Table 9 that as the *dulb* value increases, the number of sub-functions is decreased, increasing the solution quality and the run-time of MEDEA. Note that while there is a 36.3% reduction in the average lattice size, there exists a 23.8x increase in the average run-time when the results obtained with the *dulb* value 11 and 51 are compared. Observe from Table 8 that the solutions of MEDEA obtained for all *dulb* values are better than the old initial upper bound values and those of the methods [15], [16], [19] on average.

In order to show the limitations of JANUS and to demonstrate the importance of MEDEA, we used 15 logic functions taken from the LGS91 benchmark [32]. Table 10 presents the details of logic functions, the initial lower bound of the search space and its upper bounds found by different methods, and the results of algorithms. Note that the lower and upper bounds were computed in less than a second. However, the algorithms of [15], [16], [19] and JANUS cannot handle these instances in the given time limit, *i.e.*, 6 hours, and can only return a solution obtained by the techniques used to find the initial upper bounds of the LS problem. The DS method of JANUS could not find a solution on any of these instances in the given time limit. All solutions of the method [16] were found using the DP method [11] on the decomposed sub-functions since the exact algorithm of [15] could not handle these sub-functions. In this table, *size* stands for the number of switches in the lattice.

Observe from Table 10 that on each logic function, the proposed IPS and IDPS methods give an upper bound which is significantly better than that found by the pre-

TABLE 8
Summary of initial lower and upper bounds and results of algorithms on hard single functions.

Instance	Function Details			Lower and Upper Bounds				[16]		[19]	App. [15]	Exact [15]	JANUS		MEDEA	
	#in	#pi	δ	lb	oub	nub	lubt	sol	CPU	sol	sol	sol	sol	CPU	sol	CPU
apex4_15	9	13	8	18	186	50	4435.9	4x19	8520.1	31x6	8x25	8x25	5x10	21600	5x11	16.7
apex4_16	9	11	8	18	168	48	5340.6	5x11	2742.5	8x21	8x21	8x21	4x12	21600	5x12	14.2
apex4_17	9	12	8	18	184	49	1615.6	4x22	7419.7	8x23	8x23	8x23	7x7	21600	5x15	8.6
apex4_18	9	14	8	20	215	56	4090.7	22x14	21600	43x5	8x27	8x27	7x8	21600	6x13	859.4
clip_00	9	21	6	18	246	55	4249.7	5x14	341.5	6x41	6x41	6x41	5x11	21600	5x16	26.5
clip_04	9	20	6	18	234	44	1407.2	5x10	146.7	6x39	6x39	6x39	4x11	21600	4x13	14.9
inc_03	7	11	5	15	105	36	43.0	6x8	37.4	5x21	19x3	19x3	4x9	15023.7	4x11	2.6
rd53_01	5	16	5	18	155	44	53.2	4x11	16.5	5x31	9x9	9x5	4x11	21600	4x11	52.5
sao2_01	10	20	10	24	390	84	21600	25x17	21600	10x39	10x39	10x39	12x7	21600	8x18	1979.0
sao2_02	10	22	4	21	161	52	5649.2	5x15	266.8	23x7	4x43	4x43	4x13	21600	4x18	11.3
sao2_03	10	21	5	24	168	70	21600	5x14	3495.2	21x8	5x41	5x41	5x14	21600	5x18	200.0
z5xp1_03	7	18	6	18	210	50	6850.6	6x10	309.6	6x35	6x35	6x35	5x10	21600	4x18	9.3
Average	8.5	16.4	6.5	18.8	196.7	53.2	6411.3	114.1	5541.3	201.8	196.9	193.9	53.2	21052.0	72.2	266.2

TABLE 9
Summary of results of MEDEA with different dulb values on hard single functions.

Instance	$dulb$ = 11			$dulb$ = 21			$dulb$ = 31			$dulb$ = 41			$dulb$ = 51		
	#sf	sol	CPU	#sf	sol	CPU	#sf	sol	CPU	#sf	sol	CPU	#sf	sol	CPU
apex4_15	5	5x17	6.9	4	5x14	7.6	3	5x11	16.7	3	5x11	16.5	3	5x11	16.3
apex4_16	6	4x22	7.7	3	5x12	14.0	3	5x12	14.2	3	5x12	14.3	2	4x13	2650.2
apex4_17	5	6x15	7.4	4	5x15	10.2	4	5x15	8.6	4	5x15	8.5	3	6x11	1555.5
apex4_18	6	6x17	7.8	4	6x13	857.1	4	6x13	859.4	3	6x11	1045.8	3	6x11	1040.3
clip_00	7	4x26	4.4	4	5x16	26.6	4	5x16	26.5	4	5x16	26.8	4	5x16	26.2
clip_04	7	4x22	2.8	6	4x20	2.7	3	4x13	14.9	3	4x13	14.7	3	4x13	14.4
inc_03	4	4x14	1.6	4	4x14	1.6	3	4x11	2.6	2	4x9	50.2	2	4x9	50.0
rd53_01	4	4x15	1.9	4	4x15	1.7	2	4x11	52.5	2	4x11	52.2	2	4x11	52.2
sao2_01	11	8x32	356.8	6	8x20	1558.1	5	8x18	1979.0	4	8x15	2138.7	4	8x15	2132.4
sao2_02	8	4x23	3.5	6	4x20	8.2	4	4x18	11.3	4	4x18	10.9	2	4x13	2031.0
sao2_03	11	13x8	5.5	6	5x20	22.3	4	5x18	200.0	4	5x18	199.0	4	5x18	198.3
z5xp1_03	8	4x27	3.0	6	4x23	4.4	4	4x18	9.3	4	4x18	9.0	4	4x18	9.1
Average	6.8	102.8	34.1	4.8	82.6	209.6	3.6	72.2	266.2	3.3	68.5	298.9	3.0	65.4	814.6

TABLE 10
Summary of initial lower and upper bounds and results of algorithms on very hard single functions.

Instance	Function Details			Lower and Upper Bounds						[16]	[19]	[15]	JANUS	MEDEA		
	#in	#pi	δ	lb	DP	PS	DPS	IPS	IDPS	sol	sol	sol	sol	sol	size	CPU
alu4_02	14	50	6	24	2000	594	553	450	413	25x54	79x7	6x99	59x7	5x36	180	164.2
alu4_03	14	72	7	24	5184	1001	1001	749	749	56x73	143x7	7x143	107x7	5x64	320	850.1
alu4_05	14	90	9	24	8100	1611	1611	1224	1224	90x93	179x9	9x179	136x9	5x121	605	2736.0
alu4_06	14	36	7	20	1296	497	497	385	385	36x36	71x7	7x71	55x7	5x35	175	1285.0
apex4_01	9	33	9	20	1221	585	438	441	330	28x35	73x6	9x65	55x6	6x36	216	214.5
apex4_02	9	71	9	24	5183	1269	1305	954	981	47x75	9x141	9x141	9x106	6x74	444	3146.6
apex4_03	9	69	9	24	4761	1233	1096	927	824	47x69	137x8	9x137	103x8	6x78	468	1094.0
apex4_04	9	76	9	24	5624	1359	1176	1017	888	45x81	147x8	9x151	111x8	6x96	576	1583.4
apex4_05	9	78	9	25	5928	1395	1359	1044	1017	48x84	151x9	9x155	113x9	6x92	552	1196.8
apex4_06	9	76	8	24	5700	1208	1341	912	1008	46x85	8x151	8x151	8x114	5x112	560	1862.0
apex4_07	9	75	9	24	5400	1341	1144	1008	864	46x79	143x8	9x149	108x8	6x89	534	975.2
apex4_08	9	76	8	24	5700	1208	1192	912	904	46x84	149x8	8x151	113x8	6x87	522	2758.3
apex4_09	9	72	8	24	5472	1144	1208	864	920	45x75	8x143	8x143	8x108	5x104	520	275.4
apex4_10	9	74	9	24	4884	1323	1048	990	800	41x79	131x8	9x147	100x8	5x98	490	761.6
z9sym	9	84	6	24	6048	1002	1001	774	749	34x112	143x7	6x167	107x7	5x112	560	1938.7
Average	10.3	23.5	8.1	23.5	4833.4	1118.0	1064.7	843.4	803.7	3490.6	1049.1	1118.0	791.8	448.1	448.1	1389.4

viously proposed methods. Although the method of [16] improves the solution of the DP method [11] on the whole logic function on average by finding the realizations of the decomposed sub-functions using the DP method [11], its solutions are still worse than those of the IPS and IDPS methods. On the other hand, MEDEA finds significantly better solutions using a little computational effort than the available methods, yielding a 43.4% decrease in the lattice size on average when compared to JANUS. This experiment clearly indicates that MEDEA is crucial especially on logic functions that the existing algorithms cannot handle.

Observe from Table 10 that since the methods of [15], [16], [19] and JANUS include QBF-and SAT-based techniques to solve the LM problem, *i.e.*, to determine if a logic function can be realized using a given switching lattice, and the LM problem is an NP-complete problem, there exist instances that these methods find them hard to solve and even cannot handle. However, MEDEA can be applied to a logic function with a large number of inputs and products since it partitions this function into sub-functions with a small number of inputs and products which can be easily handled by JANUS.

TABLE 11
Summary of results of algorithms on multiple functions.

Instance	#out	straight-forward method			JANUS-MF		
		sol	size	CPU	sol	size	CPU
b12	9	5x46	230	37.6	4x45	180	2451.8
bw	28	5x119	595	12.7	3x135	405	14.1
misex1	7	5x31	155	25.3	3x42	126	30.4
squar5	8	5x31	155	31.7	3x36	108	59.7

Finally, Table 11 presents the results on instances including multiple functions. In this table, *#out* denotes the number of outputs of given instances and the *straight-forward method* applies JANUS on each target function and merges its solutions into a single lattice, *i.e.*, first two steps of the algorithm developed for the realization of multiple functions described in Section 3.3.

Observe from Table 11 that JANUS-MF outperforms the straight-forward method where the maximum gain is achieved as 32% on the bw instance. This is simply due to the impact of Step 3 of the algorithm as mentioned in Section 3.3, *i.e.*, finding alternative realizations of logic functions. Note that JANUS finds a realization of a logic function without considering that all the realizations of logic

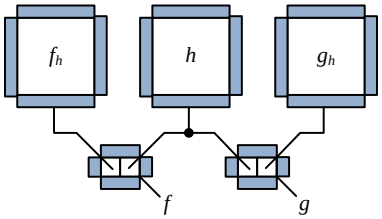


Fig. 14. Sharing common products using multiple lattices.

functions will be merged in a single lattice. However, it aims to a find a small size lattice realizing a logic function. Hence, taking the complexity of these realizations into account, in the Step 3 of the algorithm, JANUS-MF explores different realizations of logic functions systematically and tries to reduce the size of the final lattice as shown in Fig. 8. We also note that MEDEA-MF finds the same results of JANUS-MF, except the squar5 instance, where its solution, found in 7 seconds, includes a 3×40 lattice.

5 Conclusions and Future Work

This article addresses the problem of realizing a logic function on a switching lattice using a minimum number of four-terminal switches and introduces two algorithms called JANUS and MEDEA. While JANUS is developed for finding a solution close to the minimum, MEDEA is proposed to handle the instances, that JANUS finds them hard to solve, using a little computational effort. This article also introduces methods that can reduce the initial lower and upper bounds of search space, leading to significant reductions in run-time. Moreover, it presents an efficient SAT formulation of the problem of checking if a given target function can be realized using the given switching lattice. Furthermore, this article describes how multiple functions can be realized on a single lattice efficiently. Experimental results show that while JANUS can find significantly better solutions than existing exact and approximate algorithms, MEDEA can easily obtain solutions on relatively large size instances that JANUS and other exact and approximate algorithms cannot handle and its solutions can be better than those of the previously proposed algorithms.

In the realization of multiple functions, the lattice size can be further reduced by sharing the common products, which can be achieved using multiple lattices. As an illustrative example, consider the target functions f and g, where h denotes the function including the common products of these functions, f_h and g_h stand for the functions generated after the products of h are extracted from f and g, respectively such that $f = h + f_h$ and $g = h + g_h$. Fig.14 shows the realizations of multiple functions f and g. Rather than the common products, the common logic expressions, which can be found using a state-of-art logic synthesis tool, can also be utilized as done for the realization of a single function in [18]. However, although this technique may reduce the number of switches due to the sharing, it may increase the total area due to the connecting wires between the lattices. Hence, rather than the optimization of the number of four-terminal switches, the optimization of area of the whole design can be considered in this case.

Moreover, since there exist alternative realizations of a logic function using different switching lattices, each having a different area, delay, and power dissipation values, algorithms, that take into account the area, delay, and power dissipation of the design, can also be developed.

Acknowledgment

This work is part of a project that has received funding from the European Union's H2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 691178, and supported by the TUBITAK-Career project #113E760 and TUBITAK-NSF project #218E068. The authors would like to thank Luca Frontini for providing the sub-functions of logic functions used in this article which are obtained using the decomposition algorithm mentioned in [16].

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.
[2] V. V. Zhirnov, R. K. Cavin, J. A. Hutchby, and G. I. Bourianoff, "Limits to binary logic switch scaling - a gedanken model," *Proceedings of the IEEE*, vol. 91, no. 11, pp. 1934–1939, 2003.
[3] A. Y. Zomaya, *Handbook of Nature-Inspired and Innovative Computing*. Springer, 2006.
[4] A. Dehon, "Nanowire-based programmable architectures," *ACM JECT*, vol. 1, no. 2, pp. 109–162, 2005.
[5] M. Dong and L. Zhong, "Nanowire crossbar logic and standard cell-based integration," *IEEE TVLSI*, vol. 17, no. 8, pp. 997–1007, 2009.
[6] Y. Huang, X. Duan, Y. Cui, L. J. Lauhon, K.-H. Kim, and C. M. Lieber, "Logic gates and computation from assembled nanowire building blocks," *Science*, vol. 294, no. 5545, pp. 1313–1317, 2001.
[7] G. Snider, "Computing with hysteretic resistor crossbars," *Applied Physics A: Materials Science & Processing*, vol. 80, no. 6, pp. 1165–1172, 2005.
[8] G. Snider, P. Kuekes, T. Hogg, and R. S. Williams, "Nanoelectronic architectures," *Applied Physics A*, vol. 80, no. 6, pp. 1183–1195, 2005.
[9] H. Yan, H. S. Choe, S. Nam, Y. Hu, S. Das, J. F. Klemic, J. C. Ellenbogen, and C. M. Lieber, "Programmable nanowire circuits for nanoprocessors," *Nature*, vol. 470, no. 7333, pp. 240–244, 2011.
[10] D. Alexandrescu, M. Altun, L. Anghel, A. Bernasconi, V. Ciriani, L. Frontini, and M. Tahoori, "Logic synthesis and testing techniques for switching nano-crossbar arrays," *MICPRO*, vol. 54, pp. 14–25, 2017.
[11] M. Altun and M. Riedel, "Logic synthesis for switching lattices," *IEEE Transactions on Computers*, vol. 61, no. 11, pp. 1588–1600, 2012.
[12] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification. [Online]. Available: https://people.eecs.berkeley.edu/∼alanmi/abc/
[13] S. Safaltin, O. Gencer, M. C. Morgul, L. Aksoy, S. Gurmen, C. A. Moritz, and M. Altun, "Realization of four-terminal switching lattices: Technology development and circuit modeling," in *DATE*, 2019, pp. 504–509.
[14] F. Moraes, L. Torres, M. Robert, and D. Auvergne, "Estimation of layout densities for cmos digital circuits," in *PATMOS*, 1998, pp. 61–70.
[15] G. Gange, H. Søndergaard, and P. J. Stuckey, "Synthesizing optimal switching lattices," *ACM TODAES*, vol. 20, no. 1, pp. 6:1–6:14, 2014.
[16] A. Bernasconi, V. Ciriani, L. Frontini, V. Liberali, G. Trucco, and T. Villa, "Logic synthesis for switching lattices by decomposition with p-circuits," in *DSD*, 2016, pp. 423–430.
[17] A. Bernasconi, V. Ciriani, L. Frontini, and G. Trucco, "Synthesis of switching lattices of dimensional-reducible boolean functions," in *VLSI-SoC*, 2016, pp. 1–6.
[18] ——, "Composition of switching lattices for regular and for decomposed functions," *MICPRO*, vol. 60, pp. 207–218, 2018.
[19] M. Morgul and M. Altun, "Optimal and heuristic algorithms to synthesize lattices of four-terminal switches," *Integration*, vol. 64, pp. 60–70, 2019.
[20] L. Aksoy and M. Altun, "A satisfiability-based approximate algorithm for logic synthesis using switching lattices," in *DATE*, 2019, pp. 1637–1642.

[21] A. S. Cook, "The complexity of theorem-proving procedures," in *ACM Symposium on Theory of Computing*, 1971, pp. 151–158.
[22] T. Larrabee, "Test pattern generation using boolean satisfiability," *IEEE TCAD*, vol. 11, no. 1, pp. 4–15, 1992.
[23] Xilinx, "7 series fpgas configurable logic block user guide," 2016.
[24] R. C. Minnick, "Survey of microcellular research," *Journal of the Association of Computing Machinery*, vol. 14, no. 2, pp. 203–241, 1967.
[25] A. Mukhopadhyay, "Unate cellular logic," *IEEE Transactions on Computers*, vol. 18, no. 2, pp. 114–121, 1969.
[26] J. S. B. Akers, "A rectangular logic array," *IEEE Transactions on Computers*, vol. 21, no. 8, pp. 848–857, 1972.
[27] W. Rao, A. Orailoglu, and R. Karri, "Topology aware mapping of logic functions onto nanowire-based crossbar architectures," in *DAC*, 2006, pp. 723–726.
[28] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Springer, 1984.
[29] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solver," in *IJCAI*, 2009, pp. 399–404.
[30] T. Achterberg, "Scip: solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, Jul 2009.
[31] A. Bernasconi, V. Ciriani, G. Trucco, and T. Villa, "On decomposing boolean functions via extended cofactoring," in *DAC*, 2009, pp. 1464–1469.
[32] S. Yang, "Logic synthesis and optimization benchmarks user guide: Version 3.0," MCNC, Tech. Rep., Jan. 1991.

Levent Aksoy received his M.S. and Ph.D. degrees in electronics and communication engineering and electronics engineering from Istanbul Technical University (ITU), Istanbul, Turkey, in 2003 and 2009, respectively. He worked as a post-doctoral researcher in Algorithms for Optimization and Simulation (ALGOS) group of the Instituto de Engenharia de Sistemas e Computadores (INESC-ID), Lisbon. Then, he joined Dialog Semiconductor as a digital design engineer and worked mainly on the design and verification of asynchronous logic circuits. Currently, he is a research fellow in the Emerging Circuits and Computation (ECC) Group in ITU. His research interests include CAD for VLSI circuits with emphasis on solving EDA problems using SAT models and optimization techniques.

Mustafa Altun received his BSc and MSc degrees in electronics engineering at Istanbul Technical University in 2004 and 2007, respectively. He received his PhD degree in electrical engineering with a PhD minor in mathematics at the University of Minnesota in 2012. Since 2013, he has served as an assistant professor at Istanbul Technical University and runs the Emerging Circuits and Computation (ECC) Group. Dr. Altun has been served as a principal investigator/researcher of various research projects including EU H2020 RISE, National Science Foundation of USA (NSF) and TUBITAK projects. He is an author of more than 50 peer reviewed papers and a book chapter, and the recipient of the TUBITAK Success, TUBITAK Career, and Werner von Siemens Excellence awards