

Automated Mapping for Reconfigurable Single Electron Transistor Arrays

Abstract—Reducing power consumption has become one of the primary challenges in chip design, and therefore significant efforts are being devoted to find holistic solutions on power reduction from the device level up to the system level. Among a plethora of low power devices that are being explored, single electron transistors (SETs) at room temperature are particularly attractive. Although prior work has proposed a binary decision diagram(BDD) based reconfigurable logic architecture using SETs, it lacks an automated synthesis tool for the device. Consequently, in this work, we develop a product-term-based approach that synthesizes a logic circuit by mapping all its product terms into the SET architecture. The experimental results show the effectiveness and efficiency of the proposed approach on a set of MCNC benchmarks.

I. INTRODUCTION

As technology scaling enables packing of billion transistors into a single chip, power consumption becomes one of the primary bottlenecks of continuously meeting Moore’s law. At the system level, there has been a paradigm shift from frequency scaling of a monolithic processor to multiple slower computing nodes that communicate through a common network fabric [6] [12]. A tight power budget constraint is one of the primary reasons that causes this paradigm shift. Moreover, leakage power is becoming a dominant source of power consumption and several works have looked into mitigating this power wastage [5] [7].

On the device level, as the power-delay product reaches quantum tunneling in semiconductor layers as the operation basis. These novel device structures use significantly low-drive current of the order of a few electrons. Numerous demonstrations of the room temperature operation of Single Electron Transistors (SETs) have proved that these devices are very attractive as a possible way for extending Moore’s law.

Majority of these ultra-low power emerging nanodevices suffer from low transconductance and degraded output resistance, making it essential to co-explore an emerging device design in conjunction with a non-CMOS logic architecture. To this end, a novel binary decision diagram (BDD)-based [1] logic architecture was proposed as a suitable candidate for implementing logic using ultra-low power nanodevices [4]. Then, the BDD of a combinational circuit is mapped onto a hexagonal nanowire network controlled by Schottky wrap gates [3].

In the hexagonal network, a logic function is achieved by a passive path switching of messenger electrons that arrive at the root node through either the left arm (“0”) or right arm (“1”) depending on the control gate of the wrap gates. Each row of the hexagonal fabric is controlled by a single variable. Both the normal and the complement of the variable are supplied to a node of the BDD and are used to control the left and right edges as shown in Fig. 1(a).

A BDD implementation can be mapped onto this fabric and the variables implementing the given function establish a path in this fabric from the root node to either a 1 terminal or a 0 terminal to realize the desired functionality. Fig. 1(b) shows an example of a 2-bit XOR. There is a current detector at the root associated to every output bit that measures the current (if any). Depending on the operating modes, active high or active low, the current flowing is interpreted as a logic one or a zero (In the active high mode, no current is a logic zero and presence of current is a logic one and vice-versa in the active low mode).

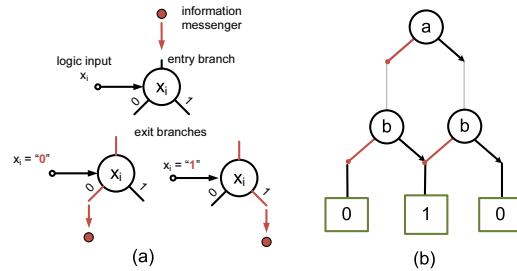


Fig. 1. A hexagonal fabric. (a) Node devices. (b) An example of a 2-bit XOR.

However, the realization of the BDD architecture in [4] is fixed and not amenable to functional reconfiguration. This is because the approach selectively etches all paths that do not lead to a 1 terminal and also customizes the edges of a hexagon to either be a conducting nanowire or have a wrapped gate. Consequently, this structure is not very regular and cannot be restructured to implement a different function due to the physical etching process involved in its realization. Furthermore, if any of the nanowire segments or the wrap gates is defective, the whole circuit becomes non-functional. This is a significant limitation considering that nanowires and few electron nanodevices have traditionally suffered from the variability and reliability issues.

To solve the problem, a reconfigurable version of SET using wrap gate tunable tunnel barriers was proposed [2] and the in-depth device simulation to study the electrostatic properties was presented [8]. This device can operate in three distinct operation states: a) active b) open and c) short state based on the wrap gate bias voltages. Such programmability leads to immense flexibility in designing a circuit. The device simulation shows that this device can provide an order of magnitude lower energy-delay than CMOS device [8].

However, the synthesis of a BDD using the device in [2] is manual rather than automated. The reason is that mapping a reduced ordered BDD (ROBDD) into a planar SET array could be very complicated, especially when the BDD has crossing edges, which is typical in minimized BDDs. In this work, we address this mapping problem and propose an automated mapping approach. Instead of mapping a BDD directly, the proposed approach first divides a BDD into a set of product terms that represent the paths leading to the 1 terminal in the BDD. Then, it sequentially maps these product terms. Since the mapping order of the product terms affects the mapping results, we propose four sorting heuristics to reduce area cost. Additionally, the automated mapping approach incorporates the granularity and fabric constraints that are imposed in order to decrease the number of metal wires used for programming the SET array and for supplying the input signals, respectively [2].

We conduct experiments on a set of MCNC benchmarks [10]. The experimental results show that the proposed approach can complete mapping within 1 second for most of the benchmarks and presents competitive results to prior manual mapping results. The main contribution of this work is proposing an automated synthesis tool for the promising energy-efficient SET array architecture.

The rest of this paper is organized as follows: Section II uses an example to demonstrate the problem considered in this paper, and introduces some notations. Section III presents the proposed mapping approach. Section IV discusses and addresses two mapping constraints. Finally, the experimental results and conclusion are presented in

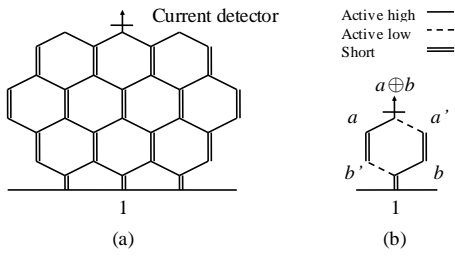


Fig. 2. (a) A SET array fabric. (b) An example of a XOR b.

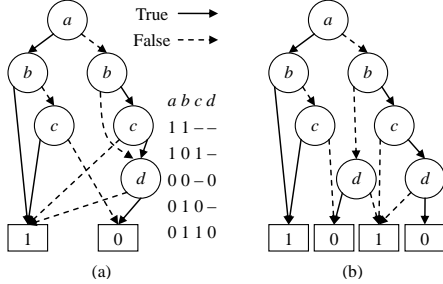


Fig. 3. An example of eliminating the crossing edges in an ROBDD by node duplication. (a) The original ROBDD. (b) The resultant BDD.

Sections V and VI.

II. BACKGROUND

A. An example

A SET array can be presented as a graph composed of hexagons. As shown in Fig. 2(a), like the hexagonal fabric mentioned above, there is a current detector at the top that measures the current coming from the bottom of the hexagonal fabric. All the vertical edges of the hexagons are electrical short. All the sloping edges can be configured as active high, active low, short or open. An active high edge is controlled by a variable x . It is conducting and non-conducting when $x = 1$ and $x = 0$, respectively. Conversely, an active low edge is an electrical opposite of an active high edge and it is controlled by a variable x' .

A Boolean function can be implemented using a SET array. All the active edges at the same row of the hexagonal fabric are controlled by a single variable, i.e., a primary input (PI). They determine whether there exists a path for the current to pass through, and thus, be detected at the top. If so, the functional output of the array is 1; otherwise, it is 0. For example, Fig. 2(b) shows a SET array implementing $a \text{ XOR } b$. When $a = 1$ and $b = 0$, the current can be detected by passing through the left path. However, if $a = 1$ and $b = 1$, the current cannot be detected.

Thus, the addressed problem of this work is synthesizing a given Boolean function into a SET array with minimized area, i.e., the number of configured hexagons.

Previous work [2] tries to manually map a Boolean function by directly mapping its BDD into a SET array. However, the mapping process could be very complicated due to the structural difference of a BDD and a SET array. For example, Fig. 3(a) shows an ROBDD that has some crossing edges. Since a SET array is a planar architecture, much effort is required to avoid having the crossing edges in the ROBDD when mapping it into a SET array. Node duplication could be a trivial method for solving this crossing edge issue while not considering the area overhead. For example, we can eliminate the crossing edges in Fig. 3(a) by duplicating the node d and the terminals. The resultant BDD having more nodes is shown in Fig. 3(b). In addition, determining the exact location of each ROBDD node in a SET array is a challenge. Thus, to address this problem, we propose a product term-based method. It first collects all the paths that lead to the terminal 1 in the ROBDD, i.e., product terms. Then, it maps each product term into a path in the SET array. The proposed method

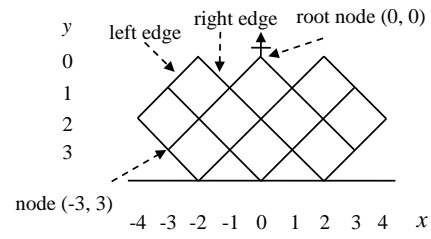


Fig. 4. An abstract diamond fabric.

simultaneously avoids the crossing edge and the BDD node mapping issues.

For example, the product terms of $a \text{ XOR } b$ are 10 and 01. Using the proposed method, we first map 10 and then 01. Finally, we obtain the resultant SET array as shown in Fig. 2(b), where the left path is configured for 10 and the right path is for 01.

B. Notations

For ease of discussion, we use an abstract graph to present a SET array. Compared to Fig. 2(a), only the configurable edges are preserved as shown in Fig. 4. In this diamond fabric, each node n , i.e., the root of a pair of left and right edges, has a unique location (x, y) . Based on the root node located at $(0, 0)$, which is below the current detector, the y value increases from top to bottom. The x value increases and decreases from center to right and left, respectively.

For simplification, let $n.left$ and $n.right$ denote the status of the left and right edges of a node n , respectively. The status could be *empty*, *high*, *low*, *short*, or *open*. *empty* indicates the edge is not configured yet (is used primarily for algorithm illustration). *high*, *low*, *short*, and *open* indicate the edge is configured as active high, active low, short, and open, respectively. Additionally, let $n_{(x,y)}$ denote the node located at (x, y) .

III. AUTOMATED MAPPING

In this section, we first discuss the motivation of our method. Next, we introduce two key mapping procedures. Finally, the overall flow is presented. Here, we first assume that each edge can be configured independently without any constraint. In the next section, we will extend our mapping method considering the granularity and fabric constraints.

To simplify the mapping problem, we divide a ROBDD into a set of paths that lead to the terminal 1 in this ROBDD. These paths represent the product terms. Then we map these product terms instead of a whole ROBDD. The overall mapping flow includes two important steps: product term computation and product term mapping.

A. Product term computation

To compute the product terms of a given Boolean function, we first build its ROBDD. Next, we compute the product terms by traversing the ROBDD to collect the paths that lead to the terminal 1. For example, Fig. 3(a) shows an ROBDD. There are four PIs: a , b , c , and d . We can collect the product terms by traversing the paths leading to the terminal 1 with depth-first search (DFS). They are $abcd = \{11--, 101-, 00-0, 010-, 0110\}$. In this work, we use the CUDD package [9] to build ROBDDs and collect the product terms.

Since we map the product terms one by one and each product term corresponds to a path in the SET array, both the number and the order of product terms we consider could affect the mapping results. In general, more product terms result in a larger area. Thus, before collecting product terms, we will try to minimize the ROBDD by performing BDD reordering. However, because the BDD reordering operation is used to minimize the number of BDD nodes instead of product terms, we only adopt the reordering result when the number of product terms is reduced. For most benchmarks considered, BDD reordering reduces the number of product terms. We use the BDD reordering heuristic

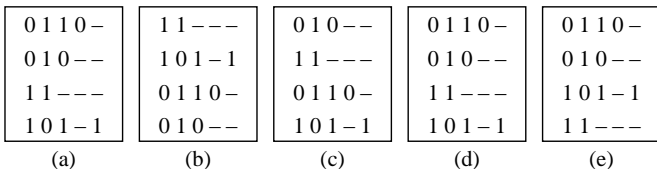


Fig. 5. Four different sorting results. (a) Original. (b) *LexSort*. (c) *InertiaSort*. (d) *ForInertiaSort*. (e) *BackForInertiaSort*.

CUDD_REORDER_SYMM_SIFT in the *CUDD* package as it achieves better reduction for most benchmarks compared to the other heuristics provided by the *CUDD* package.

Note that although there are other methods, like Espresso [11], which could compute more concise product terms, we choose to use the BDD-based computation method. This is because it ensures that each minterm appears in only one product term. As a result, when we map each product term into a path in the SET array, exactly one path is conducting at a time. For example, if we use Espresso to compute the product terms of the Boolean function in Fig. 3(a), there are only four product terms: $1-1-$, $010-$, $0--0$, and $110-$. However, since the minterm 0100 is involved in $010-$ and $0--0$, when 0100 is applied to the SET array, there are two conducting paths: one for $010-$ and the other one for $0--0$. Having multiple conducting paths leads to a higher fanout number that is not preferred for SET devices that have a low-drive strength.

As for sorting the product terms, we propose four different sorting methods: *LexSort*, *InertiaSort*, *ForInertiaSort*, and *BackForInertiaSort*. Our objective is to make the configured paths of different product terms share as many edges as possible. The details of the proposed sorting methods are as follows:

1) *LexSort*: We sort product terms by comparing the bit values from the first bit with the relationship: $- > 1 > 0$. For example, Fig. 5(b) shows the sorting result of the product terms in Fig. 5(a). Using *LexSort*, two product terms that have continuous bit value matches from the first bit will be adjacent. As a result, starting from the root node, the adjacent product terms could possibly share the edges for the continuous matching bits.

2) *InertiaSort*: Each product term has an inertia value that is the number of bit value matches with all the other product terms. We sort product terms from large to small by the inertia values. Fig. 5(c) shows the sorting result. The inertia value of the first product term in Fig. 5(c) is $1 + 2 + 0 + 2 + 2 = 7$. The inertia values of the other product terms are 7, 6, and 4, respectively. Using *InertiaSort*, the product terms that have more bit value matches with others will be mapped earlier than those having fewer bit value matches. After a product term having a larger inertia value is mapped, more product terms could possibly reuse its configured edges due to the higher bit value matches.

3) *ForInertiaSort*: Unlike the inertia value, a product term's forward inertia value is the number of continuous bit value matches from the first bit with all the other product terms. We sort product terms from large to small by the forward inertia values. Fig. 5(d) shows the sorting result. The forward inertia value of the first product term in Fig. 5(d) is $1 + 1 = 2$. This is because only the second product term has two continuous bit value matches with it. The forward inertia values of the other product terms are 2, 1, and 1, respectively. Using *ForInertiaSort*, the product terms that have more continuous bit value matches from the first bit with others will be mapped earlier. The reason behind this heuristic is that we expect many shared edges to start from the root nodes and to be connected (continuous bits).

4) *BackForInertiaSort*: Conversely, a product term's backward inertia value is the number of continuous bit value matches from the last bit to the first bit with all the other product terms. We first sort product terms from small to large by the backward inertia values. Then, we sort them again from large to small by the forward inertia values. The sorting result is shown in Fig. 5(e). Unlike the result in

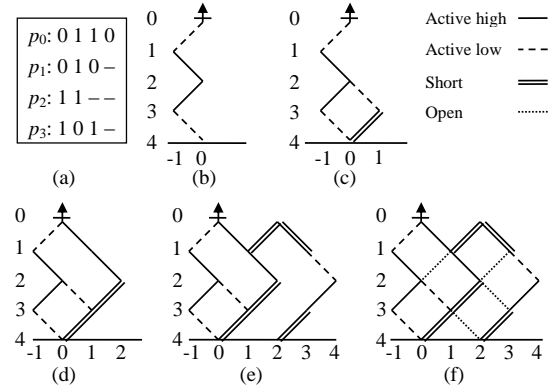


Fig. 6. A mapping example. (a) Product terms. (b) The mapping result of p_0 . (c) The mapping result of $p_0 + p_1$. (d) The mapping result of $p_0 + p_1 + p_2$. (e) The mapping result of $p_0 + p_1 + p_2 + p_3$. (f) The final mapping result.

Fig. 5(d), the third product term has a smaller backward inertia value. *BackForInertiaSort* is used to complement *ForInertiaSort*. We use the backward inertia values to distinguish the product terms having the same forward inertia values, and expect they could share edges near the leaf nodes.

B. Product term mapping

After product terms computation, we start to map these product terms. Our objective is to configure a path in the SET array for each product term, and avoid constructing a path that corresponds to an invalid product term.

Given a product term p , we start from the root node, and find or configure an edge for each bit in p from the first bit to the last bit. The mapping rules are as follows: When the bit value under consideration is 1 (or 0), we find an active high (or low) edge for it if applicable; otherwise, we configure an edge as active high (or low) for it. However, if the bit value is $-$, we find a *short* edge if applicable or configure an edge as *short* for it. After all the product terms are mapped, we finally configure the edges that are not configured yet as *open*.

We use an example in Fig. 6 to demonstrate the mapping approach. There are four product terms, $p_0 = 0110$, $p_1 = 010-$, $p_2 = 11--$, and $p_3 = 101-$, sorted by *ForInertiaSort* as shown in Fig. 6(a). First, let us consider p_0 . Starting from the root node $n_{(0,0)}$, we first configure $n_{(0,0).left}$ as *low* for the first bit 0. Next, we configure $n_{(-1,1).right}$ as *high* for the second bit 1. Using the same method, we configure $n_{(0,2).left}$ and $n_{(-1,3).right}$ as *high* and *low* for the last two bits 10, respectively. The mapping result is shown in Fig. 6(b). Here, the decision of configuring the left edge or the right edge of a node depends on its location (x, y) . If $x < 0$, we first try to configure its right edge. If inapplicable, we then try to configure its left edge. Conversely, if $x \geq 0$, we try the left edge first and then the right edge.

Next, for p_1 , because the first two bits are the same as that of the first product term, we partially reuse this mapping result. Next, we configure $n_{(0,2).right}$ as *low* and $n_{(1,3).left}$ as *short* for the last two bits $0-$, respectively. The mapping result is shown in Fig. 6(c).

For p_2 , after we configure $n_{(0,0).right}$ as *high* for the first bit 1, we first try to configure $n_{(1,1).left}$ as *high* for the second bit 1. However, since $n_{(0,2).left}$ and $n_{(0,2).right}$ have been configured as *high* and *low* that are inconsistent to the third bit $-$, we undo the configuration of $n_{(1,1).left}$ and then configure $n_{(1,1).right}$ as *high* for the second bit 1. Finally, both $n_{(2,2).left}$ and $n_{(1,3).left}$ are configured as *short* for the last two bits $--$. The mapping result is shown in Fig. 6(d).

Next, let us consider p_3 . After finding $n_{(0,0).right} = \text{high}$ for the first bit 1, we do not configure $n_{(1,1).left}$ as *low* for the second bit 0. This is because if we do so, there will exist a path $n_{(0,0)} \rightarrow n_{(1,1)} \rightarrow n_{(0,2)} \rightarrow n_{(1,3)} \rightarrow n_{(0,4)}$, which corresponds to an invalid product term $100-$. Additionally, since $n_{(1,1).right}$ has been configured as 1, we expand the structure by configuring both $n_{(2,0).left}$

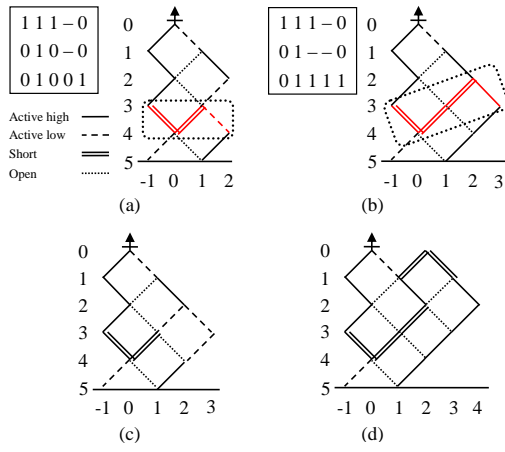


Fig. 7. Incorrect mapping examples.

and $n_{(2,0).right}$ as *short*, and start from $n_{(3,1)}$ for the last three bits. The mapping result is shown in Fig. 6(e). Finally, we configure all the non-configured edges as *open*, and obtain the final mapping result in Fig. 6(f).

To avoid creating an invalid path, we need to prevent two paths from merging and then branching during mapping. Thus, when we detect a merging node, like $n_{(0,2)}$ for p_2 or p_3 , we will check if there exists only one path from $n_{(0,2)}$. If not, there possibly exists an invalid path. Thus, we prevent the paths from merging. With this checking rule, each path from top to bottom exactly corresponds to one product term. In addition, from the viewpoint of conducting paths, this checking rule is not enough. We have to add another rule considering the conducting path issue. Fig. 7(a), (b) show two mapping examples, which are incorrect while satisfying the merging and branching rule.

In Fig. 7(a), when the input pattern is 11101, which is not a minterm, the current can be detected at the top. This is because the right edge of $n_{(-1,3)}$, the left edge of $n_{(1,3)}$, and the right edge of $n_{(1,3)}$ as highlighted are conducting simultaneously. This partial conducting path forms like a bridge that connects two paths such that the current can pass through the path $n_{(1,5)} \rightarrow n_{(2,4)} \rightarrow n_{(1,3)} \rightarrow n_{(0,4)} \rightarrow n_{(-1,3)} \rightarrow n_{(0,2)} \rightarrow n_{(-1,1)} \rightarrow n_{(0,0)}$. In addition, a partial conducting path could be composed of the edges at the different rows. For example, Fig. 7(b) shows a partial conducting path that crosses two rows as highlighted. This path, $n_{(3,3)} \rightarrow n_{(2,2)} \rightarrow n_{(1,3)} \rightarrow n_{(0,4)} \rightarrow n_{(-1,3)}$, constructs an invalid conducting path for the input pattern 11111.

The necessary condition for causing a partial conducting path is that there are at least two adjacent conducting edges. Thus, if a configuration results in two adjacent edges that could be conducting simultaneously, we check whether they create a partial conducting path that constructs an invalid path. If so, we avoid this configuration. Fig. 7(c) and Fig. 7(d) show the correct mapping results for the product terms in Fig. 7(a) and Fig. 7(b), respectively.

Additionally, because the root node has only two edges (left and right), in order to successfully map all product terms, three kinds of bit values, 0, 1, and $-$, cannot simultaneously appear as the first bits of different product terms. If they appear simultaneously, we divide each product term having $-$ in the first bit into two product terms before mapping: one begins with 0 and the other begins with 1. Furthermore, if there are two different kinds of bit values appearing in the first bits of all product terms, we will initially configure $n_{(0,0).left}$ and $n_{(0,0).right}$ based on the first bit values to ensure $n_{(0,0).left} \neq n_{(0,0).right}$ for successfully mapping all product terms.

Fig. 8 shows the proposed recursive algorithm of product term mapping. In the main function, **Mapping()**, we first configure $n_{(0,0).left}$ and $n_{(0,0).right}$ based on the first bit values of all the product terms to ensure $n_{(0,0).left} \neq n_{(0,0).right}$, when there are two different first

Mapping(set PTs) // PTs: product terms

1. Configure $n_{(0,0).left}$ and $n_{(0,0).right}$ based on the first bit values of the product terms in *PTs*;
2. For each product term t in *PTs*
 - 2.1. If (**LeftConfigure**($t, 0, 0$)), **continue**;
 - 2.2. If (**RightConfigure**($t, 0, 0$)), **continue**;
 - 2.3. **Expand**(t);
3. Configure all the edges that are not configured yet as *open*;

bool LeftConfigure(productterm t , int x , int y)

1. If $n_{(x,y).left}$ is inconsistent to the y^{th} bit in t , **return** 0;
2. If $n_{(x-1,y+1)}$ is a merging node and there is more than one path from $n_{(x-1,y+1)}$, **return** 0;
3. If the configuration of $n_{(x,y).left}$ will result in a partial conducting path that constructs an invalid conducting path, **return** 0;
4. If $n_{(x,y).left}$ is *empty*, configure it based on the mapping rules;
5. If $(x-1 < 0)$
 - 5.1. If (**RightConfigure**($t, x-1, y+1$)), **return** 1;
 - 5.2. If (**LeftConfigure**($t, x-1, y+1$)), **return** 1;
6. If $(x-1 \geq 0)$
 - 6.1. If (**LeftConfigure**($t, x-1, y+1$)), **return** 1;
 - 6.2. If (**RightConfigure**($t, x-1, y+1$)), **return** 1;
7. Undo $n_{(x,y).left}$ if necessary, and **return** 0;

bool RightConfigure(productterm t , int x , int y)

1. If $n_{(x,y).right}$ is inconsistent to the y^{th} bit in t , **return** 0;
2. If $n_{(x+1,y+1)}$ is a merging node and there is more than one path from $n_{(x+1,y+1)}$, **return** 0;
3. If the configuration of $n_{(x,y).right}$ will result in a partial conducting path that constructs an invalid conducting path, **return** 0;
4. If $n_{(x,y).right}$ is *empty*, configure it based on the mapping rules;
5. If $(x-1 < 0)$
 - 5.1. If (**RightConfigure**($t, x+1, y+1$)), **return** 1;
 - 5.2. If (**LeftConfigure**($t, x+1, y+1$)), **return** 1;
6. If $(x-1 \geq 0)$
 - 6.1. If (**LeftConfigure**($t, x+1, y+1$)), **return** 1;
 - 6.2. If (**RightConfigure**($t, x+1, y+1$)), **return** 1;
7. Undo $n_{(x,y).right}$ if necessary, and **return** 0;

bool Expand(productterm t)

1. Determine the expansion direction (left or right) based on the first bit in t .
2. If the expansion direction is left, $x = -2$; otherwise, $x = 2$;
3. While(1)
 - 3.1. Configure $n_{(x,0).left}$ and $n_{(x,0).right}$ as *short* if they are *empty*;
 - 3.2. If $(x-1 < 0)$
 - 3.2.1. If (**RightConfigure**($t, x-1, 1$)), **return** 1;
 - 3.2.2. If (**LeftConfigure**($t, x-1, 1$)), **return** 1;
 - 3.2.3. $x = x-2$;
 - 3.3. If $(x-1 \geq 0)$
 - 3.3.1. If (**LeftConfigure**($t, x+1, 1$)), **return** 1;
 - 3.3.2. If (**RightConfigure**($t, x+1, 1$)), **return** 1;
 - 3.3.3. $x = x+2$;

Fig. 8. The algorithm of product term mapping.

bit values. Next, we start to configure all the product terms from the root node $n_{(0,0)}$. For each product term t , we use a DFS-like method to construct a path for it. **LeftConfigure()** and **RightConfigure()** configure the left and right edges of a node, respectively. If we cannot successfully map t from $n_{(0,0)}$, we expand the structure by using **Expand()**. Finally, we configure all the edges that are not configured yet as *open*.

In **LeftConfigure()**, we first check if the left edge of a node $n_{(x,y)}$ is inconsistent to the y^{th} bit in t . They are inconsistent when $n_{(x,y).left}$ is configured and they do not satisfy the mapping rules: *high* for 1, *low* for 0, and *short* for $-$. If so, we return to the last procedure to consider the other edges or nodes. If they are consistent, we then check whether the situation that two paths merge and then branch occurs. Here, $n_{(x-1,y+1)}$ is the sink node of the left edge of $n_{(x,y)}$. If $n_{(x-1,y+1)}$ is a merging node and there is more than one path from it, there exists two merging and branching paths. If not, we further check if the configuration of $n_{(x,y).left}$ will result in a partial conducting path that constructs an invalid conducting path. If not, we then configure $n_{(x,y).left}$ based on the mapping rules if $n_{(x,y).left}$ is *empty*. Next, we perform **LeftConfigure()** or **RightConfigure()** on $n_{(x-1,y+1)}$ for the next bit based on the value of x . However, if we finally fail to map t

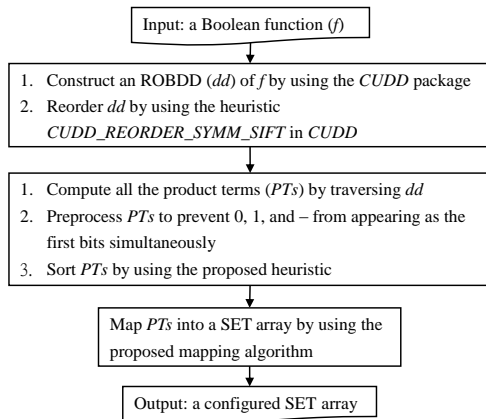


Fig. 9. The overall mapping flow.

due to the configuration of $n_{(x,y).left}$, we undo it and then consider the other edges or nodes. **RightConfigure()** is similar to **LeftConfigure()**, but considers the configuration of a right edge.

In **Expand()**, we first determine the expansion direction. For example, suppose $n_{(x,y).left}$ is *high*. If the first bit of t is 1, the expansion direction is left; otherwise, it is right. The direction also determines the initial value of x . x is -2 when the direction is left; otherwise, it is 2. Next, we start to construct a path using the same method for the second bit to the last bit in t . First, we configure $n_{(x,0).left}$ and $n_{(x,0).right}$ as *short*. Second, we determine the new root node for this configuration. It is $n_{(x-1,1)}$ if the direction is left; otherwise, it is $n_{(x+1,1)}$. However, if we still fail to map t , we expand the structure again and x is increased or decreased by 2 based on the expansion direction.

C. Overall mapping flow

Fig. 9 shows the overall mapping flow. The input is a Boolean function (f). In step 1, we first construct an ROBDD (dd) of f by using the CUDD package. Then, we reorder dd by using the heuristic `CUDD_REORDER_SYMM_SIFT` in `cudd`. In step 2, we first compute all the product terms (PTs) of f by traversing dd . Next, we preprocess PTs to prevent 0, 1, and $-$ from appearing as the first bits simultaneously. Finally, we sort PTs by using the proposed heuristic. In step 3, we map PTs into a SET array by using the proposed mapping algorithm. Finally, we get a configured SET array.

IV. MAPPING CONSTRAINTS

In this section, we discuss two mapping constraints, granularity and fabric constraints, which limit the status combinations of a pair of left and right edges of a node.

A. Configuration granularity constraint

The configuration circuitry, which involves metal wires, is used to determine whether the SET is in open, short or active mode. Since the metal wire pitches can lead to lower logic density, the same wires can be used to program multiple SETs simultaneously to reduce the metal wires. However, it also imposes a restriction that they are programmed to the same state. Consequently, when two edges of the SET array share their configuration circuitry, the combination of $n.left$ and $n.right$, ($n.left, n.right$), must be one of (*high, low*), (*low, high*), (*short, short*), and (*open, open*), where n is a node in the SET array.

According to the constraint, when one edge of the root node is configured as *short*, the other edge must be *short* as well. Thus, before mapping, unless the first bits of all the product terms are $-$, we divide each product term whose first bit is $-$ into two product terms: one has the first bit 0 and the other one has the first bit 1.

The algorithm in Fig. 8 maps product terms without any constraint. It can be easily extended to consider the granularity constraint by modifying the configuration method. Originally, two edges of a node

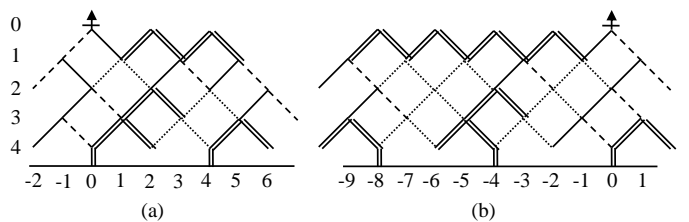


Fig. 10. The mapping results with (a) granularity constraint, and (b) fabric constraint.

are configured separately. To consider this granularity constraint, we configure them at the same time. For example, when we configure one edge of a node as *high* (or *low*), we also configure the other edge as *low* (or *high*). Similarly, when one edge is *short*, the other edge is *short* as well.

Fig. 10(a) shows the mapping result for the same set of product terms in Fig. 6(a) with the granularity constraint. Here, not all paths are connected to the current source. This is because we configure two edges of a node for each bit at a time. When we finish mapping the last bit of a product term, there are two paths are constructed simultaneously. Thus, we only connect the path with respect to the product term to the current source.

Since two edges are configured simultaneously, we check if merging and branching paths occur for both of these two edge configurations to avoid creating invalid paths. Additionally, because not all configured edges finally lead to the current source, when a partial conducting path is detected, we further check if it really causes an invalid conducting path. If not, it is allowed. For brevity, we omit the detailed mapping algorithm considering the granularity constraint.

B. Fabric constraint

In SET array implementation, the inputs to the active edges in one row need to be hard wired and they are not configurable. Each active edge is connected to either x or its complement x' wires for the row. The pattern of connections of x and x' in a row defines the SET fabric. For example, using x to control all left edges and x' to control the right edges results in the symmetric fabric proposed in [2]. In such an array, both (*high, low*) and (*low, high*) cannot simultaneously appear at the same row in a SET array. Note that the entire row pattern of (*high, low*) (or (*low, high*)) can be changed to (*low, high*) (or (*high, low*)) by swapping the normal value and its complement in the control input signals for the row.

To satisfy this symmetric fabric constraint, we need to identify which combination ((*high, low*) or (*low, high*)) appears at a certain row. One method is to follow the first configuration result at the row. For example, if (*high, low*) is first configured at a row, we then do not configure (*low, high*) at this row. Another easy method is to allow only one of (*high, low*) and (*low, high*) to appear in a SET array. For example, for a bit value 1 or 0, we can always configure the left edge as *high* and the right edge as *low*, i.e., only (*high, low*) is allowed. For simplification, we use the second method in this work.

Fig. 10(b) shows the mapping result for the same set of product terms in Fig. 6(a) considering the fabric constraint. In this example, only (*high, low*), (*short, short*), and (*open, open*) are allowed.

V. EXPERIMENTAL RESULTS

We implemented the algorithm in C language. The experiments were conducted on a 2.67 GHz Linux platform (Red Hat 5.5). The benchmarks are from the MCNC benchmark suite [10]. For each benchmark, we separately map the Boolean function of each primary output (PO), and measure the total number of configured hexagons and the total CPU time. In the experiments, we compare different product term sorting heuristics and mapping constraints.

Table I summarizes the experimental results. Column 1 lists the benchmarks. Except the *C17* benchmark, all the benchmarks have the

TABLE I
THE EXPERIMENTAL RESULTS OF USING DIFFERENT PRODUCT TERM
SORTING HEURISTICS AND MAPPING CONSTRAINTS.

Bench.	PI	PO	PT	Constraint-free				Granu.	Fabric
				Lex	Inert.	FlInert.	BFInert.	FlInert.	FlInert.
C17	5	2	8	*15	*15	16	16	41	42
cm138a	6	8	48	*71	102	72	72	272	336
x2	10	7	33	*142	152	146	148	689	755
cm85a	11	3	49	220	185	172	*170	590	521
cm151a	12	2	25	406	427	*400	*400	824	1018
cm162a	14	5	37	*244	308	285	273	980	1134
cu	14	11	24	217	218	217	*208	572	612
cmb	16	4	26	95	138	*80	*80	497	199
cm163a	16	5	27	226	*196	229	227	828	881
pm1	16	13	41	262	257	256	*254	887	863
pcler	19	9	45	274	*259	268	268	1281	1595
set	19	15	142	1438	*1270	1449	1352	3808	4494
cc	21	20	57	573	595	*545	555	2060	2250
i1	25	16	38	493	479	*463	*463	1635	1356
lal	26	19	160	1741	1949	*1614	1627	7519	8542
pcler8	27	17	68	660	764	*655	*655	2548	2424
frg1	28	3	399	5706	5362	5286	*5119	11151	13056
c8	28	18	94	*741	748	782	773	4502	4721
term1	34	10	1246	22635	23864	*21404	23016	59940	79049
count	35	16	184	1563	1523	*1099	1221	12489	14282
unreg	36	16	64	1046	921	949	*910	4416	4520
b9	41	21	352	*5395	8102	5588	5841	21502	20503
cht	47	36	92	1747	*1704	1747	1949	7563	7205
apex7	49	37	1440	32034	41936	*30147	31843	98878	129175
example2	85	66	430	8529	9082	*8510	8608	47378	47979
Total				86473	100556	82379	86048	292850	347512
Best count				6	5	10	9		

crossing edge issue in their ROBDDs. Directly mapping each of these ROBDDs into a SET array could be very difficult. Columns 2 and 3 list the number of PIs and POs in each benchmark, respectively. Column 4 lists the number of computed product terms. The remaining columns list the mapping results in terms of the number of hexagons by using different sorting heuristics and constraints. The number marked with “*” means that it is the best result among all sorting heuristics. Columns 5 to 8 are the constraint-free mapping results by using *LexSort*, *InertiaSort*, *ForInertiaSort*, and *BackForInertiaSort*, respectively. Columns 9 and 10 are the mapping results applying the granularity and fabric constraints by using *ForInertiaSort* only. This is because the *ForInertiaSort* heuristic has better results for considering all benchmarks or large benchmarks in the experiments. We omit the results by using the other sorting heuristics due to page limit.

For example, the *C17* benchmark has 5 PIs and 2 POs. The total number of computed product terms are 8. For constraint-free mapping, the mapping algorithm configured 15, 15, 16, and 16 hexagons to implement the benchmark function, when respectively using *LexSort*, *InertiaSort*, *ForInertiaSort*, and *BackForInertiaSort*. For the granularity and fabric constraints, the mapping algorithm with *ForInertiaSort* configured 41 and 42 hexagons, respectively.

According to Table I, there is no a specific sorting heuristic that completely outperforms the others for all the benchmarks. By all accounts, *ForInertiaSort* results in the best mapping for considering all benchmarks. Additionally, when the constraints are considered, the number of configured hexagons increases. This is because the number of edges shared by different paths decreases. As for the CPU time, the proposed method can map each benchmark within 1 second except the *term1* and *apex7* benchmarks that spent approximately 6 seconds.

Furthermore, we compare the experimental results with that reported in a previous work, which manually maps a BDD into a SET array [2]. Table II summarizes the comparison on the number of configured hexagons for a set of small benchmarks which are able to be manually mapped. According to the results, the proposed automated mapping approach, either with the fabric constraint or not, requires less hexagons for implementing the *2-bit adder* benchmark while needs more hexagons for implementing the *C17* and the *4-bit parity* benchmarks, compared to the manual mapping approach.

In the manual mapping approach, it simultaneously maps multiple

TABLE II
THE EXPERIMENTAL RESULTS OF MANUAL MAPPING AND AUTOMATED
MAPPING.

Benchmark	Manual mapping [2]		Automated mapping		
	Constraint-free	Fabric	Constraint-free	Fabric	Granu.*
C17	12	20	15	42	41
2 to 4 decoder	4	4	4	4	4
4-bit parity	6	6	9	14	14
2-bit adder	51	128	13	42	34

* The previous work [2] does not report the results for considering the granularity constraint and is a manual process that does not scale for large circuits.

outputs that could share some sub-circuits into one SET array. When many hexagons are shared by the different outputs, this approach can save much area in terms of the number of hexagons. This is the reason that the results of the *C17* and the *4-bit parity* benchmarks are better than ours. In the future, we will focus on extending our approach for automated mapping of multiple outputs with maximized shared hexagons.

However, since multiple outputs are mapped into one SET array, the manual mapping approach may require extra spacing hexagons to isolate the conducting paths of different outputs, and thus, results in a larger area if a poor variable ordering is used. The results of the *2-bit adder* benchmark demonstrate this situation. The proposed approach saves more than 2/3 area in terms of the number of hexagons compared to the manual mapping approach. Thus, the proposed automated mapping method is an efficient and effective approach that saves much manual effort on the SET array realization.

VI. CONCLUSION

In this paper, we propose a product-term-based approach that can efficiently map a Boolean function into a SET array. It solves the problem of automatically mapping a BDD into a SET array that previous works suffer from. The proposed approach simplifies the mapping problem by transforming a BDD into a set of product terms, and then individually mapping these product terms. Additionally, four product term sorting heuristics are proposed to enrich the approach. The granularity and fabric constraints can also be handled by the proposed approach. The experimental results show its effectiveness and efficiency of mapping a set of MCNC benchmarks. Our automated mapping is a key enabler for using the promising BDD technology.

REFERENCES

- [1] R. Bryant, “Graph-based Algorithms for Boolean Function Manipulation,” *IEEE Trans. Computers*, vol. 35, pp. 677-691, Aug. 1986.
- [2] S. Eachempati, V. Saripalli, V. Narayanan, and S. Datta, “Reconfigurable Bdd-based Quantum Circuits,” in *Proc. Int. Symp. on Nanoscale Architectures*, 2008, pp. 61-67.
- [3] H. Hasegawa and S. Kasai, “Hexagonal Binary Decision Quantum Logic Circuits Using Schottky In-Plane and Wrap Gate Control of GaAs and InGaAs Nanowires,” *Physica E: Low-dimensional Systems and Nanostructures*, vol. 11, pp. 149-154, Oct. 2001.
- [4] S. Kasai, M. Yumoto, and H. Hasegawa, “Fabrication of GaAs-based Integrated 2-bit Half and Full Adders by Novel Hexagonal BDD Quantum Circuit Approach,” in *Proc. Int. Symp. on Semiconductor Device Research*, 2001, pp. 622-625.
- [5] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi, *Low Power Methodology Manual: For System-on-Chip Design*, Springer, 2007.
- [6] S. W. Keckler, K. Olukotun, and H. P. Hofstee, *Multicore Processors and Systems*, Springer, 2009.
- [7] C. Piguet, *Low-power CMOS Circuits: Technology, Logic Design and CAD Tools*, CRC Press, 2006.
- [8] V. Saripalli, L. Liu, S. Datta, and V. Narayanan, “Energy-Delay Performance of Nanoscale Transistors Exhibiting Single Electron Behavior and Associated Logic Circuits,” *Journal of Low Power Electronics (JOLPE)*, vol. 6, pp. 415-428, 2010.
- [9] F. Somenzi, *CUDD: CU decision diagram package - release 2.4.2*, 2009. <http://vlsi.colorado.edu/~fabio/CUDD/>
- [10] S. Yang, “Logic Synthesis and Optimization Benchmarks, Version 3.0,” *Tech. Report, Microelectronics Center of North Carolina*, 1991.
- [11] <http://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm>
- [12] <http://www.intel.com/go/terascale/>