BDD-Based Synthesis of Reconfigurable Single-Electron Transistor Arrays

Zheng Zhao[1], Chian-Wei Liu[2], Chun-Yao Wang[2], and Weikang Qian[1]

[1]University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, Shanghai, China.
[2]Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, R.O.C.

***Abstract*—Single-electron transistor (SET) is an ultra-low power device, which has been demonstrated as a promising alternative for CMOS devices in reducing power consumption. A suitable structure for realizing logic function using SET is a binary decision diagram (BDD)-based SET array. Previous works proposed product term-based automated synthesis methods to map a given logic function onto an SET array. In this work, we propose a novel BDD-based synthesis method that exploits the structure similarity between an SET array and a BDD. Our method transforms a BDD of a Boolean function into a planar graph and further maps the graph onto an SET array. Experiment results showed that compared to the state-of-the-art synthesis method, our method saves 51% in area on average and is more than 16 times faster.**

I. Introduction

With continued scaling of CMOS technology, power consumption has emerged as a major obstacle to sustaining Moore's law. One solution to this problem is to explore novel devices as substitutes to CMOS device. Among many nanoscale devices researchers have proposed, single-electron transistors (SETs) have been considered as a promising choice due to their ultra-low power operation involving only a few electrons [1] [2].

However, the limited number of electrons during the operation of the device also significantly reduces its driving capability. To address this problem, a binary decision diagram (BDD)-based architecture was proposed for implementing logic functions using SETs [3] [4] [5]. The interconnection of SET devices in this architecture forms a 2-dimensional array of hexagons, as shown in Fig. 1a. To enhance the flexibility of the BDD-based SET array, a recent work also proposed a reconfigurable realization using wrap gate tunable tunnel barriers [6].

As we will show in Section II, the BDD-based SET architecture has special properties and constraints in realizing logic functions. For example, the Boolean function realized by the SET array is based on the set of conducting path from the top row of the SET array to its bottom row. In order to implement large-scale digital circuit with the SET array, a few automatic synthesis methods were proposed [7] [8] [9]. These methods start from the BDD representation of a given Boolean function. However, they do not directly map the BDD onto the SET array, since an SET array is a planar graph [1], while a BDD is not. Instead, they extract the product terms from the BDD and then map these product terms one by one onto the SET array.

In this work, we propose a new algorithm to synthesize SET arrays. Since the previous approaches separate the BDD into disjoint product terms, they may fail to exploit the efficient structural sharing presented in the BDD to be mapped. To utilize such structural sharing, our method directly transforms a BDD into a planar graph and then maps it onto an SET array. Experiments demonstrated that compared to the state-of-the-art product term-based mapping method, our approach greatly saves the area of the SET array.

[1]A *planar graph* is a graph that can be drawn in a way that no edges cross each other.

The remainder of this paper is organized as follows. Section II discusses the background, the related works, and the motivation for our work. Section III shows our new algorithm for synthesizing constraint-free SET arrays. Section IV extends the new algorithm to handle two technology constraints of SET arrays. Section V presents the experimental results. Section VI concludes the paper.

II. Background and Related Works

A. Single-Electron Transistor Array

An SET array can be presented as a graph of hexagons as shown in Fig. 1a. In the hexagonal fabric, all the vertical edges are electrical short. All the sloping edges can be configured as active high, active low, short, or open. We call these edges configured as either active high or active low the *active edges*. All the active edges on the same row are controlled by the same primary input (PI) x. Active high edges are controlled by x and active low edges by its complement x'. Active high edges are conducting when x is logic 1 and non-conducting when x is logic 0. Active low edges are opposite to active high edges: they are conducting when x is logic 0 and non-conducting when x is logic 1.

The output of the function implemented by an SET array is decided by a current detector at the top of the SET array, which measures the current coming from the current source at the bottom of the SET array. If there exists a path for the current to pass through and thus be detected at the top, the output is 1; otherwise it is 0. For example, Fig. 1b shows an SET array implementing $a \oplus b$. When $a = 1$ and $b = 0$, the left path is conducting and a current can be detected. However, if $a = 1$ and $b = 1$, no current is detected.

Since all the vertical edges are short, we can omit them for ease of discussion and only keep the configurable edges, as shown in Fig. 2. This representation, known as the diamond fabric [7], is used in our following discussion. In this diamond fabric, each node n, i.e., the root of a pair of left and right edges, has a unique coordinate (x, y), denoted as $Coord(n)$ to indicate its location. Moreover, we use $Coord(n).x$ and $Coord(n).y$ to denote the x coordinate and the y coordinate of the node n, respectively. The location of the current detector is the origin of the coordinate system. The y value increases from top to bottom, while the x value increases from left to right. Two neighboring nodes on a row have their x coordinates differ by 2. The configuration of the two edges rooted at a node is indicated in a pair of parentheses, such as (high,low), (open,open), etc., where the first one in the pair indicates the state of the left edge and the second one indicates the state of the right edge.

Due to the low driving capability of an SET device, an SET array has an important constraint that for any variable assignment, there is at most one conducting path.

A reconfigurable SET array also imposes two technology constraints on the configuration of edges. The first is the fabric constraint. In an SET array, an active edge is controlled by a metal wire

978-1-4799-6278-5/14/$31.00 ©2014 IEEE

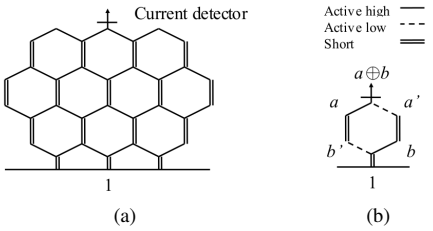

Fig. 1: (a) An SET array fabric. (b) An example of implementing $a \oplus b$ with an SET array [7].

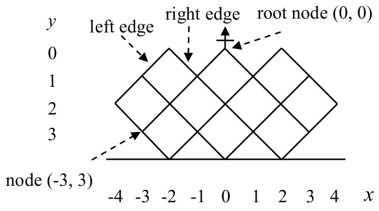

Fig. 2: An abstract diamond fabric [7].

connected to an input variable or its complement. A reconfigurable SET array requires the connections of the input variable (for each row) to be fixed at the fabrication time. Specifically, for each SET row, its corresponding variable controls all the left active edges and the complement variable controls all the right active edges, or vice versa. Therefore, the fabric constraint demands that all the left active edges of a row are of the same state (i.e., high or low), and all the right active edges are of the same state that is opposite to the state of the left active edges (i.e., low or high). Thus, for example, edge configurations (high,low) and (low,high) are prohibited to appear in the same row.

The other constraint is the granularity constraint which aims at reducing the metal wires required to configure the SET array. It requires the adjacent devices to be in the same operating state, i.e., active, short, or open. Under the constraint, only four edge configurations, (high,low), (low,high), (open,open) and (short,short), are allowed for an SET array.

B. Binary Decision Diagram

A binary decision diagram (BDD) is a directed acyclic graph that can represent an n-input Boolean function $f(x_1, \ldots, x_n)$ [10]. There are two types of nodes in a BDD, a terminal node and a non-terminal node. A terminal node v has a value 1 or 0. A non-terminal node v has a decision variable and two children, the *low child* ($lo(v)$) and the *high child* ($hi(v)$). A *high edge* (*low edge*) connects the high child (low child) to its parent. To distinguish different nodes with the same variable, we put a number after the variable. For example, in Fig. 3a, both of the nodes $b1$ and $b2$ are with the decision variable b. A BDD can be converted into a reduced-ordered binary decision diagram (ROBDD) by restricting variable ordering and applying reduction rules. For more details about BDD, readers are referred to [10].

A BDD shares two important similarities with a reconfigurable SET array. First, a high (low) edge of a BDD is functionally same as an active high (low) edge in an SET array. Second, for any input assignment, there is at most one conducting path from the root to the 1-terminal in BDD, which is similar to the SET array.

C. Related Works and Motivation

A previous work [6] attempted to manually map a BDD onto an SET array. However, they did not provide a systematic flow. Important

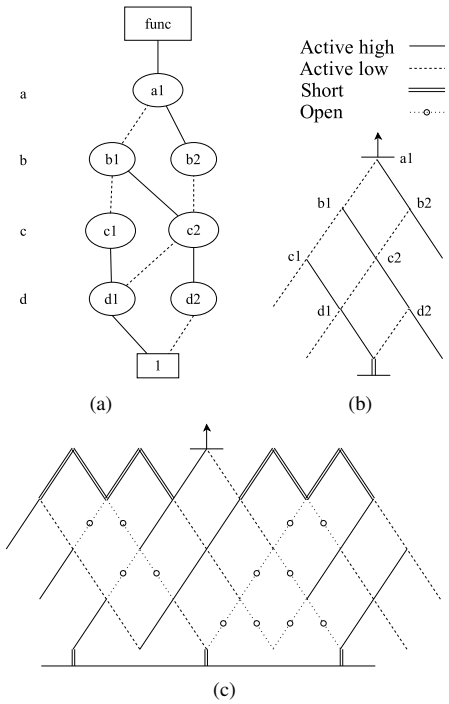

Fig. 3: Comparison of our BDD-based method and the previous product term-based method. (a) The input ROBDD to be mapped. We omit those edges connected to the 0-terminal of the ROBDD. (b) The mapping result using our method. (c) The mapping result using the product term-based method of [8].

issues including deciding coordinates of BDD nodes and resolving crossing edge in BDD were not mentioned in their work.

Besides manual design, there are a few works devoted to automated synthesis of an SET array [7] [8] [9]. These works are product term-based, in which a Boolean function is represented in a non-overlapping sum-of-product (SOP) form. The product terms are then mapped one by one onto an SET array. In these works, non-overlapping product terms are obtained by traversing the ROBDD of a function. Fig. 1b shows an example of mapping the function $a \oplus b$ by the previous product term-based methods. After obtaining the product terms 10 and 01, the left path is configured for 10, and then the right path is configured for 01. Among all the previous works, [8] provides the best result in terms of the area of the SET array.

However, SOP is a non-graphic data structure that loses structural information compared with graphic data structures like BDDs. Moreover, as mentioned above, BDDs and SET arrays also share some common properties that can be exploited, such as the limitation on the number of conducting paths. In this work, we try to directly map a BDD onto an SET array through proper BDD transformations.

Fig. 3 shows an example of mapping an ROBDD (in which the edges connected to the 0-terminal are omitted) to an SET array by our method (Fig. 3b) and by the product term-based method of [8] (Fig. 3c). Comparing the results, we observe that our BDD-based method could exploit the original graphic structure to significantly reduce the area of the mapping, while this feature is not utilized in the product term-based method.

III. Technology-Constraint-Free Mapping

In this section, we show our BDD-based algorithm to synthesize *technology-constraint-free* reconfigurable SET arrays. Although in

realistic situations, the synthesis of the SET arrays should consider the fabric and the granularity constraints, we discuss this algorithm first, since it serves as the basis for the algorithms that handle these technology constraints.

We assume that the logic function to be mapped is given to us in the form of an ROBDD. The idea of our method is to gradually transform the input ROBDD to a planar graph so that at the end of the transformation, it corresponds to the resultant SET array. Our method has three major steps.

The first step is to obtain a "full-level" BDD from an input ROBDD by adding intermediate nodes to the given ROBDD. The full-level BDD has the property that each path from the root to a terminal node visits all the input variables. The second and the third steps are executed over each level of the full-level BDD. The second step, pre-mapping, transforms a BDD into a planar form and assigns coordinates to the BDD nodes. However, two different BDD nodes may be assigned to the same location. The third step, conflict resolving, solves this problem and finally maps the planar structure onto an SET array. We will discuss the three steps in the following subsections.

Since there are no 0-terminals in an SET array, we do not need to map ROBDD paths that end at the 0-terminal. Thus, we can ignore the edges connected to the 0-terminal in an ROBDD. Fig. 4a shows an original ROBDD and Fig. 4b shows the ROBDD with the edges connected to the 0-terminal deleted. This makes some nodes have only one child. In the following discussion, we suppose that the input is an ROBDD in which only paths leading to the 1-terminal are preserved.

A. Adding Intermediate Nodes

The first step transforms a given ROBDD to a non-reduced OBDD so that every path from the root to the 1-terminal visits all the variables. We call such a BDD a "full-level" BDD. For example, for the BDD shown in Fig. 4b, the set of variables is a, b, c, d. However, there are some paths in the ROBDD that do not go through all the variables, e.g., the path $a1 \rightarrow d1 \rightarrow 1$, which does not go through the variables b and c. The first step transforms that BDD into a full-level BDD shown in Fig. 4c.

The transformation is achieved by adding the necessary intermediate nodes into the original ROBDD. In constructing the ROBDD, an important step is to remove redundant tests [10]. It means to remove each intermediate node n which satisfies $lo(n) = hi(n)$. The first step we apply here is essentially the reverse of removing redundant tests.

We do this by traversing the BDD from the root to the terminal. Each time when we encounter an edge that connects to two nodes whose variables are not adjacent in the order, we will add intermediate nodes. For example, for the edge $(a1, d1)$ shown in Fig. 4b, we add two nodes $b2$ and $c2$ as shown in Fig. 4c. The node $b2$ has both of its high edge and low edge connected to the node $c2$, while the node $c2$ has both of its high edge and low edge connected to the node $d1$. Note that the global Boolean functions of the nodes $b2$ and $c2$ are the same as that of the node $d1$. Thus, the function of the BDD does not change.

Note that for each added intermediate node, both of its high edge and low edge are connected to the same child. Thus, the Boolean function of an added node is equivalent to the Boolean function of its child. When mapping this pair of parent-child nodes onto the SET array, we can connect them by a single short edge.

When adding an intermediate node, we also check whether the node to be added is redundant. That is, whether there exists a node in the graph that has both the same variable and the same global function. For example, in Fig. 4b, after we have expanded the edge $(a1, d1)$ by adding the two intermediate nodes $b2$ and $c2$, we further process the edge $(b1, d1)$. Note that we need to add a node with variable c and with both of its high edge and low edge connected to the node $d1$. This node is redundant with regard to the node $c2$. Thus, we connect the low edge of $b1$ to the node $c2$, as shown in Fig. 4c.

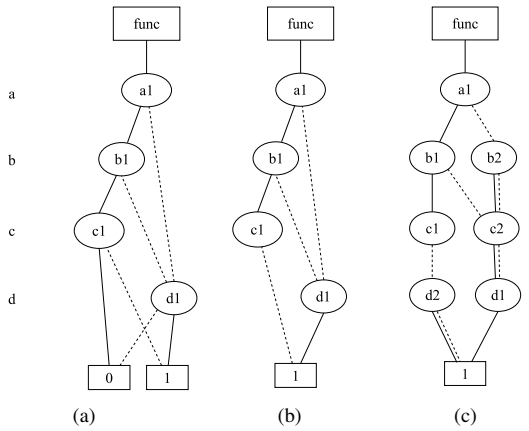

Fig. 4: Adding intermediate nodes. (a) The original ROBDD. (b) The BDD obtained from the BDD in (a) by removing all edges connected to the 0-terminal. (c) The full-level BDD.

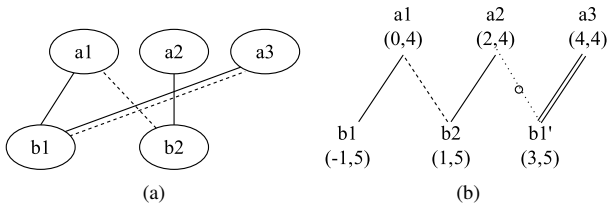

Fig. 5: A mapping example. (a) Two adjacent levels in the full-level BDD. (b) The mapping result of (a).

B. Pre-mapping

The main purpose of pre-mapping is to transform a BDD into a planar form. As shown in procedure *MAP()* in Algorithm 1, pre-mapping (procedure *PREMAP()*) is performed for each level of the full-level BDD from the top level to the bottom level.

Assuming that a parent level of BDD nodes is mapped to the SET array, pre-mapping maps its child level of BDD nodes based on the mapping of the parent level. It starts from the leftmost node of the parent level and visits each node of the parent level from left to right in sequence. When visiting a parent node, it will map the child(ren) of that node to the SET array by assigning coordinates to the child(ren). Pre-mapping ensures that each pair of a parent node and a child node in BDD is mapped *adjacently* in the SET array. Here, two nodes are said to be adjacent in an SET array if they are mapped to the two ends of an SET edge.

However, sometimes a child of the parent node under consideration has already been mapped to a location that is not adjacent to the parent node. This happens when the child is also a child of another parent node that is previously visited. In order to maintain the correct function, we need to duplicate the BDD node.

Fig. 5 shows an example. We map the level of the nodes with the variable b based on the mapping of the level of the nodes with the

variable a. Suppose that the nodes $b1$ and $b2$ have just been mapped to the coordinates shown in Fig. 5b. When we continue to process the node $a3$ and its child $b1$, we find that the node $b1$ has been mapped, but it is not adjacent to its parent $a3$. In this case, we make a copy of the BDD node $b1$ as a new BDD node $b1'$ and connect the node $a3$ to this copy. Consequently, the node $b1'$ can be mapped adjacent to the node $a3$ in the SET array by a short edge.

When duplicating a BDD node, the connection of this BDD node to the nodes on the next level are also duplicated. In essence, this step is the reverse of removing duplicate nonterminals used in constructing an ROBDD [10]. This may create new parent-child pairs, with parents in the current level and children in the next level. Some of these new parent-child pairs may be non-adjacent, but they will be resolved in the same way when doing pre-mapping on the next level.

As node duplication does not change the original function or introduce multiple conducting paths, the operation results in a valid SET mapping.

With the help of node duplication, each child of a parent node can now be mapped adjacent to the parent node. When mapping a child node to the SET array, we exploit the sharing of children nodes in BDD. If we find that the current parent node and its right neighbor at the same level are adjacent (e.g., the nodes $a1$ and $a2$ in Fig. 5b) and they share a common child, then we map this common child between these two nodes on the next level. For example, in Fig. 5, the node $b2$ is a common child of the adjacent parent nodes $a1$ and $a2$. Thus, it is mapped to the location shown in Fig. 5b. Once the common child is mapped, we will map the other child of the parent to its left. If there is no common child, we map the child adjacent to the parent using the following rule: if the lower left side of the parent is not occupied by any node, we put the child there; otherwise we put it to the lower right side of the parent.

C. Conflict Resolving

As shown in procedure *MAP()* in Algorithm 1, conflict resolving (procedure *CONFLICT_RESOLVE()*) is performed for each level of the full-level BDD from the top level to the bottom level, following pre-mapping of a level.

Even though pre-mapping has obtained a planar structure, it may not be directly mappable to an SET array as two nodes of different functions may be mapped to the same location. When it happens we say there is a *coordinate conflict*. The two nodes involved in the conflict are called *conflicting nodes*. For example, in Fig. 6a, $c1$ and $c2$ are the children of $b1$, and $c3$ and $c4$ the children of $b2$. Pre-mapping has mapped $c2$ and $c3$ to the adjacent locations of their parents $b1$ and $b2$, respectively. However, the result is that $c2$ and $c3$ are mapped at the same location, which is not a legal mapping. Conflict resolving addresses this problem and returns a final mapping.

Two techniques, row expansion and side expansion, could be used to solve coordinate conflicts. The idea of both techniques is to create extra space to separate conflicting nodes from each other.

Row expansion solves coordinate conflicts by inserting dummy levels between the current level which has a coordinate conflict and the parent level. Dummy levels are composed of two kinds of edges: short edges for extending and separating the parents of the conflicting nodes and open edges for avoiding invalid paths. We call the short edges in the expanded rows *extended edges*.

As Fig. 6b shows, to solve the conflict of the nodes $c2$ and $c3$, we use two short edges to expand the parent nodes $b1$ and $b2$ to their new locations $b1'$ and $b2'$, which leaves enough space to put their respective child $c2$ and $c3$ without conflict. If there are other nodes in the parent level, they are also expanded by parallel short edges. As Fig. 6c shows, if the nodes $a2$ and $a3$ are expanded to $a2'$ and $a3'$ to solve a coordinate conflict, then the rest of the nodes $a1$, $a4$ and $a5$ should also be expanded.

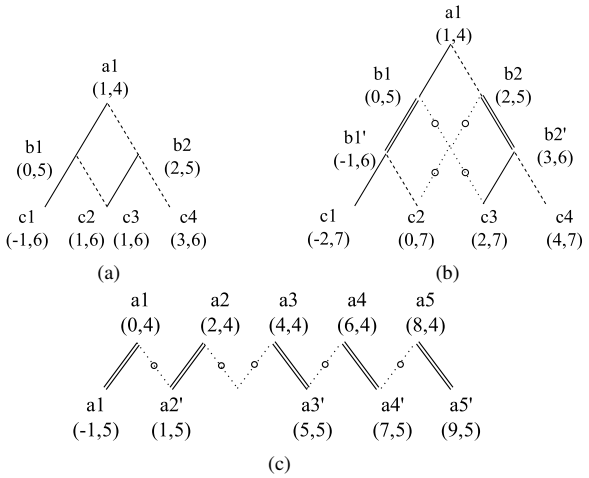

Fig. 6: (a) An example of coordinate conflict. (b) Row expansion to solve coordinate conflict. (c) An illustration that row expansion should expand an entire row.

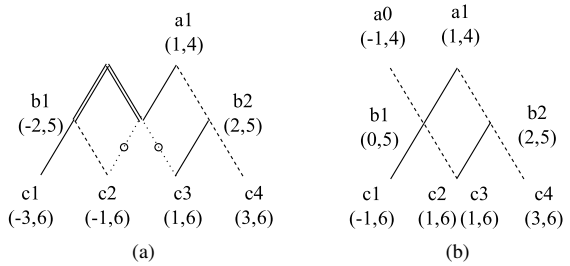

Fig. 7: (a) An example of side expansion. (b) A case that prohibits the side expansion.

No controlling variable is required for a dummy level. Yet, for technology consistency, we can simply assign any variable to the dummy level.

However, row expansion is expensive in area. Thus, we propose another method, side expansion, to resolve conflict. Side expansion creates additional space for mapping by expanding only one parent of the conflicting nodes. It can be applied if both of the following conditions are met: 1) the parent of one conflicting node appears in the leftmost or the rightmost, i.e., the "side", of their level, and 2) there are un-configured edges to allow side expansion.

In the previous example shown in Fig. 6a, if the node $b1$ happens to be at the leftmost of its level, we may expand it by two *short* edges on the same level. Fig. 7a shows the result of this expansion. Similar with the case in row expansion, the short edges in the side expansion are also called *extended edges*. However, it is possible that there are no additional edges for configuration to expand a side node. As Fig. 7b shows, if $b1$ has another parent $a0$, then the side expansion of $b1$ is impossible.

Generally, side expansion is more area-efficient in resolving a conflict than row expansion. Thus, in our algorithm we first check if it is applicable. If not, row expansion will be used.

The synthesis of an SET array requires the configuration information of the edges. The edge configuration can be easily obtained

according to how the two nodes at the end of a SET edge are connected in the BDD. Specifically, we have the following four rules.

1) If the two nodes are connected by a high edge in BDD, we configure the SET edge as active high.
2) If the two nodes are connected by a low edge in BDD, we configure the SET edge as active low.
3) If the two nodes are connected by both a high edge and a low edge in BDD, we configure the SET edge as short.
4) If the two nodes are not connected in BDD, we configure the SET edge as open.

For example, in Fig. 5b, the edge between the nodes $a1$ and $b1$ is configured as active high, while the edge between the nodes $a3$ and $b1'$ is configured as short.

The whole flow of mapping a full-level BDD to a technology-constraint-free SET array is shown in Algorithm 1. Procedure *MAP()* manipulates one level at a time using pre-mapping (procedure *PREMAP()*) and conflict resolving (procedure *CONFLICT_RESOLVE()*) described above.

In the pseudo-code, $nVar$ is the number of controlling variables, which equals the number of levels (excluding the terminal level) of the given BDD. n_j is the j-th node on a certain level. After initializing the coordinate for the root node, the mapping procedure is run $nVar$ times until all the successive levels, including the terminal level, are mapped.

Some of the transformations discussed above, such as adding intermediate nodes and duplicating nodes, can be viewed as the reverse of the common BDD reduction operations. Although they could increase the size of a BDD, they are necessary in our procedure, since our goal is to map the function represented by a BDD to the planar SET array. Further, the sharing strategy we apply ensures that we add nodes only when necessary. As the experimental results in Sec. V demonstrated, the BDD size increase due to these transformations is moderate for most benchmarks.

Algorithm 1 Technology-constraint-free Mapping.

```
1: procedure MAP(a full-level BDD)
2:     initiate the root coordinate;
3:     for i = 1 to nVar do
4:         PREMAP(i);
5:         CONFLICT_RESOLVE(i);
6:     end for
7: end procedure
8: procedure PREMAP(level i)
9:     for each node n_j in level i − 1 do
10:        if child(ren) of n_j is mapped away from n_j then
11:            duplicate the child(ren);
12:        end if
13:        if n_j and n_{j+1} are adjacent and have a shared child then
14:            map the shared child in middle;
15:            map its sibling of parent n_j;
16:            continue;
17:        end if
18:        map children of n_j to its adjacent location;
19:    end for
20: end procedure
21: procedure CONFLICT_RESOLVE(level i)
22:    for each node of level i do
23:        if there exists a coordinate conflict then
24:            if side expansion is possible then
25:                do side expansion;
26:            else
27:                do row expansion;
28:            end if
29:        end if
30:    end for
31: end procedure
```

IV. Mapping under the Technology Constraints

In this section, we show our methods to handle two technology constraints, the fabric constraint and the granularity constraint discussed in Section II. To handle the fabric constraint, we only need to make small changes to the constraint-free mapping algorithm shown in Section III. Besides the fabric constraint, we usually also need to consider the granularity constraint. Thus, we further propose a procedure to transform a mapping that satisfies the fabric constraint to one that satisfies both constraints.

A. Mapping under the Fabric Constraint

The fabric constraint requires all the left active edges of a row are active high, and the right active edges are active low, or vice versa. To satisfy this constraint, three changes are made in the flow of the original constraint-free mapping.

First, the fabric type (high,low) or (low,high) is decided before mapping a level. The type can be set for the whole SET array, or tailored for each row. We use the second method. The type is determined for each row according to the first node sharing in the row that implies one of the fabric types. For example, if the shared child is connected to its left parent by a low edge and to its right parent by a high edge, the fabric type is set to (high,low). If the type cannot be determined in this way, the default is (high,low).

Second, once the type of the row is set, mapping of nodes must follow the type. For example, given the type (high,low), we cannot map a low child to the left side of the parent.

Third, node sharing is assessed under the fabric constraint. For example, given the type (high,low), the high child of a parent node n and the low child of the right neighbor of n cannot be shared even if they are identical.

B. Mapping under the Granularity Constraint

The granularity constraint permits only four edge configurations over the SET array: (high,low), (low,high), (short,short) and (open,open). When it is imposed, a procedure called *network expansion* is executed to fix the prior mapping result, in which only the fabric constraint is satisfied. The resultant SET array is called an *expanded network* in which both constraints are satisfied.

The prior result violates the granularity constraint as configurations like (high,open), (open,short) are allowed in it. For example, in Fig. 5b, the two edges rooted at the node $a2$ has an illegal configuration (high,open). Moreover, row expansion using (open,short) and (short,open) edges is also prohibited with the granularity constraint. Clearly, we cannot impose the configurations directly on the mapping, such as changing (high,open) to (high,low), as it would change the Boolean function.

Our strategy to handle the granularity constraint is to expand the prior network with its original node connections, while creating enough space between nodes, which allows us to impose granularity configurations. For example, Fig. 8a shows a diamond in the prior mapping, and Fig. 8b expands the diamond into four. In the expanded network, parents and children are separated by two rows rather than one, and a functional edge in the prior mapping is replaced by two edges of the same type. A *functional edge* is defined to be either an active edge or a short but non-extended edge in the prior mapping. For example, the edge $(a1, b1)$ in Fig. 8a is a functional edge and it is replaced by two edges $e1$ and $e3$ in Fig. 8b. In the expanded network, one PI controls two adjacent rows. For example, PI a controls the

first two rows. Further, open edges are added to avoid false paths (e.g., path $e4 \rightarrow e11$).

Our algorithm to perform network expansion is shown in Algorithm 2. It takes a prior mapping which satisfies the fabric constraint as the input. It includes two major steps: 1) for each node in the prior mapping, deciding its new coordinate in the expanded network (Lines 2–8 in Algorithm 2), and 2) configuring the edges (Line 9 in Algorithm 2).

1) Deciding Coordinates: To decide the coordinates of the nodes in the expanded network, we expand each level of the nodes in the prior mapping (procedure *EXPAND_LEVEL()* in Algorithm 2) in turn from the bottom level to the top level of the SET array. The parent level is expanded based on its child level.

In addition to the notations in Algorithm 1, we define $Coord(n)$ and $Coord'(n)$ as the coordinates of a node n in the prior mapping and in the expanded network, respectively. Moreover, we denote p_k as the k-th direct parent of a certain node. A *direct parent* of a node is the parent connected to the node by one functional edge and possibly one or more extended edges. For example, in Fig. 6b, the direct parents of the nodes $c1$ and $b1'$ are the nodes $b1'$ and $a1$, respectively. In Fig. 7a, the direct parent of the node $b1$ is the node $a1$. The number of direct parents of a node in an SET array is at most two.

In handling the granularity constraint, we only process direct parents of the current node. Although some nodes may connect to the current node through the extended edges in the prior mapping, they are ignored. As a result, although the prior mapping may have multiple expanded rows, all the nodes on these expanded rows are ignored. Thus, the height of the SET array is not affected by the row expansion in the prior mapping.

Algorithm 2 Network Expansion.

```
1: procedure NETWORK_EXPANSION(prior mapping)
2:     for each node n_i in the terminal level do
3:         Coord'(n_i).x ← 2Coord(n_i).x;
4:         Coord'(n_i).y ← Coord(n_i).y;
5:     end for
6:     for i = nVar − 1 to 0 do
7:         EXPAND_LEVEL(i);
8:     end for
9:     EDGE_CONFIGURE();
10: end procedure
11: procedure EXPAND_LEVEL(level i)
12:     for each node n_j in level i + 1 do
13:         for each direct parent p_k of node n_j do
14:             if p_k has been expanded then
15:                 continue;
16:             end if
17:             if p_k is previously mapped to the right of n_j then
18:                 Coord'(p_k).x ← Coord'(n_j).x + 2;
19:             else
20:                 Coord'(p_k).x ← Coord'(n_j).x − 2;
21:             end if
22:             Coord'(p_k).y ← Coord'(n_j).y − 2;
23:         end for
24:     end for
25: end procedure
```

In the main procedure *NETWORK_EXPANSION()*, we first double the x coordinates of the nodes in the terminal level. With this, the distance between the adjacent terminal nodes becomes twice the original distance. The y coordinates remain the same. Then, we enter the procedure *EXPAND_LEVEL()*. For each node n_j of the last expanded level, we expand its direct parent p_k by the following rules.

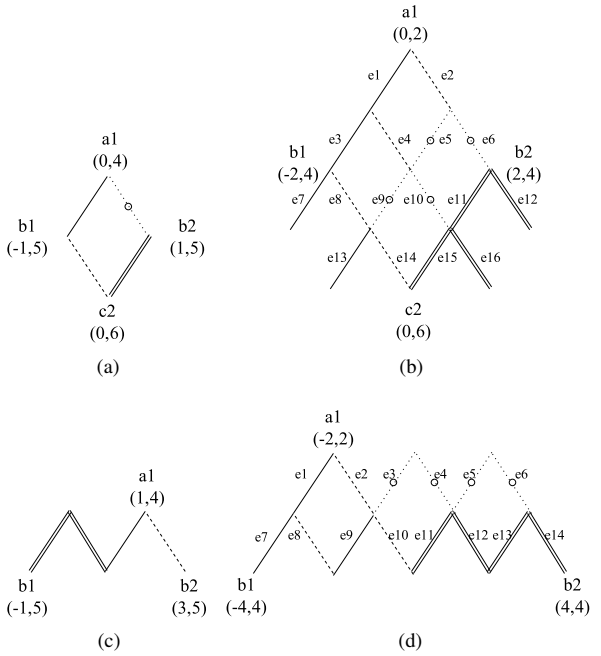

Fig. 8: Examples of network expansion. (a) A prior mapping. (b) Expansion of the mapping in (a). (c) Another prior mapping. (d) Expansion of the mapping in (c).

We first check if p_k has been already expanded. This happens when the left neighbor of n_j, n_{j-1}, also has p_k as the direct parent. In this case, p_k has already been expanded when we process the node n_{j-1}. If so, we move to the next direct parent of the node n_j (if any). On the other hand, if p_k has not been expanded, then we will expand p_k based on the relative location of p_k and n_j in the prior mapping: if p_k is mapped to the right of the node n_j, we set its new x coordinate $Coord'(p_k).x$ to $Coord'(n_j).x+2$; otherwise, it is set to $Coord'(n_j).x - 2$. The y coordinate is set to $Coord'(n_j).y - 2$, which is two levels above the child node n_j.

For example, Fig. 8a is a prior mapping and Fig. 8b is the expanded network of it. Suppose that we have already processed the level with the variable b. As a result, the nodes $b1$ and $b2$ have been mapped to the coordinates $(-2, 4)$ and $(2, 4)$, respectively, as shown in Fig. 8b. Now we expand their parent level. We do this by taking the node $b1$ and checking its parent $a1$. We notice that $a1$ is originally mapped to the right of the node $b1$. Applying the above rules, the new x coordinate of $a1$ should be set to $Coord'(b1).x + 2 = 0$. Its y coordinate is set to $Coord'(b1).y - 2 = 2$. Similarly, Fig. 8d shows the expanded network of Fig. 8c. In Fig. 8d, suppose that the nodes $b1$ and $b2$ have been mapped to the coordinates $(-4, 4)$ and $(4, 4)$, respectively. The node $a1$ is then placed at a location with x coordinate being $Coord'(b1).x + 2 = -2$ and y coordinate being $Coord'(b1).y - 2 = 2$.

2) Configuring Edges: As the previous step shows, in the expanded network, parents and children are separated by two rows rather than one. Edge configuration is no longer trivial and it is a critical step to maintain both the functional correctness and granularity constraint. Given an expanded network with only node coordinates, the edges are determined by applying the following steps.

First, for every direct parent-children pair in the prior mapping connected with a *functional edge* e, if their x coordinates differ by 2, we use two edges of the same type of e to connect them in the expanded network. For example, the active high edge $(a1, b1)$ in

TABLE I: Comparison of our method with the state-of-the-art method [8].

Benchmark	#PI	#PO	#hex		ratio	time(sec)		ratio	#BDD node		ratio
			by [8]	ours		by [8]	ours		original	after [1]	
C17	5	2	54	33	0.61	0.01	0.00	0.00	10	17	1.70
cm138a	6	8	460	88	0.19	0.01	0.00	0.00	56	104	1.86
x2	10	7	397	393	0.99	0.01	0.00	0.00	53	133	2.51
cm85a	11	3	686	234	0.34	0.01	0.00	0.00	42	83	1.98
cm151a	12	2	521	577	1.11	0.01	0.00	0.00	34	148	4.35
cm162a	14	5	578	458	0.79	0.02	0.00	0.00	50	138	2.76
cu	14	11	415	618	1.49	0.01	0.00	0.00	87	144	1.66
cmb	16	4	376	86	0.23	0.01	0.00	0.00	52	75	1.44
cm163a	16	5	391	341	0.87	0.01	0.00	0.00	45	100	2.22
pm1	16	13	586	278	0.47	0.02	0.00	0.20	78	117	1.50
pcle	19	9	751	599	0.80	0.01	0.00	0.40	95	184	1.94
sct	19	15	3168	842	0.27	0.02	0.00	0.20	124	240	1.94
cc	21	20	1040	622	0.60	0.01	0.00	0.40	96	140	1.46
i1	25	16	1190	538	0.45	0.01	0.00	0.04	77	120	1.56
lal	26	19	3312	1219	0.37	0.05	0.00	0.08	164	328	2.00
pcler8	27	17	1920	1411	0.73	0.04	0.02	0.50	164	352	2.15
frg1	28	3	13962	9756	0.70	0.14	0.00	0.02	96	934	9.73
c8	28	18	2026	1286	0.63	0.03	0.02	0.67	138	311	2.25
term1	34	10	35975	2895	0.08	0.51	0.02	0.04	173	547	3.16
count	35	16	4590	2776	0.60	0.07	0.00	0.06	232	736	3.17
unreg	36	16	1515	1132	0.75	0.03	0.02	0.67	112	245	2.19
b9	41	21	9112	2616	0.29	0.10	0.04	0.40	192	539	2.81
cht	47	36	3556	2148	0.60	0.04	0.04	1.00	191	310	1.62
apex7	49	37	49004	13737	0.28	0.41	0.06	0.15	510	2554	5.01
example2	85	66	14402	9091	0.63	0.61	0.14	0.23	720	1348	1.87
steppermotordrive	29	29	22994	6160	0.27	0.31	0.06	0.19	575	1313	2.28
usb_phy	113	116	28960	15399	0.53	0.58	0.42	0.72	674	1145	1.70
sasc	133	129	54987	22492	0.41	3.63	0.70	0.19	994	2104	2.12
i2c	147	142	115944	35209	0.30	11.99	1.10	0.09	1983	4454	2.25
simple_spi	148	144	129039	33472	0.26	12.08	1.04	0.09	1643	4161	2.53
i8	133	81	111992	136778	1.22	10.09	0.80	0.08	2537	20141	7.94
geomean			3314.17	1615.06	0.49	0.08	0.00	0.06	159.72	378.61	2.37

[1]After transformations including adding intermediate nodes and node duplication.

Fig. 8a is mapped to two active high edges $e1$ and $e3$ in Fig. 8b. Otherwise, if the x coordinate difference of a pair of a direct parent and its child is larger than 2, we configure short edges in addition to two edges of the same type of e. For example, in Fig. 8c, the nodes $a1$ and $b2$ are originally connected with an active low edge. In the expanded network shown in Fig. 8d, we configure two active low edges $e2$ and $e10$ followed by four short edges $e11, e12 \ldots, e14$ to connect the nodes $a1$ and $b2$ together.

The second step configures edges that are enforced by three of the granularity configurations (high,low), (low,high) and (short,short). For example, in Fig. 8b, by the granularity constraint, the edges $e2$ and $e4$ are configured as active low since the edges $e1$ and $e3$ have been configured as active high in the previous step. Likewise, the edges $e7$ and $e13$ are configured as active high; the edges $e12$ and $e16$ are configured as short.

The third step configures the remaining edges as open. Open edges prevent invalid paths in the expanded network. For example, in Fig. 8b, the edge $e6$ is set as open. Therefore, although the granularity constraint enforces $e2$ to be an active low edge, invalid path $e2 \rightarrow e6$ is prevented. Since the configuration in the second step does not match an edge that is not open with an open edge, thus, each open edge is paired with another open edge. In other words, only (open,open) configuration is possible in this step.

By the above construction, we make sure that the granularity constraint is satisfied. Also, the Boolean function is preserved by two facts:

1) All the functional connections in the prior mapping are correctly preserved in the extended network, with edges controlled by the same variable.
2) Only these functional connections in the prior mapping are possible in the extended network, because the existing of the open edges prevents any illegal connections.

V. Experimental Results

We implemented our method in C using CUDD package [11]. The experiments were conducted on a 2.40GHz Linux platform. A set of MCNC and IWLS 2005 benchmarks [12] were used. Same as the previous methods in [7] and [8], we apply the BDD reordering heuristic CUDD_REORDER_SYMM_SIFT [11] after reading a benchmark; also same as the previous methods, for a multiple-output function, we mapped the function of each primary output (PO) separately onto an SET array. The area of a benchmark is measured by the total numbers of hexagons in the SET arrays for all the POs. In the experiment, we assumed that both the granularity constraint and the fabric constraint are applied.

Table I shows the experimental results. We show the number of hexagons, total runtime, and the respective ratios of our proposed algorithm and the algorithm proposed in [8] in Columns 4 to 9. To study how our technique affects the BDD complexity, we also show the number of nodes in the original BDD and that after adding intermediate nodes and duplicating nodes in Columns 10 and 11, respectively. Their ratios are given in the last Column. Note that here

we do not consider the effect of network expansion operation on the BDD size, since its effect on complexity is just as multiplying a constant. The number of nodes for each benchmark is measured by adding together the nodes for each of its POs. The geometric means of the numbers of hexagons, runtime, the number of nodes, and the respective ratios are shown in the last row in the table.

We can see that compared to the method in [8], our method saves 51% area on average in terms of the number of hexagons. The average runtime of our algorithm is less than 0.005 second, and the longest runtime of all benchmarks is less than 1.1 seconds, which is a large improvement over the previous method. The main reason for the runtime improvement is that the previous method spends a large amount of time in handling false paths, while our method avoids them by construction.

The average percentage of BDD node increase due to adding intermediate nodes and node duplication is 137%. This shows that BDD size increase is not large for the benchmarks. However, the listed benchmarks are small compared to practical circuits. For example, even though `i2c` has 147 PIs and 142 POs, most PO functions use only a small fraction of the PIs and therefore its BDD size increase is small. For benchmarks containing large functions (e.g., `frg1` and `i8`), the size increase is larger. In our future work, we will further study the effect of the size increase caused by BDD transformations on larger benchmarks.

VI. Conclusion

In this work, we propose a new approach to synthesize logic function with reconfigurable SET arrays. Our approach is BDD-based which takes the advantage of the structural similarity between a BDD and an SET array. Our method transforms an ROBDD into a planar graph which enables efficient mapping onto an SET array. Compared to the state-of-the-art method proposed in [8], the SET array synthesized by our approach has a much smaller area.

Acknowledgement

This work is supported by a grant from the National Natural Science Foundation of China (NSFC), Project No. 61204042.

References

[1] H. W. C. Postma, T. Teepen, Z. Yao, M. Grifoni, and C. Dekker, "Carbon nanotube single-electron transistors at room temperature," *Science*, vol. 293, no. 5527, pp. 76–79, 2001.

[2] L. Zhuang, L. Guo, and S. Y. Chou, "Silicon single-electron quantum-dot transistor switch operating at room temperature," *Applied Physics Letters*, vol. 72, no. 10, pp. 1205–1207, 1998.

[3] 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 *International Semiconductor Device Research Symposium*, 2001, pp. 622–625.

[4] N. Asahi, M. Akazawa, and Y. Amemiya, "Single-electron logic device based on the binary decision diagram," *IEEE Transactions on Electron Devices*, vol. 44, no. 7, pp. 1109–1116, 1997.

[5] H. Hasegawa and S. Kasai, "Hexagonal binary decision diagram 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, no. 2, pp. 149–154, 2001.

[6] S. Eachempati, V. Saripalli, N. Vijaykrishnan, and S. Datta, "Reconfigurable BDD based quantum circuits," in *IEEE International Symposium on Nanoscale Architectures*, 2008, pp. 61–67.

[7] Y.-C. Chen, S. Eachempati, C.-Y. Wang, S. Datta, Y. Xie, and V. Narayanan, "Automated mapping for reconfigurable single-electron transistor arrays," in *Design Automation Conference*, 2011, pp. 878–883.

[8] C.-E. Chiang, L.-F. Tang, C.-Y. Wang, C.-Y. Huang, Y.-C. Chen, S. Datta, and V. Narayanan, "On reconfigurable single-electron transistor arrays synthesis using reordering techniques," in *Design, Automation and Test in Europe*, 2013, pp. 1807–1812.

[9] Y.-C. Chen, S. Eachempati, C.-Y. Wang, S. Datta, Y. Xie, and V. Narayanan, "A synthesis algorithm for reconfigurable single-electron transistor arrays," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 9, no. 1, p. 5, 2013.

[10] R. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[11] F. Somenzi, "CUDD: CU decision diagram package release 2.5.0," *University of Colorado at Boulder*, 1998.

[12] "IWLS 2005 benchmarks," http://iwls.org/iwls2005/benchmarks.html, accessed: 2014-02-30.