A Synthesis Algorithm for Reconfigurable Single-Electron Transistor Arrays

YUNG-CHIH CHEN, Chung Yuan Christian University
SOUMYA EACHEMPATI, Intel Corporation
CHUN-YAO WANG, National Tsing Hua University
SUMAN DATTA, YUAN XIE, and VIJAYKRISHNAN NARAYANAN, Pennsylvania State University

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-based reconfigurable logic architecture using SETs, it lacks an automatic synthesis algorithm for the architecture. 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.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Automatic synthesis*

General Terms: Algorithms

Additional Key Words and Phrases: Automatic synthesis, binary decision diagram, single-electron transistor

ACM Reference Format:
Chen, Y.-C., Eachempati, S., Wang, C.-Y., Datta, S., Xie, Y., and Narayanan, V. 2013. A synthesis algorithm for reconfigurable single-electron transistor arrays. *ACM J. Emerg. Technol. Comput. Syst.* 9, 1, Article 5 (February 2013), 20 pages.
DOI: http://dx.doi.org/10.1145/2422094.2422099

1. 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 [Keckler et al. 2009]. 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 [Keating et al. 2007; Piguet 2006].

Y.-C. Chen is currently affiliated with Yuan Ze University.
This is an extended version of a previously published conference research paper [Chen et al. 2011].
This work was supported in part by the National Science Council of Taiwan under grants NSC 99-2918-I-007-007, 99-2628-E-007-096, 99-2220-E-007-003, 100-2628-E-007-008, 100-2628-E-007-031-MY3, and 100-2218-E-033-008, and by the National Science Foundation of USA under grants 0829926, 0903432, 0916887, and 0643902.
Author's address: Y.-C. Chen; email: ycchen.phi@gmail.com.

DOI: http://dx.doi.org/10.1145/2422094.2422099

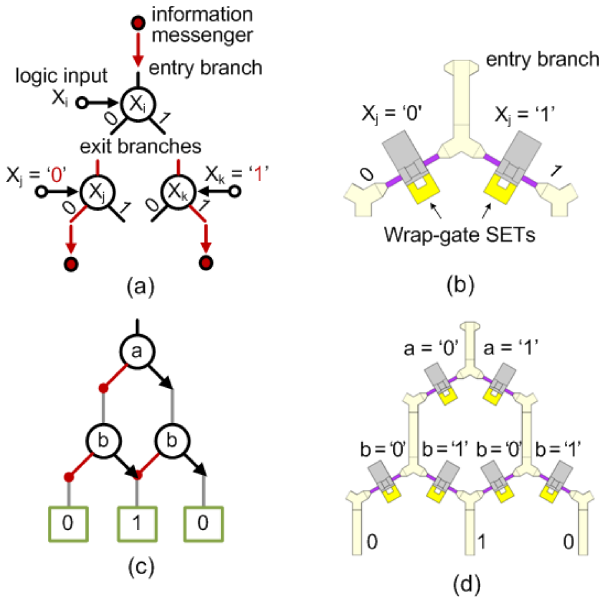

Fig. 1. (a) Node devices. (b) Node devices realized by controlling nanowires with wrap-gate SETs. (c) A BDD-logic representation of a 2-bit XOR. (d) The implementation of the 2-bit XOR in (b) using nanowires controlled by wrap-gate SET devices.

On the device level, as the power-delay product reaches quantum limits, a plethora of new device concepts are being explored to exploit 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 coexplore an emerging device design in conjunction with a non-CMOS logic architecture. To this end, a novel binary decision diagram (BDD)-based [Bryant 1986] logic architecture was proposed as a suitable candidate for implementing logic using ultra-low power nanodevices [Kasai et al. 2001]. Then, the BDD of a combinational circuit is mapped onto a hexagonal nanowire fabric controlled by Schottky wrap gates [Hasegawa and Kasai 2001].

To implement a BDD, each BDD node corresponds to a node device in the hexagonal fabric. As shown in Figure 1(a), a node device works like a switch that receives the messenger electrons from a preceding device through the entry branch and sends the electrons to a following device through either the left (0) or the right (1) exit branches according to the control variable (x_i). The node device can be realized by controlling nanowires with wrap-gate SETs as shown in Figure 1(b) [Liu et al. 2011]. Each exit branch (left or right) corresponds to a nanowire and its conductivity is controlled by a wrap-gate SET that has two operating modes: active high and active low. Furthermore, all the node devices at the same row in the hexagonal fabric are controlled by a single variable, that is, a primary input. The value of the given function is determined by observing which terminal the messenger electrons from the root node reach. For example, Figure 1(c) shows a BDD-logic representation of a 2-bit XOR. The electrons reach the 1 terminal when $a \neq b$. Figure 1(d) shows the implementation of this 2-bit XOR BDD-logic using nanowires controlled by wrap-gate SET devices.

From the viewpoint of current flowing, the behavior of the hexagonal fabric can be considered as that there is a current detector at the root that measures the current (if any) coming from the current source at the 1 terminal. The value of the function is determined by checking whether the current can be detected at the top by passing through a conducting path established by the input variables.

However, the realization of the BDD architecture in Kasai et al. [2001] is fixed and not amenable to functional reconfiguration. This is because the approach selectively etches all paths that do not lead to the 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 [Eachempati et al. 2008] and the in-depth simulation to study the electrostatic properties was presented [Saripalli et al. 2010]. This SET 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 simulation shows that this SET can provide an order of magnitude lower energy-delay than CMOS device [Saripalli et al. 2010].

However, the synthesis of a BDD using the SET array in [Eachempati et al. 2008] is manual rather than automatic. 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 automatic 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. Because both the mapping order of the product terms and the variable order in the product terms affect the mapping results, we propose four product term-sorting heuristics [Chen et al. 2011] and one variable-reordering heuristic to reduce area cost. Additionally, the automatic 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 [Eachempati et al. 2008].

We conduct experiments on a set of MCNC benchmarks [Yang 1991]. The experimental results show that the proposed approach can complete mapping within 1 second for most of the benchmarks.

Recent experimental work [Liu et al. 2011] has shown a wrap-gate SET device which is capable of operating in three distinct modes: a) active, b) open, and c) short state. The schematic of this device is shown in Figure 2(a) and the SEM image of the same is shown in Figure 2(b). Low temperature experimental characteristics of this device, shown in Figure 2(c), shows Coulomb oscillations, thus proving SET mode operation. Measured characteristics in Figure 2(d) shows that this device can operate in three different modes, thus showing that this experimental device is a practical realization of the reconfigurable SET device proposed in Eachempati et al. [2008]. Furthermore, the development of SETs at room temperature has been significantly improved. A Si-based SET with a 2nm nanodot, developed using pattern-dependent oxidation was demonstrated [Shin et al. 2010]. The electron energy-level separation of this device has been estimated to be 0.87eV ($\sim$35kT at 300K), thus making it suitable for room temperature operation. Thus, an automatic synthesis algorithm, which is the main

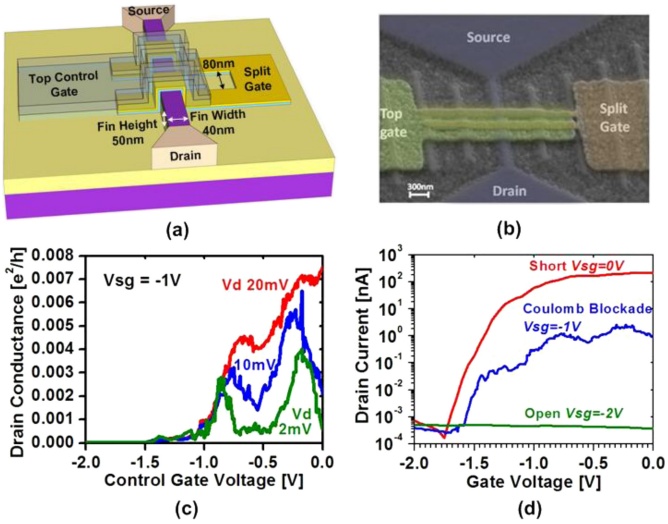

Fig. 2. (a) Schematic of wrap-gate SET device. (b) SEM image of experimental wrap-gate SET device. (c) Measured low temperature (4.2K) Coulomb oscillations of experimental device. (d) Experimental demonstration of reconfigurable operation in three separate modes.

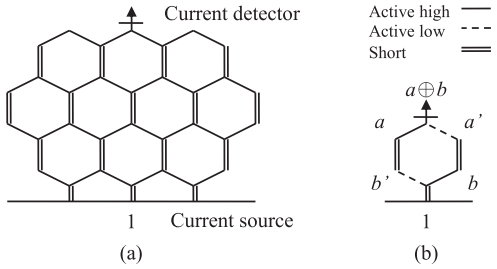

Fig. 3. (a) A SET array fabric. (b) An example of a XOR b.

contribution of this work, allows taking advantage of these novel energy-efficient devices to allow efficient realization of low-power logic circuits.

The rest of this article is organized as follows: Section 2 uses an example to demonstrate the problem considered in this article, and introduces some notations. Section 3 presents the proposed approach for mapping product terms into a SET array. Sections 4 and 5 introduce four product term-sorting heuristics and one variable-reordering heuristic, respectively. Section 6 presents the overall mapping flow. Section 7 discusses and addresses two mapping constraints. Finally, the experimental results and conclusion are presented in Sections 8 and 9.

2. BACKGROUND

2.1. An Example

A SET array can be presented as a graph composed of hexagons. As shown in Figure 3(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 individually. An active high edge is controlled by a variable x. It is conducting and non-conducting when $x = 1$ and $x = 0$,

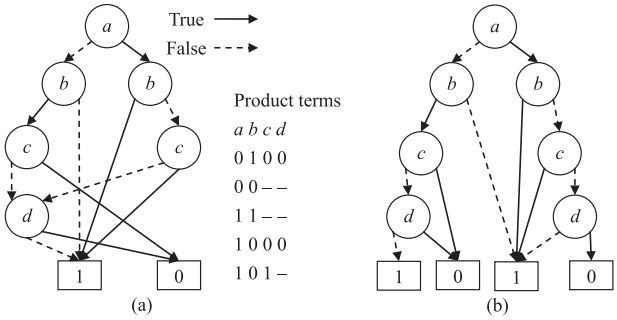

Fig. 4. An example of eliminating the crossing edges in an ROBDD by node duplication. (a) The original ROBDD. (b) The resultant BDD.

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, that is, a primary input (PI). The PIs 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, Figure 3(b) shows a SET array implementing a 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, that is, the number of configured hexagons.

Previous work [Eachempati et al. 2008] 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, an ROBDD usually has some crossing edges. Since a SET array is a planar architecture, many efforts are 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 Figure 4(a) by duplicating the node d and the terminals. The resultant BDD having more nodes is shown in Figure 4(b). Additionally, 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 1 terminal in the ROBDD, that is, product terms. Then, it maps each product term into a path in the SET array. The proposed method simultaneously avoids the crossing edge and the BDD node mapping issues.

For example, the product terms of a 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 Figure 3(b), where the left path is configured for 10 and the right path is for 01.

2.2. Notations

For ease of discussion, we use an abstract graph to present a SET array. Compared to Figure 3(a), only the configurable edges (i.e., sloping edges) are preserved as shown in Figure 5. In this diamond fabric, each node n, that is, 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*

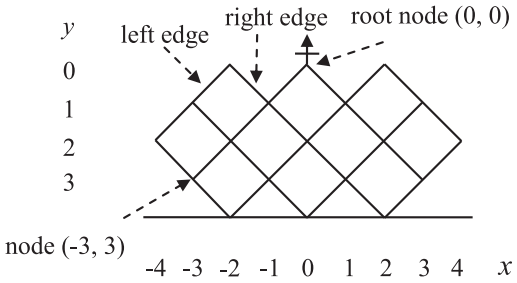

Fig. 5. An abstract diamond fabric.

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).

3. AUTOMATED MAPPING

In this section, we first discuss how to compute the product terms of a given Boolean function. Then, we present the proposed method for mapping the product terms into a SET array. Here, we first assume that each edge in a SET array can be configured independently without any constraint. In Section 7, we will extend the mapping method considering the granularity and fabric constraints.

3.1. Product Term Computation

To compute the product terms of a given Boolean function, we first build its ROBDD. Next, we traverse the ROBDD to collect the paths that lead to the 1 terminal. In this work, we use the *CUDD* package [Somenzi 2009] 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 a SET array, the number of product terms we consider could affect the mapping results. In general, more product terms result in larger area cost. Thus, before collecting product terms, we will try to minimize the ROBDD by performing BDD reordering. For simplification, we use the BDD reordering heuristic *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. However, because the BDD reordering operation is originally 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 after reordering.

Note that although there are other methods, like Espresso[1], which could compute more concise product terms, we use the BDD-based computation method, 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. Having multiple conducting paths leads to a higher fanout number that is not preferred for SET devices due to their low-drive strength.

3.2. Product Term Mapping

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

[1]Expresso. http://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm.

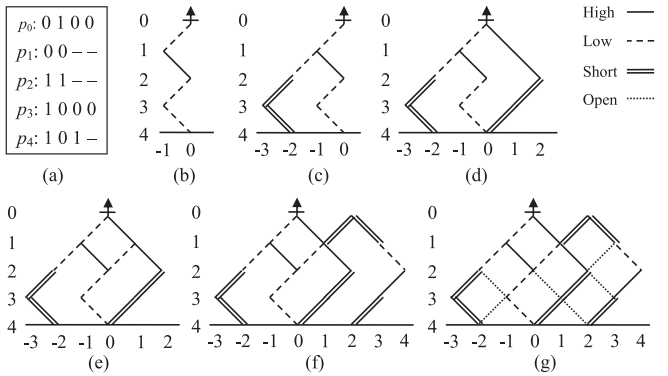

Fig. 6. A mapping example. (a) Product terms. (b) The mapping result of p_0. (c) The mapping result of $p_0 + p_1$. (d) The mapping result of $p_0 + p_1 + p_2$. (e) The mapping result of $p_0 + p_1 + p_2 + p_3$. (f) The mapping result of $p_0 + p_1 + p_2 + p_3 + p_4$. (g) The final mapping result.

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 (0), we find an active high (low) edge for it if applicable; otherwise, we configure an edge as active high (low) for it. However, if the bit value is – (don't care), 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 Figure 6 to demonstrate the mapping approach. There are five product terms, $p_0 = 0100$, $p_1 = 00--$, $p_2 = 11--$, $p_3 = 1000$, and $p_4 = 101–$ as shown in Figure 6(a). They are exactly the product terms of the example in Figure 4. First, let us consider $p_0 = 0100$. 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 both $n_{(0,2)}.left$ and $n_{(-1,3)}.right$ as *low* for the last two bits 00. The mapping result is shown in Figure 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. This method could compress the mapping result to reduce area cost.

Next, for $p_1 = 00--$, because the first bit is the same as that of p_0, we partially reuse this mapping result. Then, we configure $n_{(-1,1)}.left$ as *low* for the second bit 0. For the third bit 1, we do not configure $n_{(-2,2)}.right$ as *short* for it. This is because if we do so, we then need to configure $n_{(-1,3)}.left$ as *short* for the last bit –, and it will construct a path $n_{(0,0)} \rightarrow n_{(-1,1)} \rightarrow n_{(0,2)} \rightarrow n_{(-1,3)} \rightarrow n_{(-2,4)}$, which corresponds to an invalid product term 010–. Thus, we configure both $n_{(-2,2)}.left$ and $n_{(-3,3)}.right$ as *short* for the last two bits. The mapping result is shown in Figure 6(c).

For $p_2 = 11--$, after we configure $n_{(0,0)}.right$ as *high* for the first bit 1, we do not configure $n_{(1,1)}.left$ as *high* for the second bit 1. The reason is similar to that of mapping p_1. If we configure $n_{(1,1)}.left$ as *high*, we then need to configure both $n_{(0,2)}.right$ and $n_{(1,3)}.left$ as *short* for the last two bits, constructing a path which corresponds to an invalid product term $01--$. Thus, we configure $n_{(1,1)}.right$ as *high* for the second bit 1, and then configure both $n_{(2,2)}.left$ and $n_{(1,3)}.left$ as *short* for the last two bits. The mapping result is shown in Figure 6(d).

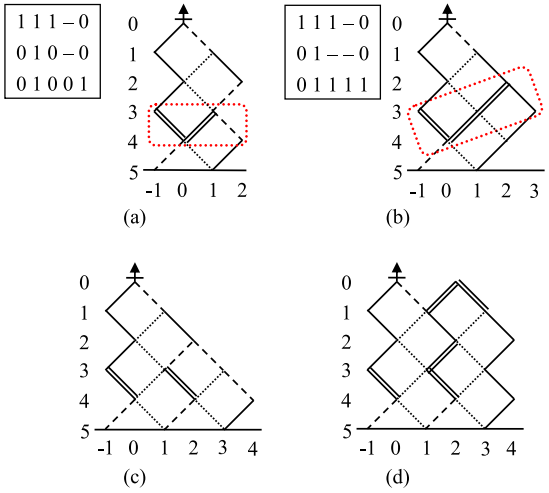

Fig. 7. Incorrect mapping examples.

Next, let us consider p_3 = 1000. After finding $n_{(0,0)}.right = high$ for the first bit 1, we consider to configure $n_{(1,1)}.left$ as *low* for the second bit 0. Because $n_{(0,2)}.left = low$ and $n_{(-1,3)}.right = low$ are compatible to the last two bits 00. We can safely configure $n_{(1,1)}.left$ as *low*. The mapping result is shown in Figure 6(e).

For p_4 = 101–, we first reuse $n_{(0,0)}.right = high$ for the first bit 1. Next, we do not reuse $n_{(1,1)}.left = low$ for the second bit 0. This is because $n_{(0,2)}.left = low$ is not compatible to the third bit 1, and if we then configure $n_{(0,2)}.right$ as *high* for the third bit 1, it will construct a path corresponding to an invalid product term. Thus, we expand the structure by configuring both $n_{(2,0)}.left$ and $n_{(2,0)}.right$ as *short*, and start from $n_{(3,1)}$ to map the last three bits 01–. The mapping result is shown in Figure 6(f). Finally, we configure all the non-configured edges as *open*, and obtain the final mapping result in Figure 6(g).

To avoid constructing an invalid path, we need to prevent two paths from merging and then branching during mapping. Thus, when we detect a merging node, like $n_{(-1,3)}$ for p_1, and $n_{(0,2)}$ for p_2, p_3, or p_4, we will check if there exists only one path from the merging node and if the path is compatible to the remaining bits. If not, there will exist an invalid path. As a result, 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 and we have to add another rule considering the conducting path issue. Figure 7(a) and Figure 7(b) show two mapping examples, which are incorrect mapping results while satisfying the merging and branching rule.

In Figure 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 also could be composed of the edges at the different rows. For example, Figure 7(b) shows a partial conducting path that crosses

ALGORITHM 1: Product Term Mapping

Input: An unconfigured SET array and product terms *PTs*.
Output: A configured SET array.
Configure $n_{(0,0)}$.*left* and $n_{(0,0)}$.*right* based on the first bit values of the product terms in *PTs*;
for *each product term t in PTs* **do**
    if ***LeftConfigure***$(t, 0, 0)$ **then**
        continue;
    end
    if ***RightConfigure***$(t, 0, 0)$ **then**
        continue;
    end
    Expand(t);
end
Configure all the edges that are not configured yet as *open*;

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.

In an abstract diamond fabric, a single diamond has two upper edges and two lower edges. A necessary condition for causing a partial conducting path is that there exist two pairs of two adjacent conducting edges: one pair is two lower edges of a diamond that could be conducting simultaneously, and the other pair is two upper edges of a diamond that could be conducting simultaneously. For example, in Figure 7(a), the right edge of $n_{(-1,3)}$ and the left edge of $n_{(1,3)}$ are the two lower edges of a diamond, and the left and right edges of $n_{(1,3)}$ are the two upper edges of a diamond. One simple method for avoiding partial conducting paths is to ensure that one of the mentioned two pairs of two adjacent conducting edges is never constructed. Thus, if a configuration results in a merging node, we check if the two edges connecting to the merging node could be conducting simultaneously. If so, we avoid this configuration. With this method, we can prevent two lower edges of a diamond from conducting simultaneously. Figure 7(c) and Figure 7(d) show the correct mapping results for the product terms in Figure 7(a) and Figure 7(b), respectively.

Note that because the root node has only two edges (left and right), in order to successfully map all the 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 the 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 the product terms. Thus, in the above example in Figure 6, actually we will configure n_{00}.*left* as *low* and n_{00}.*right* as *high* before mapping the product terms.

Algorithm 1 is the proposed algorithm for product term mapping. In the algorithm, 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 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 depth-first search (DFS)-like method to construct a path for it. **LeftConfigure()** (Algorithm 2) and **RightConfigure()** (Algorithm 3) 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()** (Algorithm 4). Finally, we configure all the edges that are not configured yet as *open*.

ALGORITHM 2: LeftConfigure

Input: A product term t, x coordinate x, and y coordinate y.
Output: A Boolean value.
if $n_{(x,y)}$.*left is incompatible to the* y^{th} *bit in* t **then**
  return 0;
end
if $n_{(x-1,y+1)}$ *is a merging node and there is more than one path from* $n_{(x-1,y+1)}$ **then**
  return 0;
end
if *the configuration of* $n_{(x,y)}$.*left will make the left edge of* $n_{(x,y)}$ *and the right edge of* $n_{(x-2,y)}$ *could be conducting simultaneously* **then**
  return 0;
end
if $n_{(x,y)}$.*left is empty* **then**
  configure it based on the mapping rules;
end
if $x - 1 < 0$ **then**
  if ***RightConfigure****(t,* $x - 1$, $y + 1$*)* **then**
    return 1;
  end
  if ***LeftConfigure****(t,* $x - 1$, $y + 1$*)* **then**
    return 1;
  end
end
if $x - 1 \geq 0$ **then**
  if ***LeftConfigure****(t,* $x - 1$, $y + 1$*)* **then**
    return 1;
  end
  if ***RightConfigure****(t,* $x - 1$, $y + 1$*)* **then**
    return 1;
  end
end
Undo $n_{(x,y)}$.*left* if necessary, and **return** 0;

In **LeftConfigure()**, we first check if the left edge of a node $n_{(x,y)}$ is incompatible to the y^{th} bit in t. They are incompatible 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 compatible, 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, the configuration of $n_{(x,y)}$.*left* will make two paths merge and branch. If not, we further check if the configuration of $n_{(x,y)}$.*left* will make the left edge of $n_{(x,y)}$ and the right edge of $n_{(x-2,y)}$ could be conducting simultaneously. If not, we then configure $n_{(x,y)}$.*left* based on the mapping rules when $n_{(x,y)}$.*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 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

ALGORITHM 3: RightConfigure

Input: A product term t, x coordinate x, and y coordinate y.
Output: A Boolean value.
if $n_{(x,y)}$.*right is incompatible to the* y^{th} *bit in* t **then**
  return 0;
end
if $n_{(x+1,y+1)}$ *is a merging node and there is more than one path from* $n_{(x+1,y+1)}$ **then**
  return 0;
end
if *the configuration of* $n_{(x,y)}$.*right will make the right edge of* $n_{(x,y)}$ *and the left edge of* $n_{(x+2,y)}$ *could be conducting simultaneously* **then**
  return 0;
end
if $n_{(x,y)}$.*right is empty* **then**
  configure it based on the mapping rules;
end
if $x - 1 < 0$ **then**
  if ***RightConfigure****(t,* $x + 1$*,* $y + 1$*)* **then**
    return 1;
  end
  if ***LeftConfigure****(t,* $x + 1$*,* $y + 1$*)* **then**
    return 1;
  end
end
if $x - 1 \geq 0$ **then**
  if ***LeftConfigure****(t,* $x + 1$*,* $y + 1$*)* **then**
    return 1;
  end
  if ***RightConfigure****(t,* $x + 1$*,* $y + 1$*)* **then**
    return 1;
  end
end
Undo $n_{(x,y)}$.*right* if necessary, and **return** 0;

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.

4. PRODUCT TERM SORTING

In this section, we present four different product term-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:

4.0.1. LexSort. We sort the product terms by comparing the bit values from the first bit with the relationship: $- > 1 > 0$. For example, Figure 8(b) shows the sorting result of the product terms in Figure 8(a). Using *LexSort*, two product terms having 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.

4.0.2. InertiaSort. Each product term has an inertia value that is the number of bit value matches with all the other product terms. We sort the product terms from large

ALGORITHM 4: Expand

```
Input: A product term t.
Output: A Boolean value.
Determine the expansion direction (left or right) based on the first bit in t.
if expansion direction is left then
    x = –2;
else
    x = 2;
end
while 1 do
    Configure n(x,0).left and n(x,0).right as short if they are empty;
    if x – 1 < 0 then
        if RightConfigure(t, x – 1, 1) then
            return 1;
        end
        if LeftConfigure(t, x – 1, 1) then
            return 1;
        end
        x = x – 2;
    end
    if x – 1 ≥ 0 then
        if LeftConfigure(t, x + 1, 1) then
            return 1;
        end
        if RightConfigure(t, x + 1, 1) then
            return 1;
        end
        x = x + 2;
    end
end
```

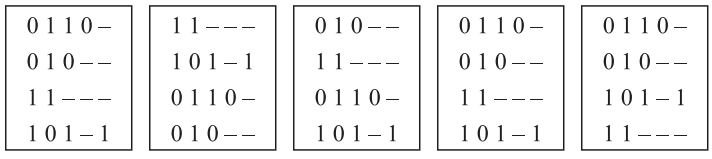

Fig. 8. Four different sorting results. (a) Original. (b) *LexSort*. (c) *InertiaSort*. (d) *ForInertiaSort*. (e) *Back-ForInertiaSort*.

to small by the inertia values. Figure 8(c) shows the sorting result. The inertia value of the first product term in Figure 8(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.

4.0.3. ForInertiaSort. Unlike the inertia value, a product term's forward inertia value is the number of continuous bit value matches with all the other product terms from the first bit. We sort product terms from large to small by the forward inertia values. Figure 8(d) shows the sorting result. The forward inertia value of the first product term in Figure 8(d) is 1 + 1 + 0 + 0 + 0 = 2. This is because only the second product

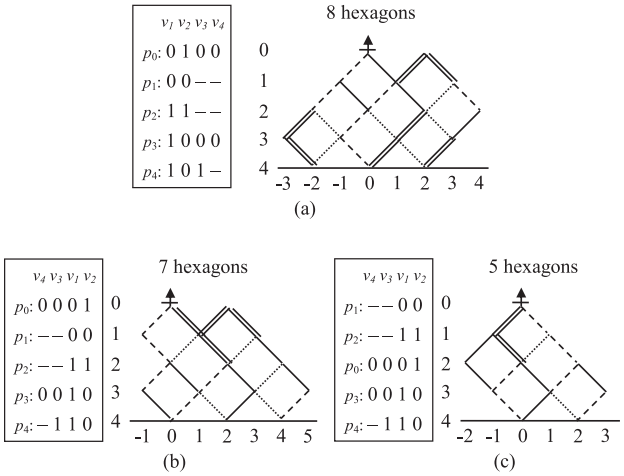

Fig. 9. An example of variable reordering and product term sorting. (a) Original. (b) Variable reordered. (c) Variable reordered and product term sorted.

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 with others from the first bit 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).

4.0.4. BackForInertiaSort. Conversely, a product term's backward inertia value is the number of continuous bit value matches with all the other product terms from the last bit to the first bit. 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. This is why this sorting method is named *BackForInertiaSort*. The sorting result is shown in Figure 8(e). Unlike the result in Figure 8(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.

5. VARIABLE REORDERING

In this section, we present a heuristic for reordering the variables in the computed product terms. The objective is to reduce the area cost required for mapping the product terms.

First, we use an example to demonstrate our motivation. In Figure 9, suppose we use the number of hexagons to measure area cost. The set of product terms requires 8 hexagons for mapping as shown in Figure 9(a). However, if we reorder the variables in the product terms as shown in Figure 9(b), we obtain a new mapping result having less area cost, 7 hexagons, by using the same mapping approach. Thus, the variable reordering could affect the mapping results.

Unfortunately, it is difficult to determine a variable order which results in the least area cost. Thus, in this work, we use an empirical approach to develop a reordering heuristic. We separately applied the four product-sorting methods mentioned in Section 4 to map a set of MCNC benchmarks [Yang 1991]. The experimental results that will be presented in Section 8 show that using *ForInertiaSort* obtains the better mapping results for more benchmarks. Thus, increasing the forward inertia value of

each product term possibly enhances the mapping results. As a result, we develop a greedy method to determine the variable order which aims to maximize the forward inertia value of each product term.

Given n variables, $v_1 \sim v_n$, we first prepare n positions for them. Starting from the first position, we iteratively assign a variable to it until all the positions are occupied. For example, we first select a variable for the first position. Next, we select a variable from the remaining variables for the second position and so on. Finally, the new variable order is obtained from the first position to the last position. The variable selection rule is as follows: For the position under consideration at each iteration, we select the variable which will result in the largest forward inertia value, when the variable is assigned to the position. Because not all variables have been assigned to a position during the reordering process, we only consider the variables having been assigned when computing the forward inertia value at each iteration. We will use an example to demonstrate the variable selection method in the following paragraphs.

Additionally, for the first position, we prevent from selecting a variable which simultaneously has three different kinds of bit values, 0, 1, and –, among all the product terms, like v_3 in the example in Figure 9(a). This is because the root node in a SET array has only two edges (left and right). Thus, if we select such a variable at the first position, we need to divide each product term having – in the first bit into two product terms (one begins with 0 and the other begins with 1) for successfully mapping them. As a result, this method increases the number of product terms and could result in more area cost. Thus, a variable having three different bit values among all the product terms has a lower priority to be selected at the first position, even it has a higher forward inertia value.

We use the example in Figure 9(a) to demonstrate the reordering method. In this example, there are four variables, $v_1 \sim v_4$. First, let us consider the first position. In the first variable column v_1 = 00111, the first two bits 00 are identical and the last three bits 111 are identical as well. Thus, for the first two bits, each bit is the same with the other bit. Also, each bit in the last three bits is identical to the other two bits. Thus, if v_1 is selected at the first position, the forward inertia value is $1+1+2+2+2=8$. Based on the same method, if the second variable v_2 = 10100 is selected at the first position, the forward inertia value is 1 + 2 + 1 + 2 + 2=8. For v_3 = 0 – –01, because it simultaneously has 0, 1, and –, we prevent from selecting it at the first position. For v_4 = 0 – –0–, the forward inertia value is $1+2+2+1+2=8$. Thus, based on the forward inertia values, v_1, v_2, and v_4 have the same priority to be selected at the first position. Here, suppose we select v_4 at the first position. The result is shown in Figure 9(b). In the experiments, when there is more than one variable having the same priority to be selected, we randomly choose one from them.

Next, let us consider the second position. Because v_4 has been assigned at the first position, we also need to consider its bit values, when we determine a variable for the second position by computing the forward inertia value. First, for v_1 = 00111, if it is selected at the second position, the forward inertia value from v_1 is 0 + 0 + 1 + 0 + 1=2 (only p_2 and p_4 have continuous bit value matches). Next, for v_2 = 10100, the forward inertia value is 0 + 1 + 0 + 0 + 1=2 (only p_1 and p_4 have continuous bit value matches), when it is selected. Finally, when v_3 = 0 – –01 is selected, the forward inertia value is 1 + 1 + 1 + 1 + 0=4 (p_0 and p_1 have continuous bit value matches with p_3 and p_2, respectively). As a result, we select v_3 at the second position and continue to consider the next position.

For the third position, the inertia values from v_1 and v_2 are both 0. Thus, we can select either one of them. Here, suppose we select v_1 and v_2 for the third and last positions, respectively. The variable reordering and the mapping results are shown in Figure 9(b).

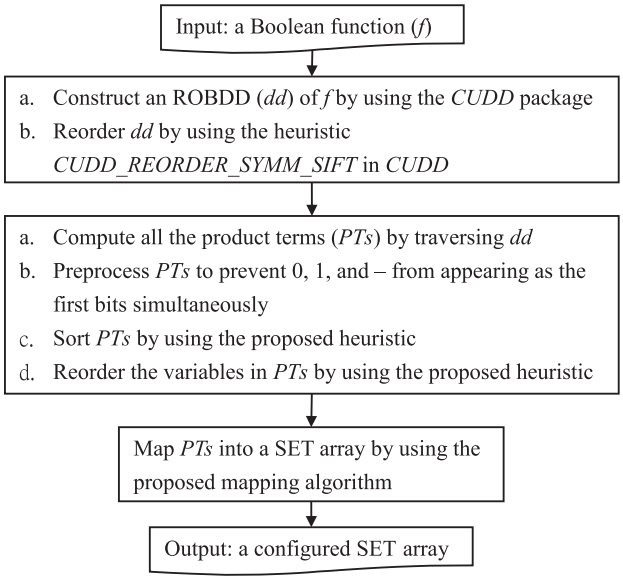

Fig. 10. The overall mapping flow.

Furthermore, if we sort the product terms in Figure 9(b) by using *ForInertiaSort* and then map them, we obtain a better mapping result as shown in Figure 9(c).

6. OVERALL MAPPING FLOW

Figure 10 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. At the end, we sort *PTs* and then reorder the variables in *PTs* by using the proposed heuristics. In step 3, we map *PTs* into a SET array by using the proposed mapping algorithm. Finally, we get a configured SET array.

7. 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.

7.1. Configuration with Granularity Constraint

The configuration circuitry, which involves metal wires, is used to program the SET into open, short, or active mode. As the metal wire pitches are larger than nanowire pitches, the circuit density would be determined by the number of metal wires. Limiting the number of metal wires can lead to higher circuit density at a loss of flexibility. Thus, the granularity constraint, where the same configuration circuitry is used to program multiple SETs simultaneously, was introduced by Eachempati et al. [2008]. Consequently, 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, we divide each product

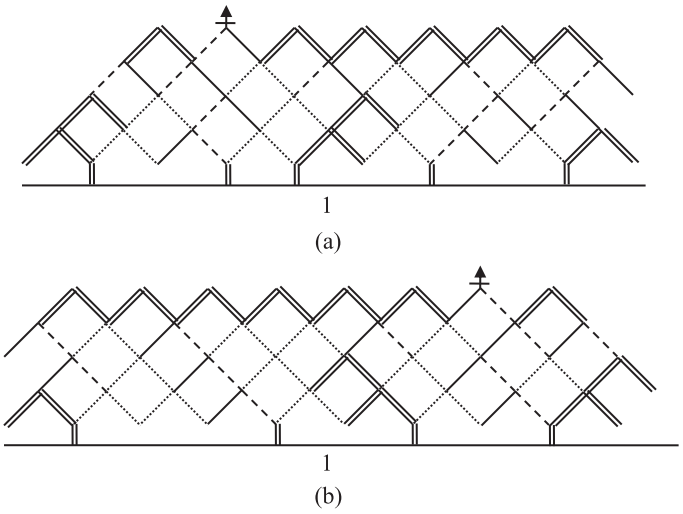

Fig. 11. The mapping results with (a) granularity constraint, and (b) fabric constraint.

term whose first bit is – into two product terms: one has the first bit 0 and the other has the first bit 1, unless the first bits of all the product terms are –.

Algorithm 1 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 are configured separately. To consider this granularity constraint; however, we configure them at the same time. For example, when we configure one edge of a node as *high* (*low*), we also configure the other edge as *low* (*high*). Similarly, when one edge is *short*, the other edge is *short* as well.

Figure 11(a) shows the mapping result for the same set of product terms in Figure 9(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 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, we also prevent two lower edges of a diamond from conducting simultaneously to avoid creating partial conducting paths. For brevity, we omit the detailed mapping algorithm considering the granularity constraint.

7.2. Configuration with Fabric Constraint

In SET array implementation, the inputs to the active edges in a row are supplied by metal wires. We need two wires to supply both the normal and complement of an input to a row. Each 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 and it is fixed during manufacturing.

For example, using x to control all left edges and x' to control all right edges results in a symmetric fabric proposed in Eachempati et al. [2008]. In this mapping algorithm, we also apply the symmetric fabric constraint. In the future, we will extend our mapping algorithm to accommodate any fabric specification.

Under such a constraint, 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*)

((*low*, *high*)) can be changed to (*low*, *high*) ((*high*, *low*)) by swapping the normal value and its complement in the control input signal for the row.

To satisfy this symmetric fabric constraint, we need to identify which combination ((*high*, *low*) or (*low*, *high*)) will appear at a certain row. One method is to follow the first configuration obtained 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 either (*high*, *low*) or (*low*, *high*) to appear in an entire 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*, that is, only (*high*, *low*) is allowed. For simplification, we use the second method in the experiments of this work.

Figure 11(b) shows the mapping result for the same set of product terms in Figure 6(a) considering the fabric constraint. In this example, only (*high*, *low*), (*short*, *short*), and (*open*, *open*) are allowed. Since the fabric constraint is more restrictive than the granularity constraint, more area is required for most benchmarks. Additionally, if a mapping result satisfies the fabric constraint, it satisfies the granularity constraint as well.

8. EXPERIMENTAL RESULTS

We implemented the algorithm in C language. The experiments were conducted on a 3.0 GHz Linux platform (CentOS 4.8). The benchmarks are from the MCNC benchmark suite [Yang 1991]. 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. The experiments consist of two parts: First, we compare different product term sorting-heuristics and mapping constraints without reordering the variables in the computed product terms. Next, we reorder the variables before sorting the product terms to show the effectiveness of the proposed variable-reordering heuristic.

Table I summarizes the experimental results of the first part. Column 1 lists the benchmarks. Except the *C17* benchmark, all the benchmarks have the 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 of applying the granularity and fabric constraints by using *ForInertiaSort* only. This is because the *ForInertiaSort* heuristic has better results in 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 16, 16, 18, and 18 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 61 and 68 hexagons, respectively.

Table I demonstrates that the proposed approach can automatically map the benchmarks. The experimental results also show that there is no specific sorting heuristic that completely outperforms the others for all the benchmarks. By all accounts, *ForInertiaSort* results in the best mapping for considering all the benchmarks. Additionally, when the constraints are considered, the number of configured hexagons

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.	FInert.	BFInert.	FInert.	FInert.
C17	5	2	8	*16	*16	18	18	61	68
cm138a	6	8	48	177	152	*144	*144	464	528
x2	10	7	33	*149	152	153	154	725	790
cm85a	11	3	49	219	197	197	*195	608	528
cm151a	12	2	25	406	427	*400	*400	885	1045
cm162a	14	5	37	292	336	294	*287	1077	1163
cu	14	11	24	240	242	*238	*238	609	662
cmb	16	4	26	191	214	*188	*188	711	808
cm163a	16	5	27	275	*257	260	260	907	1029
pm1	16	13	41	337	342	*335	*335	1186	1239
pcle	19	9	45	*291	292	293	293	1553	1775
sct	19	15	142	1890	*1661	1725	1741	4665	5186
cc	21	20	57	618	658	*585	603	2214	2306
i1	25	16	38	632	650	*627	*627	1773	1920
lal	26	19	160	1968	2157	1832	*1799	7838	8684
pcler8	27	17	68	*737	850	*737	*737	3160	3435
frg1	28	3	399	*5993	5602	5612	5612	11029	13731
c8	28	18	94	*836	884	881	894	4663	4869
term1	34	10	1246	23494	25297	*22426	23856	70844	80293
count	35	16	184	1936	1861	*1336	1465	13509	14678
unreg	36	16	64	1288	*1259	1280	1280	4518	4632
b9	41	21	352	*6333	8650	6478	6542	24272	22089
cht	47	36	92	*2380	2390	*2380	*2380	7857	7934
apex7	49	37	1440	36252	44001	*35999	36317	123003	135543
example2	85	66	430	9737	10164	9623	*9494	53597	50471
Best				**8**	**4**	**12**	**12**		
Total				**96687**	**108711**	**94041**	**95859**	**341728**	**365406**

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. The CPU time includes the required time for computing product terms.

Furthermore, when we apply the proposed variable-reordering heuristic before sorting the product terms, we can obtain better mapping results for most benchmarks. The experimental results are shown in Table II and they correspond to that shown in Table I.

Table II shows that, for constraint-free mapping with *ForInertiaSort*, we can save a total of 14871 (94041–79170) hexagons by reordering the variables. Additionally, although the proposed variable-reordering heuristic aims to maximize the forward inertia value, other product term-sorting heuristics, *LexSort*, *InertiaSort*, and *Back-ForInertiaSort*, also take advantage of the variable reordering. Even though considering the granularity and fabric constraints, the proposed variable-reordering heuristic is effective as well. Although the reordering process requires some CPU time overhead, among the benchmarks, the largest CPU time overhead is less than 4 minutes required for the *apex7* benchmark, which is reasonable.

In summary, both product term sorting and variable reordering are important for mapping the product terms. Although the proposed mapping algorithm is not an optimal approach, the experimental results show that it is efficient and effective. More

Table II. The Experimental Results of Using Different Product Term-Sorting Heuristics and Mapping Constraints with Variable Reordering

Bench.	PI	PO	PT	Constraint-free				Granu.	Fabric
				Lex	Inert.	FInert.	BFInert.	FInert.	FInert.
C17	5	2	8	*14	14	*12	*12	49	73
cm138a	6	8	48	142	156	*120	*120	400	501
x2	10	7	33	122	125	*107	*107	703	772
cm85a	11	3	49	227	*197	205	205	569	533
cm151a	12	2	25	172	192	*147	*147	467	589
cm162a	14	5	37	206	*199	214	214	923	1046
cu	14	11	24	170	167	*164	*164	595	566
cmb	16	4	26	*80	148	*80	*80	772	853
cm163a	16	5	27	150	142	*130	*130	747	774
pm1	16	13	41	*218	225	*218	*218	1164	1261
pcle	19	9	45	*247	275	*247	*247	1316	1402
sct	19	15	142	957	*841	889	885	4620	5165
cc	21	20	57	463	454	*452	*452	1947	2067
i1	25	16	38	432	429	*422	*422	1680	1753
lal	26	19	160	*1191	1287	1376	1376	7706	7233
pcler8	27	17	68	618	647	*617	*617	3157	3706
frg1	28	3	399	*4999	5055	*4999	5404	19534	19923
c8	28	18	94	708	695	*667	*667	4297	4715
term1	34	10	1246	*19368	19940	19395	20768	66979	60849
count	35	16	184	1906	1814	*1528	*1528	11337	13569
unreg	36	16	64	*593	*593	*593	*593	4394	4512
b9	41	21	352	*3859	4508	3969	4085	26059	26845
cht	47	36	92	1708	1709	*1708	*1708	8398	8456
apex7	49	37	1440	*38237	39271	*34601	36386	117762	126496
example2	85	66	430	6326	6410	6310	*6302	42900	38944
Best				**9**	**4**	**18**	**17**		
Total				**83113**	**85493**	**79170**	**82837**	**328475**	**332603**
Total in Table I				**96687**	**108711**	**94041**	**95859**	**341728**	**365406**
Improvement				**13574**	**23218**	**14871**	**13022**	**13253**	**32803**

important, the automatic mapping approach solves the inefficiency problem that previous manual approach [Eachempati et al. 2008] suffered from.

9. CONCLUSION

In this article, 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 work suffered 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 and one variable-reordering heuristics are proposed to enrich the approach. The granularity and fabric constraints are also handled by the proposed approach. The experimental results show its effectiveness and efficiency of mapping a set of MCNC benchmarks. Our automatic mapping is a key enabler for using the promising BDD-based SET technology.

REFERENCES

Bryant, R. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput. 35*, 8, 677–691.

Chen, Y. C., Eachempati, S., Wang, C. Y., Datta, S., Xie, Y., and Narayanan, V. 2011. Automated mapping for reconfigurable single-electron transistor arrays. In *Proceedings of the Design Automation Conference*. 878–883.

Eachempati, S., Saripalli, V., Narayanan, V., and Datta, S. 2008. Reconfigurable bdd-based quantum circuits. In *Proceedings of the International Symposium on Nanoscale Architectures*. 61–67.

Hasegawa, H. and Kasai, S. 2001. Hexagonal binary decision diagram quantum logic circuits using schottky in-plane and wrap gate control of gaas and ingaas nanowires. *Physica E 11*, 2–3, 149–154.

Kasai, S., Yumoto, M., and Hasegawa, H. 2001. Fabrication of gaas-based integrated 2-bit half and full adders by novel hexagonal bdd quantum circuit approach. In *Proceedings of the International Symposium on Semiconductor Device Research*. 622–625.

Keating, M., Flynn, D., Aitken, R., Gibbons, A., and Shi, K. 2007. *Low Power Methodology Manual: For System-on-Chip Design*. Springer.

Keckler, S. W., Olukotun, K., and Hofstee, H. P. 2009. *Multicore Processors and Systems*. Springer.

Liu, L., Saripalli, V., Narayanan, V., and Datta, S. 2011. Device circuit co-design using classical and non-classical iii-v multi-gate quantum-well fets (muqfets). In *Proceedings of the IEEE International Electron Devices Meeting*.

Piguet, C. 2006. *Low-Power CMOS Circuits: Technology, Logic Design and CAD Tools*. CRC Press.

Saripalli, V., Liu, L., Datta, S., and Narayanan, V. 2010. Energy-delay performance of nanoscale transistors exhibiting single electron behavior and associated logic circuits. *J. Low Power Electron. 6*, 415–428.

Shin, S. J., Jung, C. S., Park, B. J., Yoon, T. K., Lee, J. J., Kim, S. J., Choi, J. B., Takahashi, Y., and Hasko, D. G. 2010. Si-based ultrasmall multiswitching single-electron transistor operating at room-temperature, *Appl. Phys. Lett. 97*, 103101–1–103101–3.

Somenzi, F. 2009. Cudd: Cu decision diagram package - release 2.4.2. http://vlsi.colorado.edu/_fabio/CUDD/.

Yang, S. 1991. Logic synthesis and optimization benchmarks, version 3.0. Tech. rep., Microelectronics Center of North Carolina.

Received September 2011; revised January 2012, February 2012; accepted February 2012