Sneak Path Free Reconfiguration With Minimized Programming Steps for Via-Switch Crossbar-Based FPGA

Ryutaro Doi, *Student Member, IEEE*, Jaehoon Yu, *Member, IEEE*, and Masanori Hashimoto, *Senior Member, IEEE*

***Abstract*—Field programmable gate array (FPGA) that utilizes via-switches, which are a kind of nonvolatile resistive RAMs, for crossbar implementation is attracting attention due to higher integration density and performance. However, programming via-switches arbitrarily in a crossbar is not trivial since a programming current must be provided through signal wires shared by multiple via-switches. Consequently, depending on the previous programming status in sequential programming, unintentional switch programming may occur due to signal detour, which is called the sneak path problem. This article identifies the circuit status that causes the sneak path problem and proposes a sneak path avoidance method that gives sneak path free programming order of via-switches in a crossbar. We prove that sneak path free programming order necessarily exists for arbitrary ON-OFF patterns in a crossbar as long as no loops exist. This article also proposes a partial reconfiguration method that achieves the minimum number of switch programming steps while avoiding the sneak path problem. This method contributes to the extension of via-switch lifetime and fast reconfiguration of the via-switch FPGA. Experimental results show that the proposed partial reconfiguration method reduces the number of programmed switches by 77.4% compared to the conventional approach. This 77.4% reduction improves the number of reconfigurations of the via-switch FPGA by 4.4× and reduces reconfiguration time by 77.4%.**

***Index Terms*—Crossbar programming, nonvolatile via-switch field programmable gate array (FPGA), partial reconfiguration, sneak path, switch programming order.**

I. Introduction

Field programmable gate arrays (FPGAs) are gaining their popularity since the development cost of application-specific integrated circuits (ASICs) is elevating due to the device miniaturization and larger scale integration. However, conventional FPGAs are still inferior to ASICs regarding operating speed, power consumption, and implementation area [1]. These drawbacks originate from a large number of programmable switches that are included in FPGAs to acquire reconfigurability. In static random access memory (SRAM)-based FPGAs, which are the most widely used FPGAs, a programmable switch is composed of a switch gate, such as transmission gate and multiplexer, and an SRAM cell to hold the switch state. These components consist of transistors, and hence the switch gate has high resistance and large capacitance and the SRAM cell having six transistors consumes the large area. Therefore, SRAM-based programmable switches lead to the degradation of interconnect performance and area efficiency [2].

To overcome the drawbacks of conventional FPGAs, FPGAs that exploit resistive random access memories (RRAMs) as programmable switches instead of SRAM-based ones are widely studied [3]–[9]. In these RRAM-based FPGAs, however, one or two access transistors per a programmable switch are required for switch programming. The access transistor is relatively large despite the small footprint of an RRAM-based switch, and hence it prevents further area reduction. To eliminate access transistors, nonvolatile via-switch is actively developed [10]–[12]. The via-switch consists of atom switches, which are a kind of nonvolatile RRAMs developed for application to FPGAs, and varistors in place of access transistors.

In the via-switch FPGA, the crossbar, which has a via-switch at each intersection of horizontal signal wire and vertical signal wire, is responsible for the signal routing. However, programming those via-switches arbitrarily in a crossbar is not trivial since a programming voltage must be given through signal wires that are shared by multiple via-switches. In this case, unintentional switch programming may occur depending on the previous programming status due to signal detour, which is called the sneak path problem (SPP). This problem interferes the reconfiguration of FPGA, and hence the verification of occurrence conditions and countermeasures is crucially important.

This article is an extension of our preliminary work reported in [13]. Reference [13] identifies the crossbar programming status that causes the sneak path problem and proposes a method that provides a programming sequence of via-switches for the sneak path free programming. We prove that such an order for sneak path free programming must exist for arbitrary ON-OFF patterns in a crossbar as long as no loops exist,

Manuscript received July 5, 2019; revised October 25, 2019; accepted December 7, 2019. Date of publication December 17, 2019; date of current version September 18, 2020. This work was supported in part by Japan Society for the Promotion of Science KAKENHI under Grant JP17J10008, and in part by Japan Science and Technology Agency CREST under Grant JPMJCR1432, Japan. This article was recommended by Associate Editor W. Hung. *(Corresponding author: Ryutaro Doi.)*

The authors are with the Department of Information Systems Engineering, Graduate School of Information Science and Technology, Osaka University, Suita 565-0871, Japan (e-mail: doi.ryutaro@ist.osaka-u.ac.jp; yu.jaehoon@ist.osaka-u.ac.jp; hasimoto@ist.osaka-u.ac.jp).

Digital Object Identifier 10.1109/TCAD.2019.2960331

and devise an algorithm to find the programming order by representing the connection status of signal wires in a crossbar as a tree structure. This journal paper newly proposes a partial reconfiguration method that minimizes the number of switch programming steps while avoiding the sneak path problem. This method contributes to extending the lifetime of via-switches and speeding up the reconfiguration of the via-switch FPGA for both user programming and manufacturing test. Thanks to the proposed methods, any practical configurations of via-switch FPGA can be successfully programmed as we intend without the sneak path problem.

The remainder of this article is organized as follows. Section II explains the structure of via-switch FPGA and sneak path problem. In Section III, we investigate occurrence conditions of the sneak path problem and identify the circuit status that causes the sneak path. Section IV proposes a sneak path avoidance method that determines the sneak path free programming order of the via-switches in a crossbar. In Section V, we propose a partial reconfiguration method that reduces the number of switches to be programmed in the crossbar without causing the sneak path problems. Section VI quantitatively evaluates the advantage of the proposed methods. Concluding remarks are given in Section VII.

II. Via-Switch FPGA

A. Via-Switch

The via-switch is a nonvolatile, rewritable, and compact switch that is developed to implement a crossbar switch by Banno *et al.* [10], and it is composed of atom switches and varistors. Here, we explain the device structure, functionality, and characteristics in the following. The programming of the via-switch crossbar will be explained later in Section II-B.

The atom switch consists of a solid electrolyte sandwiched between copper (Cu) and ruthenium (Ru) electrodes as shown in Fig. 1(a). By applying a positive voltage to the Cu electrode, a Cu bridge is formed in the solid electrolyte, and the switch turns on. On the other hand, when a negative voltage is applied, Cu atoms in the bridge are reverted to the Cu electrode, and then the switch turns off. The switching between on-state and off-state is repeatable, and each state is nonvolatile. For improving the device reliability, the complementary atom switch (CAS) is devised, where it consists of two-atom switches connected in series with opposite direction as shown in Fig. 1(b). In the programming of CAS, a pair of the signal line and control line supply a programming voltage to each atom switch, and two-atom switches are programmed sequentially. During normal operation, on the other hand, only signal lines are used for routing [9].

To accurately provide the programming voltage only to the target atom switch in a switch array, the varistor is introduced into the via-switch. Fig. 2 shows the structure of via-switch, where the varistor is connected to the control terminal of CAS. When a voltage higher than the threshold value (programming voltage) is applied between the signal and control lines, the varistor supplies programming current to an atom switch. On the other hand, the varistor isolates the control lines from the signal lines during normal operation [10].

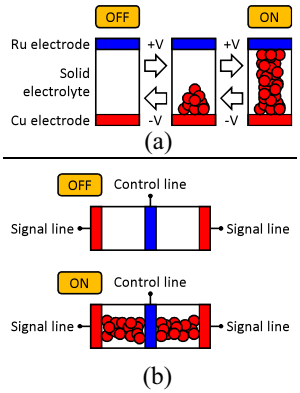


Fig. 1. Structure and operation of (a) atom switch and (b) CAS.

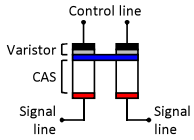


Fig. 2. Via-switch structure.

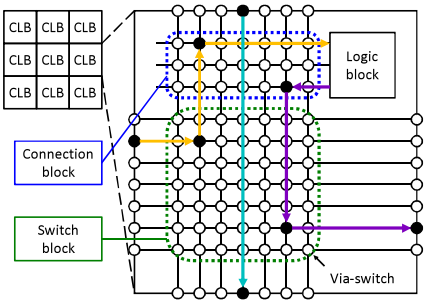


Fig. 3. Structure of via-switch FPGA.

Here, we summarize the main features of via-switches. The footprint, on-resistance, and capacitance are 18 F^2, 400 Ω, and 0.14 fF, respectively, [10], [14]. The via-switch can be reprogrammed about 1000 times [9]. Thanks to these characteristics, the area efficiency and performance of via-switch FPGA are dramatically improved compared to SRAM-based one. Ochi *et al.* [14] reported that the crossbar density is improved by 26×, and the delay and energy in the interconnection are reduced by 90% or more at 0.5 V operation.

B. Sneak Path Problem in Via-Switch FPGA

The structure of via-switch FPGA is an array of configurable logic blocks (CLBs), and each CLB is composed of a logic block and a crossbar where a via-switch is placed at each intersection of signal lines as shown in Fig. 3 [14]. The via-switch in the crossbar is responsible for the connection and disconnection between the horizontal and vertical signal lines. Besides, the top half of the crossbar serves as input and output multiplexers to the logic block and corresponds to the connection block in conventional FPGAs. On the other hand, the bottom half of the crossbar, which corresponds to the switch block, routes global interconnections. The logic block organizes combinational and sequential circuits.

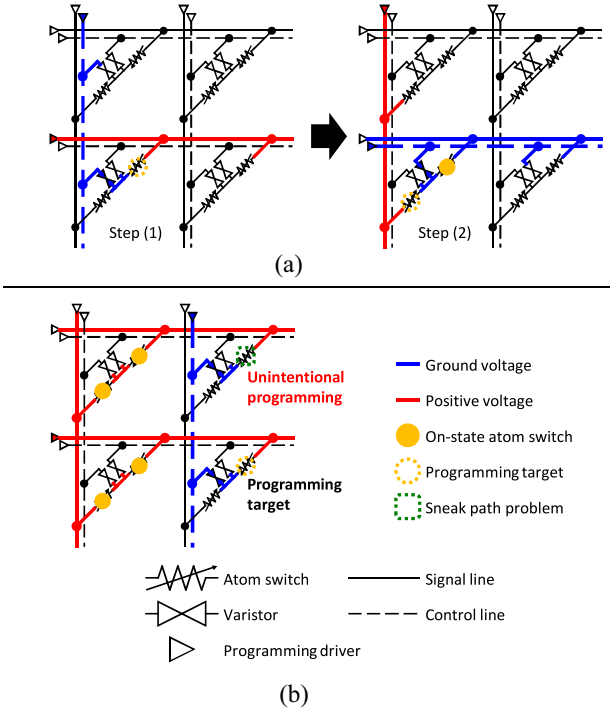


Fig. 4. (a) Via-switch-based crossbar structure and switch programming steps. (b) Sneak path problem in crossbar programming.

The following explains the programming of a via-switch crossbar and sneak path problem. Fig. 4 illustrates the via-switch-based crossbar structure. Both signal and control lines are aligned horizontally and vertically. Fig. 4(a) exemplifies programming steps in 2×2 crossbar where an atom switch is turned on at each step. A pair of the perpendicular signal and control lines crossing at the via-switch of interest are used for switch programming. Two programming drivers are activated at each step, and a positive voltage is given to one of the signal lines, and a ground voltage is given to one of the control lines. Other lines are floated. We can see that the via-switch at the bottom left is successfully turned on at STEPs (1) and (2). However, this crossbar programming structure may cause the sneak path problem depending on via-switches' ON-OFF patterns in a crossbar. For example, the programming of the bottom right via-switch in Fig. 4(b) cannot be performed correctly. The atom switch that composes the top right via-switch is under programming unintentionally since the positive voltage is provided through the on-state via-switches at the bottom left and top left. Such an unintentional switch programming due to signal detouring through on-state via-switches is the sneak path problem. The sneak path problem interferes the reconfiguration of FPGA, and hence it is essential to identify the occurrence conditions and find countermeasures of this problem.

C. Conventional Countermeasure for Sneak Path Problem and Advantages of Proposed Method

In the last decade, countermeasures for the sneak path problem in RRAM-based crossbars are widely studied [15]–[22]. These countermeasures can be categorized into three groups, namely, device-level structure modifications [16]–[18], voltage bias schemes [19], [20], and multistage reading schemes [21], [22]. Zangeneh and Joshi [16] and Lee *et al.* [17] proposed one transistor-one resistor (1T1R) and one diode-one resistor (1D1R) structures for each crossbar intersection, respectively. These structures mitigate the sneak path current since the transistor or diode act as a gating device, but the integration density of crossbars decreases due to the added gating devices. Jung *et al.* [18] adopted a complementary resistive switch (CRS), which is composed of two anti-serial memristors, as an intersection switch. This structure ensures that either memristor in a CRS is always off-state (high resistance) whenever the CRS retains logical 0 or 1, and hence it alleviates the sneak path problem. However, write and read operations for the CRS become complicated. In bias schemes, we have to apply VDD/3 or VDD/2 to unused wires for minimizing the sneak path current [19], [20]. These schemes need to drive all the wires, and therefore large switching power is consumed. Also, they require complicated controls and additional hardware overheads for write/read operations. Vontobel *et al.* [21] and Zidan *et al.* [22] proposed a reading scheme that reads the target switch multiple times while changing conditions to eliminate the sneak path effect.

In the above, we summarize the sneak path countermeasures reported in literature, but all the countermeasures focus on the sneak path problem in the crossbar used as a memory. On the other hand, we utilize the crossbar as programmable interconnections for FPGA implementation and solve the sneak path problem in the write operation. To reduce the signal propagation delay in crossbars for FPGA, we set on-resistance of a via-switch to 400 Ω [14] whereas the on-resistance for memory purpose is in a range of a few kilo-ohms to hundreds kilo-ohms [23]–[26]. This smaller on-resistance in FPGA purpose makes the sneak path problem more severe since the countermeasures that insert high-resistance gating devices to signal lines increase the delay and diminish the integration density. Besides, voltage bias countermeasures are undesirable due to the large power consumption and hardware overheads. Therefore, the sneak path countermeasure that does not degrade the FPGA performance and does not need hardware modifications is required.

As a sneak path countermeasure for FPGA purpose, Ochi *et al.* [14] have revealed that the sneak path problem can be avoided by imposing a programming constraint. This constraint allows multiple on-state via-switches on the same signal line only in one direction. In other words, this constraint prohibits the configurations in which multiple on-state via-switches exist in both the same horizontal line and the same vertical line such as Fig. 4(b). The authors also prove that there is no sneak path problem in the programming of any-sized crossbar under the programming constraint with mathematical induction. However, their countermeasure involves a clear disadvantage. The programming constraint prohibits some configurations of via-switch FPGA, and hence imposing the constraint leads to a decrease in the number of available configurations. Consequently, routing flexibility is limited. For

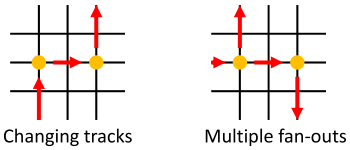


Fig. 5. Routing patterns that are prohibited in conventional countermeasure [14]. These patterns are often used in practical applications.

example, practical applications often use routing patterns illustrated in Fig. 5. The left pattern changes vertical routing tracks, and the right one realizes multiple fan-outs in the vertical direction. However, these patterns cannot be programmed in a crossbar when we prohibit multiple on-state switches in the same horizontal line [14]. To achieve the same function, we consume more interconnect resources due to detour routing.

This article, on the other hand, proposes a programming constraint free countermeasure for the sneak path problem in Sections IV and V. The proposed method can program arbitrary practical configuration patterns including the patterns depicted in Fig. 5. The advantages of the proposed method are as follows. The proposed method completely eliminates the programming status that causes the sneak path problem by arranging the programming order and accepts all the practical configuration patterns. The computational complexity of the proposed algorithm is low, and there is no additional hardware overhead. Furthermore, by eliminating programming constraints, we can simplify the algorithm and data structure in the routing CAD software.

III. Occurrence Conditions of Sneak Path Problem

This section clarifies the programming status of a crossbar that leads to the sneak path problem for developing a more efficient countermeasure. As explained in Section II-B, the atom switch at the intersection is turned on when a positive voltage is provided to the signal line and a ground voltage is provided to the control line. When the number of such intersections is two or more, the sneak path problem occurs. Focusing on the number of bends of the programming signal given to the signal line, we can classify the circuit status that causes the sneak path problem into two situations, namely, conditions (a) and (b) as shown in Fig. 6. In condition (a), the programming signal provided to the signal line bends twice or more, whereas the number of signal bends is one or zero in condition (b). The followings discuss each condition in detail. It should be noted that we only consider the programming operations to turn on the atom switch in the following because programming operations to turn on and off an atom switch are symmetrical operations and the same discussion can be done by swapping the voltage given to the signal line and control line.

In condition (a) of Fig. 6, two vertical signal lines SV1 and SV2 are connected by multiple on-state via-switches VS3 and VS4 in the same line SH2. When a programming signal is given to one of the signal lines SV1 and SV2, the same signal is provided to the other signal line, which means we cannot distinguish SV1 and SV2 anymore in the programming.

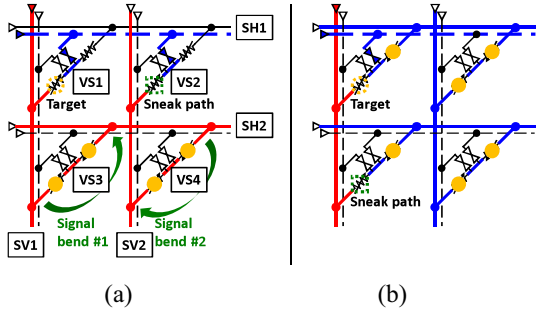


Fig. 6. Two occurrence conditions of sneak path problem. Condition (a) #bends of prog. signal given to signal line is two or more. Condition (b) #bends of prog. signal given to signal lines is zero or one.

Therefore, when we try to turn on the lower atom switch of via-switch VS1 in the line SV1, the atom switch in the same position of signal line SV2, i.e., lower atom switch of via-switch VS2 is programmed simultaneously. In summary, the sneak path problem arises when programming an atom switch in already indistinguishable vertical lines or indistinguishable horizontal signal lines.

From the opposite point of view, we could avoid the sneak path problem if we would program such an atom switch before multiple vertical/horizontal signal lines become indistinguishable. For example, in condition (a) in Fig. 6, we need to turn on the lower atom switch of via-switch VS1 before programming VS3 and VS4. It should be noted that, for programming a via-switch, we must turn on two atom switches, which are the lower atom switch connected to the vertical signal line and the upper atom switch connected to the horizontal signal line. The vertical signal line is used when programming the lower atom switch [e.g., STEP (2) in Fig. 4], and hence we need to pay attention to multiple on-state via-switches in the same horizontal signal line that connect multiple vertical signal lines. On the other hand, we should care about multiple on-state via-switches in the same vertical signal line when programming the upper atom switch.

Let us move to condition (b) in Fig. 6, where the number of bends of programming signal given to the signal line is one or zero. In this case, the programming signal that is provided to a control line is detoured and the sneak path problem arises. On the other hand, such a condition is satisfied only when a loop is intentionally programmed in a crossbar. For example, in Fig. 6(b), all the four via-switches are intended to be turned on, otherwise, the top two-atom switches of VS1 and VS3 never be on. This condition arises only when programming the last two-atom switches that compose a loop, and the ground voltage applied to the control line is propagated to the nontarget switch because of a loop structure.

From the above discussion, the sneak path problem cannot be avoided in the configurations that include a loop. Fortunately, such configurations with a loop are not used in practical applications since the looped signal routing increases the wire capacitance and degrades delay and power compared to nonloop routing. Therefore, we do not need to take care of condition (b). Consequently, what we have to consider for sneak path avoidance is only condition (a).

IV. Proposed Sneak Path Free Initial Programming

In this section, we propose a sneak path free initial programming method based on the discussion in Section III, where the initial programming means the programming that is performed for the crossbars whose via-switches are all off-state. Partial reconfiguration for the crossbars where some via-switches are on-state will be discussed in Section V. The proposed initial programming method gives a sneak path free programming order of via-switches, where the target configurations are nonlooped configurations for signal routing.

Section IV-A explains the overview of the proposed sneak path avoidance method followed by its details with examples in Section IV-B. Some key properties necessary for proving that a sneak path free programming order necessarily exists are in *italic*. With those properties, Section IV-C generalizes the proposed method and gives a proof that a sneak path free programming order necessarily exists for arbitrary nonlooped configurations. Section IV-D gives a pseudo code and execution examples of the proposed method.

A. Overview of Proposed Method

Table I shows the proposed method consisting of two steps: STEP 1 turning on all the upper atom switches of interest and STEP 2 turning on all the lower atom switches of interest. Each step is explained in the following.

In STEP 1, all the upper atom switches of the via-switches to be turned on in a given target configuration are programmed. The via-switch connects the vertical and horizontal signal lines only when both the upper and lower atom switches composing a via-switch are on-state. Therefore, in STEP 1, any signal lines are not connected to each other and the programming signal never detours. Hence, no sneak path problem arises in this step. Then, *arbitrary programming order works fine in STEP 1.*

STEP 2 turns on all the lower atom switches to be programmed. In STEP 2, a vertical line and a horizontal line are connected by a via-switch each time a lower atom switch is turned on since the corresponding upper atom switch is already turned on in STEP 1. Therefore, we have to determine the programming order of the lower switches paying attention to the occurrence condition of the sneak path problem, which is discussed in Section III. Please remind that we provide a programming signal to a vertical signal line when programming a lower atom switch, and hence we consider only multiple on-state via-switches in the same horizontal signal line since they make multiple vertical lines indistinguishable. Multiple on-state via-switches in the same vertical signal line do not matter. STEP 2 categorizes those multiple on-state switches in the same horizontal line as connector switches (CSs) and other switches as nonconnector switches (NCSs). STEPs 2a and 2b turn on NCSs and CSs, respectively.

Table I demonstrates that all the target switches are programmed by the proposed STEPs. The fifth column of Table I shows lemma numbers that correspond to each STEP. Proving those lemmas, we ensure that no sneak path problems arise in the proposed method.

TABLE I
Summary of Proposed Initial Programming Method

Prog. STEP	Upper atom switches	Lower atom switches		Relevant lemma
		NCSs	CSs	
Start				
STEP 1	Turned on			Lem. 1
STEP 2a	Programmed	Turned on		Lem. 2
STEP 2b	Programmed	Programmed	Turned on	Lem. 3-5
End	Programmed	Programmed	Programmed	

NCSs: non-connector switches, CSs: connector switches

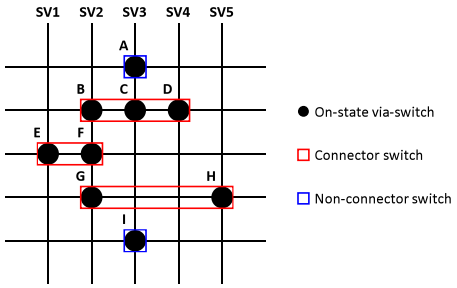


Fig. 7. Example of nonlooped configuration and definition of connector/NCSs.

It should be noted that the swapped sequence of STEP 2 followed by STEP 1, i.e., programming all the upper atom switches after turning on all the lower atom switches can also avoid the sneak path problem since the crossbar has a symmetrical structure. In this case, we need to determine the programming order of the upper switches.

B. Programming Order Determination With Connection Tree

This section details the proposed method explaining how to derive a sneak path free programming order of via-switches in a crossbar. In the following, we use a configuration of a 5×5 crossbar shown in Fig. 7 as an example.

As mentioned in the previous section, what we have to do is only determining the programming order in STEP 2 since STEP 1 does not cause sneak path problems with an arbitrary programming order. In STEP 2, we have to pay attention to multiple on-state via-switches in the same horizontal signal line that connect multiple vertical signal lines and make them indistinguishable. Therefore, we introduce two categories of the via-switch, namely, CSs and NCSs, which are defined in the previous section. For example, via-switches B, C, D, E, F, G, and H are CSs and via-switches A and I are NCSs in Fig. 7. A pair of CSs connects two vertical signal lines, e.g., via-switches E and F connect vertical signal lines SV1 and SV2 in Fig. 7. The nonconnector switch, on the other hand, does not connect any vertical signal lines.

Please remind that the sneak path problem occurs when turning on the lower atom switch included in the already connected vertical signal lines as explained in Section III. Therefore in STEP 2a, *all the NCSs should be programmed before the CSs* so that we can avoid the sneak path problem in programming the NCSs since the CSs which may connect vertical lines are still off-state. In this case, *the programming order of the NCSs is arbitrary.*

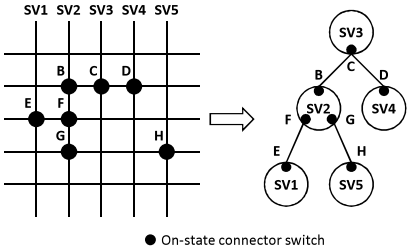


Fig. 8. Example of connection tree for CSs in Fig. 7.

Next, in STEP 2b, we determine the programming order of CSs. In this step, the programming order is not arbitrary since the sneak path problem may arise depending on the programming order. For example in Fig. 7, when we are turning on the CSs B or G after programming the CSs of E and F, the sneak path problem occurs since switches E and F have connected vertical lines SV1 and SV2.

To determine the programming order, we construct a connection tree that represents the connection status of vertical signal lines in a crossbar. Fig. 8 exemplifies a connection tree for the CSs in Fig. 7, where each node corresponds to a vertical signal line. The root node can be arbitrarily selected. Vertical line SV3 is selected as the root node in Fig. 8. When two vertical signal lines are supposed to be connected in the configuration of interest, we give an edge between the two nodes corresponding to these two vertical signal lines. Two black dots located at both ends of an edge represent CSs, and when both the two CSs are turned on, we suppose the edge is activated and two vertical lines are connected. From the definition, *any nonlooped configurations can be necessarily expressed by a tree structure*, which means loops are not included in the graph.

The connection tree tells us which connector switch can be programmed such that *the connector switch that can be turned on at the end of programming is indicated as the leaf node of the connection tree.* Let us explain what happens when we program the connector switch in the leaf node and nonleaf node of the connection tree at the last programming step, where the last programming step means that only one switch remains off and the others are already turned on in the target configuration.

In Fig. 9(a), the connector switch in the leaf node of SV1 is under programming, and the other CSs are already on-state. In this case, node SV1 and node SV2 are not connected yet, and hence the programming signal never propagates to any other vertical signal lines. Consequently, the target connector switch can be turned on without the sneak path problem. We can also confirm that there is no sneak path problem when programming the connector switch in the leaf node from Fig. 9(b), which is the circuit diagram corresponding to Fig. 9(a). Next, let us turn on the connector switch in nonleaf node SV2 in Fig. 9(c) at the last programming step. In this case, the programming signal reaches the other vertical signal lines through the CSs that are already on-state, and consequently the sneak path problem arises. The circuit diagram of Fig. 9(d) also indicates that atom switches placed at the same vertical position

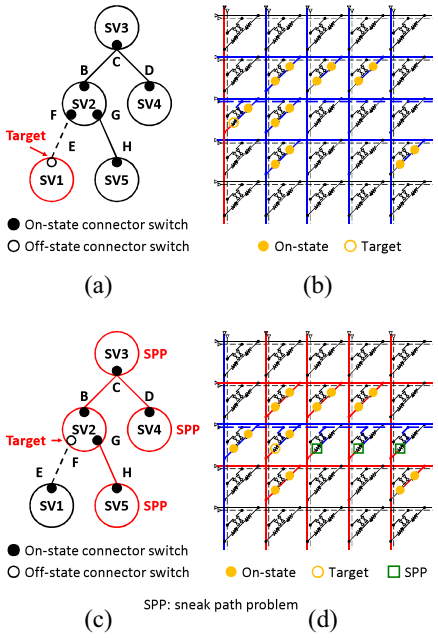


Fig. 9. Programming of connector switch in leaf/nonleaf node at the last programming step. (a) Programming of connector switch in leaf node. (b) Circuit diagram corresponding to (a). (c) Programming of connector switch in nonleaf node. (d) Circuit diagram corresponding to (c).

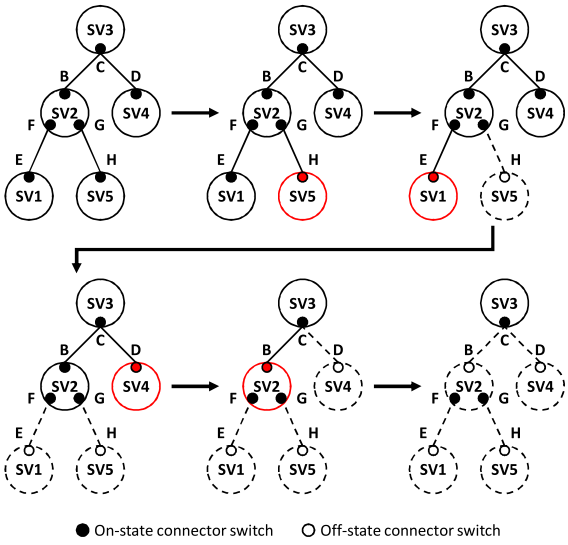


Fig. 10. Recursively searching switch which can be programmed lastly for each shrinking graph.

as the target on the connected indistinguishable vertical signal lines are unintentionally programmed.

Then, we propose to *recursively search a connector switch that can be turned on at the final programming step* for obtaining a sneak path free programming order of CSs. Fig. 10 illustrates the recursive process. Here, there are two types of CSs in each node, namely, the connector switch connecting with the parent node (e.g., switch B in node SV2) and the connector switch connecting with the child node (e.g., switches F and G in node SV2). *In each node, all the switches connecting with the child node must be turned on before the switch connecting with the parent node.* Otherwise, the sneak path problem arises when programming the switch connecting with

the child node since the programming signal is propagated to the parent node through the on-state switch to the parent node as pointed out in Fig. 9(c).

Let us roll back the recursive programming step one by one with Fig. 10. As we discussed, we can program only the connector switch in the leaf node at the final programming step. Then, switch H in SV5 is selected as the last switch to be programmed and node SV5 and the edge between SV2 and SV5 are deleted. This modified graph is again analyzed to find the next last switch to be programmed. In this case, switch E is selected. After SV5 and SV1 are removed from the graph, SV2 has no child nodes, and hence switch B in leaf node SV2 connecting with parent node SV3 can be programmed. In this way, one recursive process chooses one leaf node, identifies the switch in the leaf node connecting with the parent node as the last switch to be programmed in the current graph, and remove the leaf node and its edge to the parent node. Eventually, *all the edges are removed from the connection tree, and the recursive process finishes.* At this time, all the vertical lines are not connected to any other vertical lines anymore, and hence we can distinguish all the vertical lines. It should be noted that there remain some on-state CSs, for example switches C, F, and G in Fig. 10. *These switches can be programmed in an arbitrary order as long as they are programmed before the switches selected in the recursive processes.* One of the finally obtained programming orders is C, F, G, B, D, E, and H.

Depending on the target configuration, multiple connection trees may be constructed for a nonlooped configuration. Each tree has no connection to other trees, and hence the programming signal never propagates to other trees when programming the switch in the tree of interest. Consequently, we can handle each connection tree independently with the proposed method.

C. Proof of Existence of Sneak Path Free Programming Order

As indicated in Table I, the proof consists of five lemmas, and we prove them in the following, where Lemmas 1–5 demonstrate that there is no sneak path problem in the programming of upper atom switches, NCSs, and CSs, respectively.

Lemma 1: All the upper atom switches can be programmed in an arbitrary order without the sneak path problem.

Proof: Only when both the upper and lower atom switches composing a via-switch are on-state, the via-switch connects the vertical and horizontal signal lines. Hence, any signal lines are not connected to each other in this programming step because all the lower atom switches are still off-state. Therefore, the programming signal never detours and no sneak path problem occurs. From the same reason, the programming order of this step is arbitrary. ■

Lemma 2: There is no sneak path problem in the programming of all the NCSs, and arbitrary programming order works fine in this step.

Proof: In programming lower atom switches, the sneak path problem occurs when we turn on an atom switch in already indistinguishable vertical signal lines as mentioned in Section III. The indistinguishable signal lines originate from on-state CSs that connect multiple vertical lines. By programming all the NCSs before CSs, we can distinguish all the vertical lines in programming NCSs since all the CSs are still off-state in this step. Therefore, no sneak path problem arises and arbitrary programming order is acceptable in this programming step. ■

Lemma 3: Given a nonlooped configuration, it can be expressed by a connection tree or multiple connection trees.

Proof: When we treat each vertical signal line as a node and draw an edge between corresponding nodes if CSs exist, these nodes and edges compose a graph or multiple graphs that represent the connection status of vertical lines. Each graph does not contain closed loops unless the target configuration has loops. When a node is selected as the root node, the graph is expressed by a tree structure, which is called the connection tree defined in the previous section. ■

Lemma 4: At the last programming step for a connection tree, only a switch in a leaf node connecting to its parent node can be programmed without the sneak path problem.

Proof: When programming a switch in a leaf node at the final programming step, the target switch is still off-state and the connection between the leaf node and its parent node is not established yet. Hence, the programming signal given to the leaf node never propagates to any other nodes, and consequently there is no sneak path problem in this programming. On the other hand, a nonleaf node has at least two connections, i.e., connections to its parent node and child node. Therefore, the target nonleaf node has at least one connection to the other node at the last programming step since all the switches except the target switch are already on-state. Consequently, programming a switch in a nonleaf node at the end always causes the sneak path problem by propagating the programming signal to other nodes through the connection to the parent or child node. ■

Lemma 5: Recursively searching a switch which can be programmed at the last programming step always finds the sneak path free programming order.

Proof: By recursively searching a switch that can be turned on at the final programming step in the current graph and removing the leaf node and its edge to the parent node from the graph, all the nodes except the root node must be eventually eliminated and the recursive search necessarily finishes since the tree must have at least one leaf node when the number of nodes is two or larger. The remaining switches can be programmed in an arbitrary order before the switches chosen in the recursive search since each node has no connection to other nodes at this moment, i.e., all the vertical signal lines are distinguishable. ■

D. Pseudo Code and Execution Example

Algorithm 1 summarizes the overall determination procedure of a programming order. This algorithm determines a sneak path free programming order of nonconnector and CSs, and stores it to queue O_{prog}. Line 1 defines a set $\mathbb{S}$ of switches to be turned on. Lines 2–4 search NCSs by checking the number of on-state switches in each horizontal signal line. Specifically, line 2 creates a set $\mathbb{H}_j$ of on-state switches in jth

Algorithm 1 Programming Order Determination of Nonconnector and Connector Switches

1: $\mathbb{S} = \{S_{i,j} \mid S_{i,j}$ is on-state, $0 \le i < W,\ 0 \le j < H\}$
2: $\mathbb{H}_j = \{S_{i,j} \mid S_{i,j} \in \mathbb{S}\}$
3: $\mathbb{S}_{\text{non-con}} = \{S_{i,j} \mid |\mathbb{H}_j| = 1,\ S_{i,j} \in \mathbb{S}\}$
4: ENQUEUE($\mathbb{S}_{\text{non-con}}$) to O_{prog}
5: $\mathbb{S} = \mathbb{S} - \mathbb{S}_{\text{non-con}}$
6: **for** $i \in \{i \mid \exists S_{i,j} \in \mathbb{S}\}$ **do**
7: SEARCH($i,\ \mathbb{S},\ O_{\text{prog}},\ O_{\text{tmp}}$)
8: ENQUEUE(O_{tmp}) to O_{prog}

9: **function** SEARCH($i,\ \mathbb{S},\ O_{\text{prog}},\ O_{\text{tmp}}$)
10: $\mathbb{S}_{\text{child}} = \{S_{i,j} \mid S_{i,j} \in \mathbb{S}\}$
11: **if** $\mathbb{S}_{\text{child}} = \emptyset$ **then**
12: **return**
13: $\mathbb{S}_{\text{parent}} = \{S_{k,j} \mid k \ne i,\ \exists S_{i,j} \in \mathbb{S}_{\text{child}}\}$
14: ENQUEUE($\mathbb{S}_{\text{child}}$) to O_{prog}
15: ENQUEUE($\mathbb{S}_{\text{parent}}$) to O_{tmp}
16: $\mathbb{S} = \mathbb{S} - (\mathbb{S}_{\text{child}} \cup \mathbb{S}_{\text{parent}})$
17: **for** $k \in \{k \mid \exists S_{k,j} \in \mathbb{S}_{\text{parent}}\}$ **do**
18: SEARCH($k,\ \mathbb{S},\ O_{\text{prog}},\ O_{\text{tmp}}$)

W: crossbar width, H: crossbar height

horizontal signal line, line 3 enumerates NCSs in jth horizontal line where the number of elements of $\mathbb{H}_j$ is one, and line 4 enqueues all the NCSs to O_{prog}. Then, line 5 removes all the NCSs from the set $\mathbb{S}$, and subsequent lines 6–8 determine the programming order of connector switches. Line 6 selects ith vertical signal line, in which the connector switch to be turned on exists, as the root node of a connection tree, and the programming order of connector switches in this connection tree is determined by the function SEARCH in line 7.

SEARCH is a recursive function and traverses a connection tree from the root node to leaf nodes. Please remind that all the switches connected to child nodes need to be programmed before programming any switch connected to its parent node. SEARCH classifies the switches connected to child nodes of the parent node i (line 10) and the switches connected to the parent node i (line 13). The former is enqueued to O_{prog} in line 14 since switches connected to the child have to turn on before the switches connected to the parent. On the other hand, the latter is enqueued in O_{tmp} in line 15 until all the switches connected to the child node are enqueued to O_{prog} by the recursive function SEARCH. This function is recursively executed for all the child nodes of the node of interest (lines 17–18), and returns when the node of interest is a leaf node of the connection tree, i.e., no child nodes exist (lines 11 and 12). In the case that there exist multiple connection trees, $\mathbb{S}$ is not empty after line 7 completion. In this case, "for statement" in line 6 re-executes SEARCH with the updated $\mathbb{S}$ until $\mathbb{S}$ becomes empty. Finally, all the switches connected to the parent node in all nodes are enqueued to O_{prog} in line 8.

Let us explain an example when the proposed algorithm is applied to the configurations as shown in Fig. 7. Lines 2–4 find NCSs A and I, and enqueue them to O_{prog}. Line 7 determines

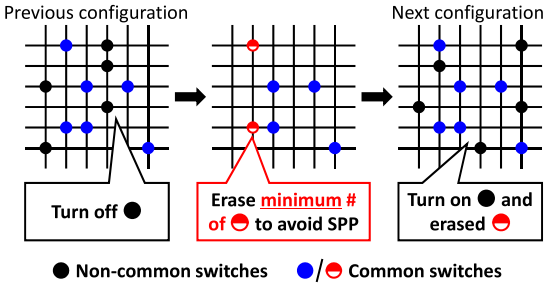


Fig. 11. Concept of partial reprogramming.

a programming order of the remaining connector switches B-H. Assuming the vertical line SV3 is selected as a root node i, at the first execution of the function SEARCH, the set $\mathbb{S}_{\text{child}}$ contains the switch C connected to the child node as shown in Fig. 8, and the set $\mathbb{S}_{\text{parent}}$ contains switches B and D connected to the parent node (root node). Line 14 enqueues the switch C to O_{prog} and line 15 stores switches B and D to O_{tmp}. After that, function SEARCH is executed again for vertical lines SV2 and SV4 where switches B and D exist. When SEARCH is executed for line SV2, set $\mathbb{S}_{\text{child}}$ contains switches F and G, and set $\mathbb{S}_{\text{parent}}$ contains switches E and H. On the other hand, when SEARCH is executed for line SV4, set $\mathbb{S}_{\text{child}}$ is empty and SEARCH returns. Eventually, we successfully obtain a sneak path free programming order and it is A, I, C, F, G, B, D, E, and H.

Thus far, we discussed the programming order for turning-on operations. Programming operations to turn on and off an atom switch are symmetric except that applied voltages to the signal line and control line are reversed. Therefore, we can turn off all the switches without the sneak path problem in the reverse order.

V. Proposed Partial Reprogramming Method

This section discusses partial reprogramming for changing crossbar configurations. Needless to say, we can change the crossbar configuration by turning off all the on-state switches in the previous configuration and then writing the next configuration to the crossbar with the proposed method explained in the previous section. However, this approach involves unnecessary reprogramming when both the previous and next configurations partially share the same on-state via-switches. The unnecessary reprogramming of via-switches directly leads to an increase in the reconfiguration time, and also shortens the lifetime of via-switches since the maximum number of reprogramming is limited [14]. For solving this problem, we propose a partial reconfiguration method that minimizes the number of programmed switches while avoiding the sneak path problem. This method contributes to extending the lifetime of via-switch FPGA and speeding up the reconfiguration.

Table II summarizes the proposed partial programming steps and indicates switches to be turned on/off in each step and their abbreviations in parentheses. Fig. 11 illustrates the basic strategy that programs noncommon switches and minimizes the number of common switches to be programmed for avoiding

TABLE II
SUMMARY OF PROPOSED PARTIAL RECONFIGURATION METHOD

Prog. STEP		Non-common in prev. config.		Common in both config.				Non-common in next config.			Relevant lemma
		Upper ASs (S_{PU})	Lower ASs (S_{PL})	Upper ASs (S_{CU})	Lower ASs			Upper ASs (S_{NU})	Lower ASs		
					Horizontal CSs (S_{CH})	Vertical CSs (S_{CV})	Others (S_{CO})		Horizontal CSs (S_{NC})	Horizontal NCSs (S_{NN})	
Start		Proged.	Proged.	Proged.	Proged.	Proged.	Proged.				
Partial erase (STEP 1)		Turn off	Turn off	Proged.	Proged.	Proged.	Proged.				Lem. 6
Partial write	STEP 2a			Proged.	Proged.	Turn off	Proged.				Lem. 7
	STEP 2b			Proged.	Proged.		Proged.	Turn on			Lem. 7
	STEP 3a			Proged.	Proged.		Proged.	Proged.			Lem. 8
	STEP 3b			Proged.	Turn off		Proged.	Proged.			Lem. 8
	STEP 3c			Proged.		Turn on	Proged.	Proged.		Turn on	Lem. 8
	STEP 3d			Proged.	Turn on	Proged.	Proged.	Proged.	Turn on	Proged.	Lem. 8
End				Proged.	Proged.	Proged.	Proged.	Proged.	Proged.	Proged.	

ASs: atom switches, CSs: connector switches, NCSs: non-connector switches, Proged.: programmed

the sneak path problem, where the proposed method erases only either upper or lower atom switch in such common switches depicted with red semicircle symbols in Fig. 11. Section V-A and V-B discuss how to partially turn off and on via-switches, respectively. The last column of Table II shows relevant lemmas to each programming step. By proving those lemmas, Section V-C gives a proof that the proposed partial reprogramming method can avoid the sneak path problem. Section V-D and V-E propose a minimization method of the number of programmed switches in the partial reprogramming. Section V-F gives a pseudo code of the proposed partial reprogramming method.

A. Partial Erasing

STEP 1 of partial erasing turns off both upper and lower atom switches of noncommon parts, which are S_{PU} and S_{PL}, respectively, in Table II. Similar to the case of turning on an atom switch, the sneak path problem arises if we turn off an atom switch in the already indistinguishable lines. Fig. 12 illustrates such a case. We can see that the harmful sneak path problem is not occurring since the detoured routing provides the turning-off voltage to the off-state switch. Fortunately, in nonlooped configurations, we can ensure that switches under unintentional programming by the sneak path problem are always off-state. If the switch under unintentional programming is on-state, a loop is composed by these switches beforehand. We use only nonlooped configurations, and therefore we can turn off any switch regardless of the programming order. Note that the programming order obtained in the previous section can erase all the switches without harmful or nonharmful sneak path problems.

B. Partial Writing

The partial writing process consists of two steps, where STEPs 2 and 3 turn on all the upper (S_{NU}) and lower (S_{NC} and S_{NN}) atom switches, respectively, of the via-switches to be newly turned on in the target configuration as shown in Table II. For avoiding the sneak path problem at each STEP, we turn off a part of common and already on-state switches S_{CH} and S_{CV} before programming S_{NU}, S_{NC}, and S_{NN} so that the programming signal does not detour to any nontarget switch. The following explains the details of each step.

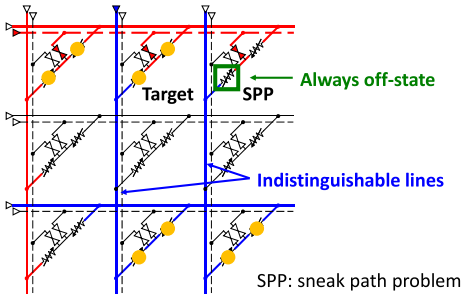


Fig. 12. Sneak path problem in erasing process.

STEP 2 is to turn on S_{NU} while avoiding the sneak path problem. As discussed in Section III, in programming upper atom switches, the sneak path problem arises when we turn on an atom switch in already indistinguishable horizontal lines. Therefore, we should care about S_{CV}, which represents the connector switches in the same vertical signal line. When S_{CV} exists in the same horizontal line of the target switch S_{NU}, we erase the lower atom switches S_{CV} in the same line of S_{NU} at STEP 2a. Such erasing is always possible as explained in the previous section. The erased S_{CV} will be reprogrammed later at STEP 3. After STEP 2a, STEP 2b can turn on S_{NU} without the sneak path problem since the programming signal is never propagated to any other horizontal lines. Thus, STEP 2 ensures that all the upper atom switches of via-switches to be turned on are on-state at the beginning of STEP 3.

STEP 3 turns on S_{NC} and S_{NN}, which are lower atom switches to be newly turned on in the target configurations. In addition, S_{CV} erased at STEP 2a is also turned on. The basic idea to avoid the sneak path problem in STEP 3 is as follows. First, we turn off a part of the connector switches and disconnect the node containing a target switch from the connection tree so that the programming signal is never propagated to other nodes. Then, we turn on the target switch in the isolated node, followed by turning on the erased connector switches to restore the connection between the isolated node and the connection tree.

STEP 3 is decomposed into the following four steps, which are listed in Table II. Let us explain each step with an example shown in Fig. 13. As explained in Section IV, the root node can be arbitrarily selected. Using this property, STEP 3a changes

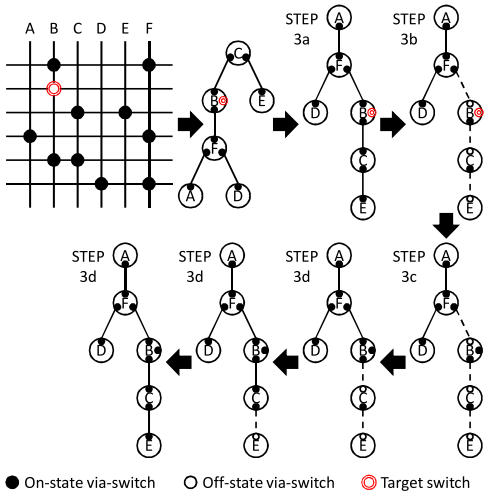


Fig. 13. Proposed partial writing method.

the root node of the connection tree for minimizing the number of switches to be programmed in STEPs 3b–3d, where the detail will be discussed in Section V-D. In Fig. 13, node A is selected as the root node. Next, STEP 3b is to turn off S_{CH}, which represents horizontal connector switches connecting to the parent node in the target and its descendant nodes. All the connections below the target node are disconnected from the connection tree by this step. The sneak path problem does not arise in STEP 3b since only erasing is performed. In Fig. 13, STEP 3b turns off the connector switches in node B, C, and E connecting to their parent nodes. In STEP 3c, we turn on switches S_{NC} and S_{CV}. There is no connection between the target node and the others thanks to STEP 3b, and therefore no sneak path problem arises in this step. Finally, STEP 3d reconnects the target and its descendant nodes to the connection tree. By turning on connector switches S_{CH} and S_{NC} in order from the shallow level to the deep level of the connection tree, we always turn on a connector switch in a leaf node in each phase, and hence there is no sneak path problem. As proved in Section IV, a connector switch in a leaf node of the connection tree can be turned on without the sneak path problem. In Fig. 13, STEP 3d turns on connector switches in order of node B, C, and E.

It should be noted that the swapped sequence of STEP 3 followed by STEP 2, i.e., programming upper atom switches after turning on lower atom switches is also acceptable since the crossbar has a symmetrical structure. In this case, we need to determine the programming order of the upper switches. Depending on the target configuration, the number of switches to be programmed is different for upper switch programming first or lower switch programming first. Therefore, we evaluate both the cases for reducing the number of programmed switches.

C. Proof of Sneak Path Avoidance in Partial Reconfiguration

This section summarizes a proof of sneak path problem avoidance with three lemmas that correspond to STEPs 1–3 in Table II.

Lemma 6 [Lemma for the Partial Erasing Process (STEP 1)]: The sneak path problem in erasing process always apply a voltage to turn off to off-state switches that are placed in the same position as the target switch in the indistinguishable lines.

Proof: When we turn off an atom switch in the indistinguishable lines, the sneak path problem arises in switches at the same position as the target switch in the indistinguishable lines. Assuming that these switches are on-state, these on-state switches and the target on-state switch connect already indistinguishable lines, i.e., the configuration has loops. This causes a contradiction since we program only nonlooped configurations. Therefore, these switches that are affected by the sneak path problem in the erasing process are always off-state. Applying a voltage to turn off an already off-state switch does not matter, and hence we can accept the sneak path problem in the erasing process. ■

Lemma 7 [Lemma for STEP 2 of the Partial Writing Process]: The sneak path problem in the turning-on process of upper atom switches can be avoided by pre-erasing lower atom switches of on-state vertical connector switch in the same horizontal line of the target switch.

Proof: As discussed in Section III, indistinguishable horizontal lines lead to the sneak path problem in the turning-on process of upper atom switches. The cause of indistinguishable horizontal lines is vertical connector switches that are multiple on-state via-switches in the same vertical line. If these on-state via-switches exist in the same horizontal line of the target switch, we turn off the lower atom switches of these via-switches before the target switch programming. In this case, we can distinguish the target horizontal line and the programming signal never detours to other horizontal lines. ■

Lemma 8 [Lemma for STEP 3 of the Partial Writing Process]: The sneak path problem does not arise in each STEPs 3a–3d.

Proof: STEPs 3a and 3b do not include turning on operations, and therefore the sneak path problem never occurs thanks to Lemma 6. In STEP 3c, we turn on atom switches in nodes that are isolated from the connection tree, and hence the programming signal is never propagated to other vertical lines and there is no sneak path problem. STEP 3d turns on connector switches to restore the connections of isolated nodes. When turning on connector switches in order from the shallow to the deep of the connection tree, each turning on operation becomes a leaf node programming. As proved in Lemma 4, the leaf node switch programming does not cause the sneak path problem, and hence there is no sneak path problem in STEP 3d. ■

D. Minimizing Number of Switches Programed in Partial Reconfiguration

This section proposes a minimization method of the number of switches programmed in partial reconfiguration. The total number of programmed switches can be calculated by summing the number of switches programmed in partial erasing (STEP 1) and writing (STEPs 2 and 3) process, which are

expressed in

$$\begin{aligned}&(\text{Total } \#SW_{prog}{}^{[*]} \text{ in partial reprogramming})\\&\quad = (\#SW_{prog} \text{ in STEPs 1, 2, and 3}),\\&\quad {}^{[*]}\#SW_{prog}\text{: \# of programmed switches.} \qquad (1)\\&(\#SW_{prog} \text{ in STEP 1}) = (\text{\# of } S_{PU} \text{ and } S_{PL}). \qquad (2)\\&(\#SW_{prog} \text{ in STEP 2}) = (\text{\# of } S_{NU})\\&\quad + (\text{\# of } S_{CV} \text{ in same horizontal line of } S_{NU}). \qquad (3)\\&(\#SW_{prog} \text{ in STEP 3}) = (\text{\# of } S_{CH} \text{ to be pre-erased}) \times 2\\&\quad + (\text{\# of } S_{CV} \text{ in same horizontal line of } S_{NU})\\&\quad + (\text{\# of } S_{NC} \text{ and } S_{NN}). \qquad (4)\end{aligned}$$

STEP 1 programs twice as many switches as the number of noncommon via-switches in the previous configuration since the via-switch is composed of two-atom switches and we have to turn off both switches S_{PU} and S_{PL}. STEP 2 turns on all the upper atom switches S_{NU} of the noncommon via-switches in the next configuration at STEP 2b. In addition, STEP 2a turns off the lower atom switch of on-state via-switches in the same horizontal line of S_{NU} if this on-state via-switch is a vertical connector switch S_{CV}. Therefore, the number of switches programmed in STEP 2 is given by (3). In STEP 3, the number of programmed switches can be calculated by (4). The pre-erased connector switches S_{CH} are programmed twice, i.e., turning off (pre-erasing) before the target programming at STEP 3b and turning on after the target programming at STEP 3d. The target switches S_{NC}, S_{NN}, and S_{CV}, where S_{CV}, which is turned off at STEP 2a, is programmed back at STEP 3c.

The number of S_{CH} to be pre-erased in (4) varies depending on the chosen root node of the connection tree since it changes the tree structure and the number of descendant nodes of the target node. On the other hand, (2), (3), and remaining terms of (4) are fixed for a given configuration. Hence, we minimize the number of pre-erased connector switches S_{CH}.

When we suppose that there is only one target switch, the number of connector switches S_{CH} to be turned off before the target programming can be given by

$$\begin{aligned}&(\text{\# of } S_{CH} \text{ to be pre-erased for one target switch})\\&\quad = (\text{\# of } S_{CH} \text{ to parent in target and its descendant nodes})\\&\quad - (\text{\# of } S_{CH} \text{ already turned off among first term}). \qquad (5)\end{aligned}$$

In the pre-erasing phase at STEP 3b, we turn off the connector switches connecting to the parent node in the target and its descendant nodes for disconnecting these nodes from the connection tree. There are already off-state connector switches since these switches are turned off in STEP 2a, and therefore the number of pre-erased S_{CH} is obtained by (5).

On the other hand, when there are multiple target switches in a connection tree, the total number of pre-erased connector switches is not necessarily the sum of (5) for each target switch. This is because some target switches could be programmed during the programming process of another target switch, where we regard the former switches are dominated by the latter switch. For example in the connection tree of Fig. 14, target switches a and f can be programmed in the programming process of target switch b when the root node is node E.

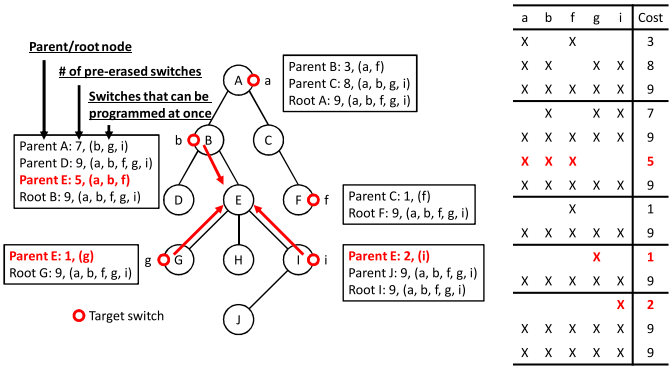


Fig. 14. Minimization method of the number of switch programming.

Here, switches a and f are dominated by switch b. In this case, STEP 3b disconnects target node B and its descendant nodes A, C, D, and F from the connection tree. Then, STEP 3c can turn on not only target switches b but also switches a and f since nodes A, B, and F are isolated, and the programming signal is never propagated to any other nodes.

Thanks to this dominance property, we can reduce the number of programmed switches compared to the case that we separately apply STEP 3 to each target switch. In the example of Fig. 14, the minimum number of programmed switches is achieved when we select node E or H as the root node and divide target switches into three groups, i.e., "switches a, b, and f," "switch g," and "switch i," where switches a and f are dominated by switch b. We separately apply STEP 3 to each switch group, and all the switches in the same group are programmed in the same process of STEP 3. The following paragraphs explain how to derive the optimal root node and the target switch groups.

We model this optimization problem as a set cover problem with cost minimization. For each target switch, we calculate the number of pre-erased connector switches and enumerate other target switches that are dominated by the target switch of interest while changing the root node. The number of pre-erased connector switches, which is the cost of this problem, can be given by (5). We can enumerate dominated target switches that can be programmed in the same process by checking whether other target switches are included in the target node of interest and its descendant nodes. Here, the dominance relation is determined once the parent node of the target switch is fixed. Therefore, the above enumeration for a target switch is repeated for the number of edges of the target node, not for the number of nodes in the connector tree. In the other case, the target node is the root node, and the dominance relation is also fixed. The above procedure gives some pairs of the number of pre-erased connector switches and the group of target switches that can be turned on in the same process of STEP 3. From the combination of these pair information, we find a set that covers all the target switches and minimizes the total number of pre-erased connector switches.

Fig. 14 exemplifies the set cover problem defined above. For target switch a, we count the number of pre-erased connector switches and the covered target switches in three cases; i.e., when node B is parent, when node C is parent, and when

node A is root. In case when node B is parent, we disconnect nodes A, C, and F in STEP 3b, and hence the number of pre-erased switches is 3 and the dominated switches are a and f. In the same way, we count the number of pre-erased switches and dominated switches for each target switch. Then, we construct the table in the right side of Fig. 14, where each row represents a group of a dominant node and dominated nodes and the corresponding cost of the number of pre-erased switches. From this table, we find a set of three red rows that covers all the target switches and minimizes the cost to 8. In this example, we can minimize the number of programmed switches so that node E or H is selected as the root node and the target switches are divided into three groups "switches a, b, and f," "switch g," and "switch i."

E. Root Node Selection With Lower Computational Complexity

The previous section demonstrates that we can minimize the number of programmed switches in partial reconfiguration by solving the set cover problem. However, the set cover problem is well known to be NP-hard. Thus, we propose an efficient root selection method to minimize the number of programmed switches without solving the set cover problem.

We exploit a property of the connection tree that at least one solution set of rows in the set cover problem can share the same root node. With this property, we only need to count the number of programmed switches sequentially supposing each node of the connection tree is the root node, and we choose the one with the minimum number of programmed switches. This approach reduces the computational complexity compared to solving the set cover problem since the number of nodes of the connection tree is the number of vertical signal lines at most. The proposed root selection method can be implemented as a polynomial-time algorithm and details will be explained in Section V-F.

The following paragraphs prove that the above property is always satisfied. For this proof, we introduce representative switches, which are defined as the target switches that are not dominated by other target switches in a solution of the set cover problem. One representative switch is included as one row in the table like Fig. 14, and this entry is obtained by assuming the parent node or the root node. Therefore, the direction to the root node is specified. For example in Fig. 14, the solution selects three representative switches b, g, and i, and three red arrows indicate the direction to the root node. In this case, node E or H is selected as the root node that satisfies all the red arrows at the same time. If such a root node exists in any case, there is no need to solve the set cover problem. In the following, we call the node that contains a representative switch as the representative node.

To derive the property that optimal representative switches can share the same root node, we prove that representative switches that have common dominated switch never compose an optimal solution first. We suppose that there are two cases of representative switches that cover all the target switches, and the one is with one or more common dominated switches and the other has no common dominated switch. For example, a pair of second and sixth rows of the table in Fig. 14 belongs to the former case, which has common dominated switch a. On the other hand, a pair of three red (6th, 10th, and 12th) rows corresponds to the latter case without common dominated switches. When we compare the number of pre-erased switches in both cases, the former is always costlier than the latter. This is because the node, including the dominated switch is disconnected and reconnected in each programming of a representative switch. The former disconnects/reconnects such a node multiple times, but the latter disconnects/reconnects it only once. Therefore, the optimal solution selects representative switches that have no common dominated switches.

Consequently, we can conclude that optimal representative switches are not dominated by other target switches. Hence, the original property can be proved by

$$\begin{aligned} &\mathrm{SW}_{\text{opt-rep}}{}^{[**]} \text{ are not dominated by other target switches.} \\ &\quad \Rightarrow \text{Multiple } \mathrm{SW}_{\text{opt-rep}} \text{ are not dominated by each other.} \\ &\quad \Leftrightarrow \text{Multiple } \mathrm{SW}_{\text{opt-rep}} \text{ are not descendants of each other.} \\ &\quad \Rightarrow \text{Multiple } \mathrm{SW}_{\text{opt-rep}} \text{ share a root node.} \\ &{}^{[**]}\mathrm{SW}_{\text{opt-rep}}\text{: optimal representative switches.} \end{aligned} \tag{6}$$

Another proof in an exhaustive manner for (6) is given in the Appendix.

F. Pseudo Code of Partial Reprogramming

This section gives a pseudo code of the proposed partial reprogramming method. Algorithm 2 includes only STEP 3 of the partial writing process since STEPs 1 and 2 are straightforward. Lines 3–8, line 9, lines 10–12, and line 13 correspond to STEP 3a, 3b, 3c, and 3d, respectively.

As explained in Section V-E, lines 3–8 calculate the number of programmed switches by function CALCCOST repeatedly changing the root node, and choose the one with the minimum number of programmed switches. Function CALCCOST recursively traverses the connection tree with depth-first search, and count the number of programmed switches. Inside this function, lines 15 and 16 search representative nodes from the closer nodes to the root node for each edge. After that, lines 17 and 18 enumerate on-state connector switches connecting to the parent in a representative node and its descendant for all representative nodes. We put these connector switches in $\mathbb{S}_{\text{erase}}$ and increment the cost, which is the number of programmed switches, by one each time we put. STEP 3a performs a depth-first search as many times as the number of nodes in the connection tree, and hence the worst time complexity is $O(|V|(|V| + |E|))$ where $|V|$ and $|E|$ are the number of nodes and edges of the connection tree, respectively. Here, $|E| = |V| - 1 < |V|$ holds in a tree structure, and $|V|$ is N_{vertical} at most, which is the number of vertical signal lines in a crossbar. Therefore, the worst time complexity can be written as $O({N_{\text{vertical}}}^2)$.

After the completion of lines 3–8, the switches to be pre-erased are in $\mathbb{S}_{\text{erase}}$, and we turn off them in line 9 as STEP 3b. Then, STEP 3c turns on target switches $\mathrm{S_{CV}}$ and $\mathrm{S_{NN}}$ in

Algorithm 2 Minimizing Programmed Switches

1: $\mathbb{S}_{\text{target}} = \{S_{\text{CV}},\ S_{\text{NC}},\ S_{\text{NN}}$ to be turned on$\}$
2: $\mathbb{S}_{\text{ch}} = \{$On-state $S_{\text{CH}}\}$
3: $\mathbb{S}_{\text{erase}} = \emptyset,\ \text{cost} = \infty$
4: **for** each node N_i **do**
5: $\quad \mathbb{S}_{\text{tmp}} = \emptyset,\ \text{cost}_{\text{tmp}} = 0$
6: $\quad$ CALCCOST($N_i,\ \emptyset,\ \mathbb{S}_{\text{tmp}},\ \text{cost}_{\text{tmp}},$ False)
7: $\quad$ **if** $\text{cost}_{\text{tmp}} < \text{cost}$ **then**
8: $\quad\quad \mathbb{S}_{\text{erase}} = \mathbb{S}_{\text{tmp}},\ \text{cost} = \text{cost}_{\text{tmp}}$
9: Turn off all $\mathbb{S}_{\text{erase}}$
10: **for** $S_i \in \mathbb{S}_{\text{target}}$ **do**
11: $\quad$ **if** S_i is not connector switch to parent **then**
12: $\quad\quad$ Turn on $S_i,\ \mathbb{S}_{\text{target}} = \mathbb{S}_{\text{target}} - \{S_i\}$
13: Turn on all $\mathbb{S}_{\text{erase}} \cup \mathbb{S}_{\text{target}}$ in order from shallow to deep
14: **function** CALCCOST($N_i,\ \mathbb{N}_{\text{rep}},\ \mathbb{S}_{\text{erase}},$ cost, flag)
15: $\quad$ **if** flag = False, N_i has $S_j \in \mathbb{S}_{\text{target}}$ **then**
16: $\quad\quad \mathbb{N}_{\text{rep}} = \mathbb{N}_{\text{rep}} \cup \{N_i\},$ flag = True
17: $\quad$ **if** flag = True, N_i has $S_j \in \mathbb{S}_{\text{ch}}$, S_j is connecting to parent **then**
18: $\quad\quad \mathbb{S}_{\text{erase}} = \mathbb{S}_{\text{erase}} \cup \{S_j\},\ \text{cost} = \text{cost} + 1$
19: $\quad$ **for** $N_{\text{child}} \in \{N_j \mid N_j$ is child node of $N_i\}$ **do**
20: $\quad\quad$ CALCCOST($N_{\text{child}},\ \mathbb{N}_{\text{rep}},\ \mathbb{S}_{\text{erase}},$ cost, flag)
21: $\quad$ **if** $N_i \in \mathbb{N}_{\text{rep}}$ **then**
22: $\quad\quad$ flag = False

lines 10–12. Finally, line 13 turns on connector switches S_{CH} and S_{NC} in order from shallow to deep at STEP 3d.

VI. Evaluation Results

This section discusses how many configurations are increased by the proposed sneak path problem aware programming method and demonstrates how much the number of programmed switches is reduced by the proposed partial reprogramming method.

A. Improvement on Number of Available Configurations

This section discusses an increase in the number of available configurations thanks to the proposed method. The conventional countermeasure for the sneak path problem, which is explained in Section II-C, imposes a programming constraint that prohibits a class of configurations of the via-switch FPGA. Therefore, the conventional countermeasure reduces the number of usable configurations and consequently diminishes the routing flexibility. On the other hand, the proposed method can give a sneak path free programming order for any nonlooped configurations. As discussed in Section III, the sneak path problem is unavoidable in looped configurations, but those configurations are practically meaningless for signal routing.

Fig. 15 compares the number of programmable configurations with the conventional countermeasure and the proposed method in 2×2, 3×3, 4×4, and 5×5 crossbars, where such small crossbars are used to enumerate all the nonlooped configurations. In this evaluation, the conventional countermeasure prohibits configurations in which multiple on-state via-switches exist in the same horizontal line [14]. We can see that the proposed method increases the number of usable configurations compared to the conventional countermeasure. Even in the small 5×5 crossbar, the number of available configurations increases by over two orders of magnitude. We also confirm that the number of usable configurations in the proposed method is equal to the number of all the nonlooped configurations. As proved in Section IV, the proposed method can find a sneak path free programming order for arbitrary nonlooped configurations in an arbitrarily sized crossbar. Another observation is that the increasing ratio of the number of usable configurations becomes larger as the crossbar size increments, which suggests a significant increase in the number of configurations in practically sized crossbars.

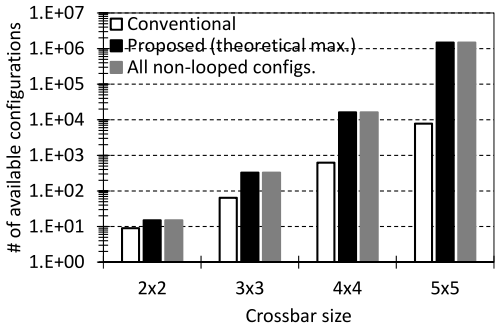


Fig. 15. Number of available configurations with conventional countermeasure and proposed method in small crossbars.

TABLE III
NUMBER OF USABLE CONFIGURATIONS AMONG 10 000 RANDOM CONFIGURATIONS IN A PRACTICALLY SIZED 100×100 CROSSBAR

% of on-state	# of samples	# of available configs.	
		Conventional	Proposed
0.1	10,000	6,347	10,000
0.2	10,000	1,324	10,000
0.3	10,000	91	10,000
0.4	10,000	1	10,000
0.5	10,000	0	10,000

Motivated by this, we assess the number of available configurations in a practically sized crossbar. Here, the total number of configurations exponentially increases as the crossbar size n becomes larger, like 2^n, and the comprehensive simulation of larger crossbars is infeasible. Instead, we generated random configurations for a large crossbar in Monte Carlo manner and compared the number of programmable configurations with conventional and proposed methods. The locations of on-state via-switches were determined by uniformly distributed random numbers. In this evaluation, the crossbar size was set to 100×100 and the number of trials was 10 000. We also varied the percentage of on-state via-switches from 0.1% to 0.5%. Table III shows the evaluation results. We can see that the proposed method achieves a significant increase in the number of usable configurations. When 0.5% of via-switches are on-state, the number of programmable configurations increases by over four orders of magnitude.

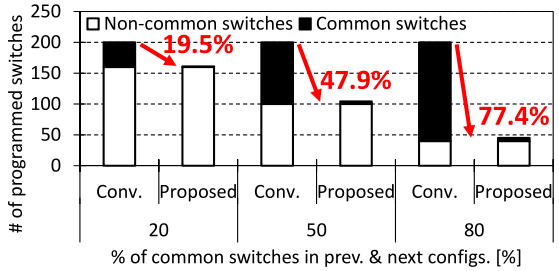


Fig. 16. Number of programmed switches in reconfiguration with conventional and proposed methods when 0.5% of via-switches are on-state in 100×100 crossbar.

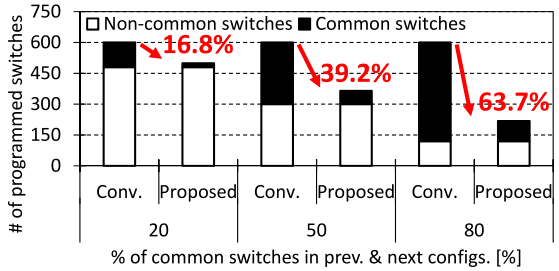


Fig. 17. Number of programmed switches in reconfiguration with conventional and proposed methods when 1.5% of via-switches are on-state in 100×100 crossbar.

B. Minimizing Number of Programmed Switches by Partial Reconfiguration

This section experimentally demonstrates that the proposed partial reconfiguration method can achieve a reduction in the number of programmed switches. Figs. 16 and 17 compare the number of programmed switches in the conventional method and that in the proposed method. Here, the conventional method turns off all the on-state via-switches in the previous configuration and then writes the next configuration to the crossbar. This evaluation randomly generates nonlooped previous and next configurations and derives a sneak path free programming order minimizing the programmed switches by the proposed method in Section V. We chose the locations of on-state switches using uniformly distributed random numbers. The crossbar size was set to 100×100, and the percentage of on-state via-switches in a crossbar was 0.5% in Fig. 16 and 1.5% in Fig. 17. We varied the percentage of common on-state via-switches in the previous and next configurations from 20% to 80%, and performed 10 000 trials for each case. Programmed common and noncommon switches are depicted with different colors.

From Fig. 16, we can see that the number of programmed switches in the conventional method is fixed to 200, which is 100 × 100 via-switches × 0.5% × 2 atom switches × 2 configurations (previous and next), for each case. In the proposed method, the number of programmed noncommon switches is the same as that of the conventional method since it is essential to erase and write noncommon switches for reconfiguration. On the other hand, the number of programmed common switches is dramatically reduced by the proposed method. As a result, the proposed method reduces the total number of programmed switches by 19.5% to 77.4%. The

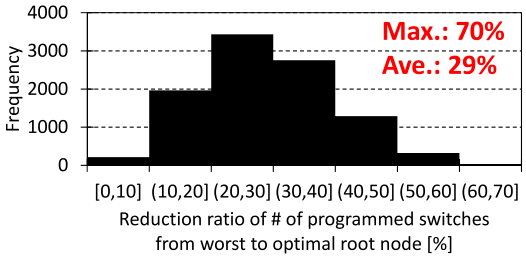


Fig. 18. Histogram of reduction ratio in number of programmed switches from worst root selection to optimal root selection.

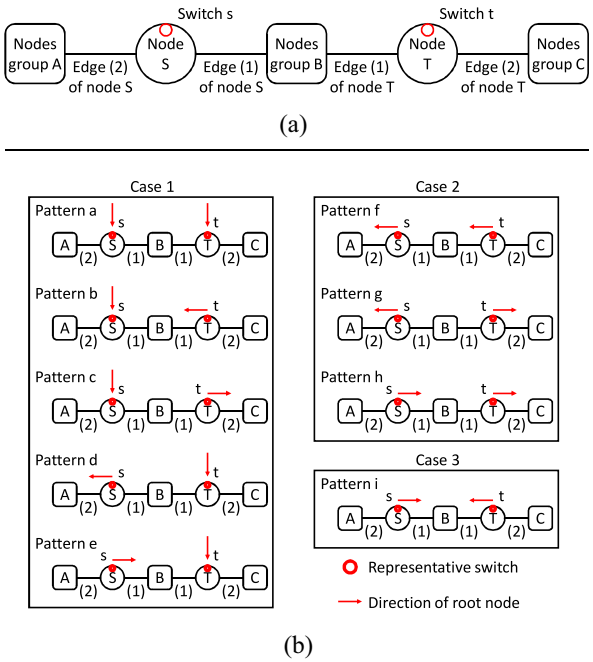


Fig. 19. All combinations of two representative switches.

reduction of 77.4% increases the number of possible reconfiguration executions of the via-switch FPGA by 4.4×. The reduction on the number of programmed switches also reduces reconfiguration time. If we share the programming drivers between the entire CLBs array for reducing the area of peripheral circuits, we need to program via-switches sequentially. In this case, the reconfiguration time is the programming time of each switch multiplied by the number of programmed via-switches. Therefore, 77.4% reduction on the number of programmed switches reduces reconfiguration time by 77.4%. On the other hand, we can also adopt a parallel programming scheme by increasing the number of drivers. Even in this case, the reconfiguration time decreases while the reduction ratio may vary depending on the maximum number of programmed switches in independent programming regions.

Comparing Figs. 16 and 17, the reduction ratio on the number of programmed switches decreases slightly as the percentage of on-state via-switches increases from 0.5% to 1.5%. This is because the density of on-state via-switches is relatively high and we have to erase more common switches for avoiding the sneak path problem. However, we can still see

that the proposed method considerably reduces the number of programmed switches.

Next, we discuss the impact of optimal root node selection in STEP 3a of partial writing. We compare the number of programmed switches in both cases that STEP 3a selects an optimal root node and a worst root node. Fig. 18 shows a histogram of reduction ratio in the number of programmed switches from the worst case to the optimal case. In this evaluation, the crossbar size is 100×100, the percentage of on-state via-switches in a crossbar of the previous and next configurations are 1% and 1.1%, all the on-state via-switches in the previous configuration are included in the next configuration, and consequently 0.1% via-switches are newly turned on in the next configuration. The number of trials is 10 000. The number of programmed switches can be reduced by 70% at maximum and 29% on average thanks to the optimal root selection. This result indicates that the optimal root selection plays an important role in the proposed method.

VII. Conclusion

This article has identified the programming status of the via-switch FPGA that cause the sneak path problem, and has proved that a via-switch programming order which can avoid the sneak path problem always exists for all the non-looped configurations. We proposed a sneak path avoidance method that gave sneak path free programming order of via-switches in a crossbar, and a minimization method of the number of programmed switches in the partial reconfiguration. In a test case, the proposed method reduced the number of programmed switches by 77.4% compared to the conventional approach, which enables $4.4\times$ more reconfigurations of the via-switch FPGA and reduces reconfiguration time by 77.4%. The proposed methods successfully solve the sneak path problem in any practical configurations and contribute to longer via-switch lifetime and faster configuration of the via-switch FPGA. Future works include the generalization of the proposed method for other crossbar structures with various type of switches beyond the via-switch FPGA.

Appendix

Another proof for (6) is presented. We separately give the proof for the two cases; when only one representative switch is given by the solution of the set cover problem, and when multiple representative switches are given.

When there is only one representative switch, the proof is self-evident. The direction to the parent node is specified only by one representative switch, and hence no contradiction occurs. Otherwise, the solution says that the representative node is the root node, and again no contradiction occurs.

Next, we discuss the proof for the case of multiple representative switches. We prove the case of two representative switches since this proof is easily extended to the cases that there are three or more representative switches. Fig. 19(a) illustrates the generalized tree structure with two representative switches, and Fig. 19(b) shows all the combinations of two representative switches. No matter how many edges a representative node has, we can categorize them into two groups: 1) an edge directed to another representative node and 2) other edges. All the nodes except representative nodes can be divided into node groups A, B, and C in Fig. 19(a). Node group B contains the nodes between the 1) edges of the two representative nodes. Nodes beyond the edges 2) of representative node S and T are categorized into node group A and C, respectively. In Fig. 19(b), each red arrow indicates the direction to the root node from each representative switch, and the red arrow pointing to the representative node itself indicates that the representative node is the root node. There are three directions for each red arrow [edge (1), edge (2), and the representative node itself], and hence there are $3 \times 3 = 9$ patterns of two red arrows. We divide these nine patterns into three cases. Case 1 contains patterns that include at least one red arrow pointing to the representative node itself. Patterns that have the red arrow indicating the direction of edge (2) are categorized into case 2. Case 3 contains patterns where both the red arrows indicate the direction of edge (1).

From now, we prove that the possible pattern is only case 3 in Fig. 19(b). In case 1, there is at least one red arrow pointing to the representative node itself, which indicates that choosing this representative node as the root node is optimal. To program the representative switch in the root node, we need to disconnect the descendant nodes of the root node, i.e., all the nodes of the connection tree. Therefore, another representative node is also disconnected and can be programmed in this programming step, which means there is a dominance relationship. From the definition, representative switches do not dominate each other, and hence the patterns in case 1 never exist as a solution of the set cover problem. In general, from the above reason, the red arrow pointing to the representative node itself never exists in cases where there are two or more representative nodes. Next, the same discussion is applied to case 2 where at least one red arrow indicates the direction of edge (2). In this case, another representative node is included in the descendant nodes of this representative node. For example in pattern f of Fig. 19(b), representative switch s argues that the root node is included in node group A. At that time, another representative switch t is one of the descendant nodes of node S, and then there is a dominance relationship. Hence, patterns in case 2 do not exist as a solution of the set cover problem. The same discussion is applicable to cases of three or more representative nodes. From the above discussion, the possible pattern of red arrows is only case 3 where all the red arrows indicate the direction to intermediate nodes of all representative nodes. We can satisfy all the red arrows by choosing the root node from intermediate nodes of representative switches, and can minimize the number of programmed switches without solving the set cover problem.

References

[1] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

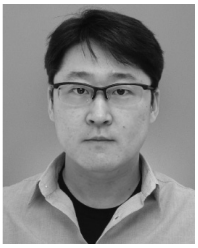
[2] M. Lin, A. E. Gamal, Y.-C. Lu, and S. Wong, "Performance benefits of monolithically stacked 3-D FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 216–229, Feb. 2007.

[3] P. E. Gaillardon *et al.*, "Design and architectural assessment of 3-D resistive memory technologies in FPGAs," *IEEE Trans. Nanotechnol.*, vol. 12, no. 1, pp. 40–50, Jan. 2013.
[4] S. Tanachutiwat, M. Liu, and W. Wang, "FPGA based on integration of CMOS and RRAM," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 11, pp. 2023–2032, Nov. 2011.
[5] J. Cong and B. Xiao, "FPGA-RPI: A novel FPGA architecture with RRAM-based programmable interconnects," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 4, pp. 864–877, Apr. 2014.
[6] X. Tang, P.-E. Gaillardon, and G. D. Micheli, "A high-performance low-power near-Vt RRAM-based FPGA," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Dec. 2014, pp. 207–214.
[7] Y. Y. Liauw, Z. Zhang, W. Kim, A. E. Gamal, and S. S. Wong, "Nonvolatile 3D-FPGA with monolithically stacked RRAM-based configuration memory," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, Feb. 2012, pp. 406–408.
[8] K. Okamoto *et al.*, "Conducting mechanism of atom switch with polymer solid-electrolyte," in *Proc. Int. Electron Devices Meeting (IEDM)*, Dec. 2011, pp. 1–4.
[9] M. Miyamura *et al.*, "Low-power programmable-logic cell arrays using nonvolatile complementary atom switch," in *Proc. 15th Int. Symp. Qual. Electron. Design (ISQED)*, Mar. 2014, pp. 330–334.
[10] N. Banno *et al.*, "A novel two-varistors (a-Si/SiN/a-Si) selected complementary atom switch (2V-1CAS) for nonvolatile crossbar switch with multiple fan-outs," in *Proc. IEEE Int. Electron Devices Meeting (IEDM)*, Dec. 2015, pp. 1–4.
[11] N. Banno *et al.*, "50x20 crossbar switch block (CSB) with two-varistors (a-Si/SiN/a-Si) selected complementary atom switch for a highly-dense reconfigurable logic," in *Proc. IEEE Int. Electron Devices Meeting (IEDM)*, Dec. 2016, pp. 1–4.
[12] N. Banno *et al.*, "Low-power crossbar switch with two-varistor selected complementary atom switch (2V-1CAS; via-switch) for nonvolatile FPGA," *IEEE Trans. Electron Devices*, vol. 66, no. 8, pp. 3331–3336, Aug. 2019.
[13] R. Doi, J. Yu, and M. Hashimoto, "Sneak path free reconfiguration of via-switch crossbars based FPGA," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.
[14] H. Ochi *et al.*, "Via-switch FPGA: Highly dense mixed-grained reconfigurable architecture with overlay via-switch crossbars," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 12, pp. 1–14, Dec. 2018.
[15] M. A. Zidan, H. A. H. Fahmy, M. M. Hussain, and K. N. Salama, "Memristor-based memory: The sneak paths problem and solutions," *Microelectron. J.*, vol. 44, no. 2, pp. 176–183, 2013.
[16] M. Zangeneh and A. Joshi, "Design and optimization of nonvolatile multibit 1T1R resistive RAM," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 8, pp. 1815–1828, Aug. 2014.
[17] M.-J. Lee *et al.*, "2-stack 1D-1R cross-point structure with oxide diodes as switch elements for high density resistance RAM applications," in *Proc. IEEE Int. Electron Devices Meeting (IEDM)*, Dec. 2007, pp. 771–774.
[18] C.-M. Jung, J.-M. Choi, and K.-S. Min, "Two-Step Write Scheme for Reducing Sneak-Path Leakage in Complementary Memristor Array," *IEEE Trans. Nanotechnol.*, vol. 11, no. 3, pp. 611–618, May 2012.
[19] S.-J. Ham, H.-S. Mo, and K.-S. Min, "Low-power $V_{DD}/3$ write scheme with inversion coding circuit for complementary memristor array," *IEEE Trans. Nanotechnol.*, vol. 12, no. 5, pp. 851–857, Sep. 2013.
[20] Y. Yang, J. Mathew, M. Ottavi, S. Pontarelli, and D. K. Pradhan, "Novel complementary resistive switch crossbar memory write and read schemes," *IEEE Trans. Nanotechnol.*, vol. 14, no. 2, pp. 346–357, Mar. 2015.
[21] P. O. Vontobel, W. Robinett, P. J. Kuekes, D. R. Stewart, J. Straznicky, and R. S. Williams, "Writing to and reading from a nano-scale crossbar memory based on memristors," *Nanotechnology*, vol. 20, no. 42, Sep. 2009, Art. no. 425204.
[22] M. A. Zidan, A. M. Eltawil, F. Kurdahi, H. A. H. Fahmy, and K. N. Salama, "Memristor multiport readout: A closed-form solution for sneak paths," *IEEE Trans. Nanotechnol.*, vol. 13, no. 2, pp. 274–282, Mar. 2014.
[23] Y. Deng *et al.*, "RRAM crossbar array with cell selection device: A device and circuit interaction study," *IEEE Trans. Electron Devices*, vol. 60, no. 2, pp. 719–726, Feb. 2013.
[24] S. Kim, J. Zhou, and W. D. Lu, "Crossbar RRAM arrays: Selector device requirements during write operation," *IEEE Trans. Electron Devices*, vol. 61, no. 8, pp. 2820–2826, Aug. 2014.
[25] I. Gupta, A. Serb, R. Berdan, A. Khiat, A. Regoutz, and T. Prodromakis, "A cell classifier for RRAM process development," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 62, no. 7, pp. 676–680, Jul. 2015.
[26] W. Banerjee *et al.*, "Investigation of retention behavior of TiO_x/Al_2O_3 resistive memory and its failure mechanism based on Meyer–Neldel rule," *IEEE Trans. Electron Devices*, vol. 65, no. 3, pp. 957–962, Mar. 2018.

Ryutaro Doi (Student Member, IEEE) received the B.E. and M.E. degrees in information systems engineering from Osaka University, Suita, Japan, in 2015 and 2017, respectively, where he is currently pursuing the doctoral degree with the Department of Information Systems Engineering.

His research interests include reconfigurable architecture and its test.

Mr. Doi is a Research Fellow of the Japan Society for the Promotion of Science.



Jaehoon Yu (Member, IEEE) received the B.E. degree in electrical and electronic engineering and the M.S. degree in communications and computer engineering from Kyoto University, Kyoto, Japan, in 2005 and 2007, respectively, and the Ph.D. degree in information systems engineering from Osaka University, Suita, Japan, in 2013.

He is currently an Assistant Professor with the Department of Information Systems Engineering, Osaka University.

Dr. Yu is a member of IEICE and IPSJ.

Masanori Hashimoto (Senior Member, IEEE) received the B.E., M.E., and Ph.D. degrees in communications and computer engineering from Kyoto University, Kyoto, Japan, in 1997, 1999, and 2001, respectively.

He is currently a Professor with the Department of Information Systems Engineering, Osaka University. His current research interests include design for manufacturability and reliability, timing and power integrity analysis, reconfigurable computing, soft error characterization, and low-power circuit design.

Prof. Hashimoto was on the technical program committees of international conferences, including DAC, ICCAD, ITC, Symposium on VLSI Circuits, ASP-DAC, and DATE. He serves/served as an Associate Editor for the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS I: REGULAR PAPERS, and the *ACM Transactions on Design Automation of Electronic Systems*.