Numberlink Problem Variants Modeled after FPGA Routing Fabrics and their Solvers that Enumerate all the Solutions

Ryohei Komi Hiroyuki Ochi

Graduate School of Information Science and Engineering
Ritsumeikan University
Ibaraki, Osaka 567-8570, Japan
{is0559xe@ed,ochi@cs}.ritsumei.ac.jp

Abstract— In this study, we define numberlink problem variants that mimic the routing fabrics of FPGAs, and develop solvers that enumerate all their solutions. The target FPGA architectures are early SRAM-based FPGAs and via-switch FPGAs. The existing method, which uses a top-down ZDD construction method (TdZdd), efficiently enumerates all solutions to the numberlink problem; however, it is specialized for planar grid-based routing problems. In this study, we extend the algorithm to multi-layer problems, targeting actual FPGAs where horizontal and vertical segments may overlap without intersections.

I. Introduction

Numberlink[1] is a logic puzzle that can be considered a model of the planar (single-layer) grid-based routing problem. Especially, numberlink problem has something in common with FPGA routing: a path must be completed using a combination of predefined wire segments.

A general routing algorithm can be used to find a solution to the numberlink problem[2]. On the other hand, a solver has also been proposed to find all solutions to numberlinks[3]. This solver uses a top-down/breadth-first decision diagram manipulation framework, TdZdd[4], to construct a ZDD that represents all solutions to the problem in a reasonable amount of time. Li et al. defined a multi-terminal numberlink problem that is more similar to the routing of actual integrated circuits. They proposed an algorithm using a top-down ZDD construction method to enumerate all solutions[5].

If we relate the numberlink problem to the routing problem of integrated circuits, we can say the following.

- When no solution exists, routing resources or flexibility are insufficient for the problem.
- When only a limited number of solutions exist, routing resources and flexibility are sufficient, but there is little margin.
- When there are many solutions, the routing resources and flexibility are excessive.

For this reason, the number of solutions (or routing alternatives) of numberlink is expected to be used to assess the routing resources and flexibility[5].

In this study, we define numberlink problem variants that mimic the routing fabrics of FPGAs, and develop solvers that enumerate all their solutions. The target FPGA architectures are early SRAM-based FPGAs and via-switch FPGAs. The proposed solver is an extension of the conventional solver [3] that is specialized for the original numberlink, a planar grid-based routing problem.

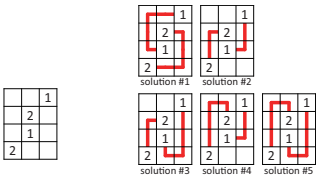
This paper is organized as follows. Section II provides an overview of a solver that uses TdZdd to enumerate all solutions to the numberlink problem, and describes the FPGA architecture we are targeting. Sections III and IV describe numberlink problems extended to via-switch FPGAs and early SRAM-based FPGAs, respectively, and propose dedicated solvers for them. In Section V, we present the experimental results. Finally, Section VI concludes our work and presents future challenges.

II. Preliminaries

A. Numberlink problem

Numberlink is a logic puzzle[1]. A problem instance consists of an $m \times n$ rectangle array of cells and one or more pair(s) of numbers (or labels) placed disjointly on the cells in the array. The solver must connect each pair of numbers with a continuous line. Each line (or path) consists of vertical or horizontal edges connecting the centers of cells. Paths for different pairs of numbers must not overlap or cross each other. Figure 1(a) shows an example problem of numberlink, and Fig. 1(b) shows the solutions.

Solutions #1 and #2 in Fig. 1(b) have very different routes and are considered representative solutions to this problem. On the other hand, solutions #3 to #5 are similar solutions that add a redundant U-shaped detour to the margin of solution #2. Ref. [3] also proposes a method to exclude U-shaped patterns from the numberlink solution.



(a) 4 × 3 problem. (b) All solutions.

Fig. 1. Problem instance of numberlink and its solutions.

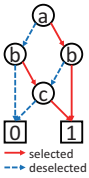


Fig. 2. ZDD representing {{a,b},{a,c},{b,c}}[3].

B. ZDD (Zero-suppressed Binary Decision Diagram)

A zero-suppressed binary decision diagram (ZDD) represents a set of combinations of elements[6]. A ZDD is a directed acyclic graph, as exemplified in Fig.2, which consists of a 0-terminal node and a 1-terminal node (squares), non-terminal nodes (circles), 1-arcs (solid arrows), and 0-arcs (dashed arrows). Each non-terminal node is labeled by an element and has one outgoing 1-arc and one outgoing 0-arc. There is only one node with no incoming edge, called the root node. Non-terminal nodes are ordered so that elements appear in a specific order along the paths from the root node to the terminal nodes. The 1-arc represents that the element of the source node is "selected," and a path from the root node to the 1-terminal node represents a combination of selected elements.

For example, the set of combinations of selecting two elements from {a,b,c} is {{a,b},{a,c},{b,c}}. This can be represented in ZDD as shown in Fig. 2.

ZDD excels at enumerating all solutions to combinatorial problems and can also efficiently narrow down the solutions by applying additional constraints.

C. Top-down ZDD Construction Method

Set operations, including union and intersection, can be performed efficiently on ZDDs, and a ZDD representing the desired set of combinations can be constructed by repeating basic operations on ZDDs. However, such a bottom-up ZDD construction method often runs out of memory space in the middle of computation since intermediate ZDD can be extremely large even when the final ZDD is compact.

The top-down ZDD construction method[3] constructs the ZDD for a target set of combinations directly from scratch. It first generates a root node, then its two children, and so forth in a breadth-first manner, level-by-level, from the root node to the terminal nodes. To allow breadth-first construction, each non-terminal node is given additional space for its computation state in addition to basic attributes of a ZDD node, such as level. Note that if two non-terminal nodes have the same level and computation state, these nodes can be uniquified since they will produce the same subgraphs. Edges that cannot reach a 1-terminal node are immediately pruned, i.e., connected to a 0-terminal node.

The top-down ZDD construction method has successfully generated ZDDs faster and reduced the memory usage compared to conventional bottom-up ZDD construction methods. The top-down ZDD construction method effectively solves combinatorial problems, especially when finding all the solutions. In the rest of this paper, we implement the algorithm using the C++ library TdZdd, which provides various functions for constructing and processing ZDDs using the top-down ZDD construction method. When using TdZdd, if we design a data structure for *mate*[] to hold the calculation state and define functions *getRoot*() and *getChild*() to generate the root and child node, respectively, we can construct a ZDD according to the top-down ZDD construction method.

D. Conventional TdZdd-based Numberlink Solver

Here we provide an overview of the conventional numberlink solver[3] using TdZdd. This algorithm has three main features.

Elements of combination that represent the solution A solution to a numberlink problem can be represented as a combination of edges that form paths. More notably, it is sufficient to use only the horizontal edges, not all the vertical and horizontal edges[3]. As a result, the number of elements of combination, that is, the number of ZDD variables, can be reduced to $m(n-1)$.

Number of elements of state variable *mate*[] When the cells are searched from top to bottom row and left to right within a row, the boundary (frontier) between the unexplored and explored cells spans only one row on the board. As a result, the state variable *mate*[] in each ZDD node needs to hold information for only n cells.

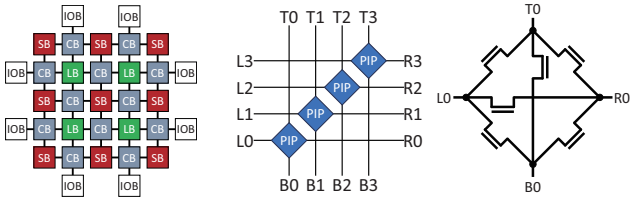
Compact encoding of *mate*[] $mate[j] (0 \le j < m)$ compactly represents the connection state of the jth column of the frontier row in one byte.

$$mate[j] = \begin{cases} j & \text{Unlabelled and unconnected} \\ j' & \text{An open end of a path to column } j' \\ n & \text{Additional edge is not allowed} \\ n+q & \text{Labelled } q \text{ but unconnected, or an} \\ & \text{open end of a path labelled } q \end{cases}$$

When searching the cell in the jth column, deciding whether to accept the right and bottom edges is necessary. During the process, it is necessary to prohibit (1) selecting too few edge (dead end), (2) selecting too many edges (branch), (3) connecting cells with paths to different labels (short circuit), and (4) reconnecting cells that have been connected (cycle). *mate*[] can provide enough information to make the right choice.

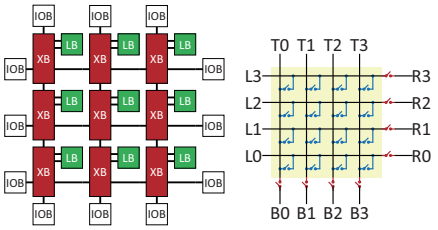
E. SRAM-based FPGA

SRAM-based FPGAs use SRAM for configuration memory, and are currently the mainstream FPGA[7]. As



(a) Top level architecture. (b) Disjoint-type SB. (c) PIP.

Fig. 3. Routing architecture of island style SRAM FPGA[7][8].



(a) Top level architecture. (b) XB.

Fig. 4. Routing architecture of via-switch FPGA[9].

shown in Fig. 3(a), the internal structure is typically an island style. IOB (input output block) interfaces with external devices. LB (logic block) performs operations. CB (connection block) bridges the LB's inputs/outputs and the wire segments. SB (switch block) connects the vertical and horizontal wire segments to form routing paths.

In the early SRAM-based FPGAs, the switches inside the SB were implemented using pass transistors. Figure 3(b) shows the disjoint type, one of the SB architectures adopted in Xilinx's XC4000 series FPGAs[8]. The diamond shapes at the intersections of the vertical and horizontal wire segments are PIPs (programmable interconnection points) that determine the routing path. The PIP comprised six pass transistors as shown in Figure 3(c).

F. *Via-switch FPGA*

Via-switch FPGAs[9] store configuration information in via switches. A via-switch is a non-volatile resistive-change switch element that can be programmed to ON or OFF. The overall FPGA architecture is similar to that of an island-style FPGA, as shown in Fig. 4(a), but XBs (crossbar blocks) are used instead of CBs and SBs. The XB is an array of via-switches, as shown in Fig. 4(b), and allows programming of the connections between vertical and horizontal wire segments. The XB has a very small footprint and can be implemented in the metal layer of an integrated circuit, so a via-switch FPGA with the same functionality as an SRAM-based FPGA has been realized with an area of only 8.3%[10]. As shown in Fig. 4(b), in addition to the switch array (blue) in the crossbar, there are switches (red) for programming the connections between the wire segments of adjacent XBs.

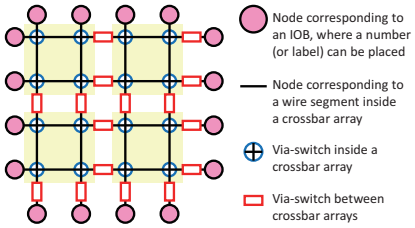


Fig. 5. Simplified model of via-switch FPGAs ($m = n = 4$, $m_{\mathrm{xb}} = n_{\mathrm{xb}} = 2$).

III. Numberlink Problem Modeled after Via-switch FPGA

In this section, we propose an algorithm based on the top-down ZDD construction method to enumerate all solutions to the numberlink problem for a "board" of simplified via-switch FPGA by extending the existing one for a general numberlink problem.

A. *Simplified model of via-switch FPGAs*

When designing a solver for a numberlink problem for a "board" of via-switch FPGA, it is first necessary to model the FPGA. In this paper, we focus on examining the feasibility of the algorithm, so we adopt a model that is as simple as possible. In via-switch FPGAs, blocks that determine the routing paths are XBs. This paper models the via-switch FPGA as an array of only IOBs and XBs, as shown in Fig. 5. We also assume that the labels for the numberlink problem can only be placed on the IOBs. In Fig. 5, the red circles are nodes that represent IOBs, the lines are nodes that represent wire segments inside the crossbar, the blue circles represent via-switches in the crossbar, and the red rectangles represent via-switches that connect wire segments of adjacent crossbars.

B. *Designing state variables for constructing ZDDs*

In the following, we design a data structure to solve the above numberlink problem as a combinatorial problem with all via-switches as elements. The total number of via switches is $mn + m_{\mathrm{xb}}n + mn_{\mathrm{xb}}$, where m and n are the total number of rows and columns, respectively, in the all crossbars, m_{xb} and n_{xb} are the number of XBs in the vertical and horizontal direction, respectively. Therefore, to enumerate all solutions to this problem using ZDD, we decided to use $mn + m_{\mathrm{xb}}n + mn_{\mathrm{xb}}$ ZDD variables. Our solver makes the binary decision (on or off) of via-switches one by one in the order from the upper rows (and from the left columns if in the same row).

In the algorithm for a general numberlink problem in [3], the state of one row to be searched is stored in the state variable $mate[]$, whose number of elements is n, where n is the number of columns. In the proposed algorithm for via-switch FPGA, the state of one row to be searched is stored in the state variable $mate[]$, whose number of elements is $n+1$. This is the sum of n, the number

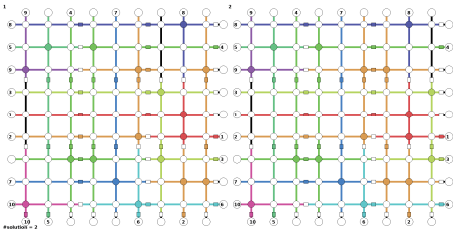


Fig. 6. Solutions to a sample problem with two solutions ($m = n = 9$, $m_{\mathrm{xb}} = n_{\mathrm{xb}} = 3$).

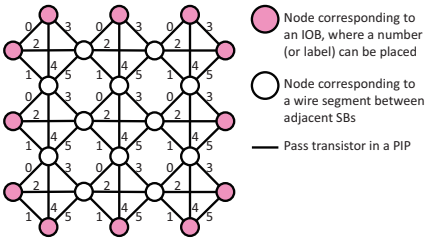


Fig. 7. Simplified model of early SRAM-based FPGAs ($m = n = 3$)

of vertical wire segments that cross the current row, and 1, the number of horizontal wire segment that cross the current cell. The value stored in $mate[j]$ is almost the same as in the algorithm of [3].

Now the algorithm was able to be implemented using TdZdd package without any problems. Figure 6 shows the solutions to a sample problem with two solutions.

IV. Numberlink Problem Modeled after SRAM-based FPGAs

Following the previous section, targeted at a simplified SRAM-based FPGA "board", we propose an algorithm to enumerate all solutions of the numberlink problem.

A. Simplified model of SRAM-based FPGAs

First, we introduce a simplified model of SRAM-based FPGAs. Among various blocks in SRAM-based FPGAs, the SB is the most essential block that determines the routing path. In this paper, we model the routing structure of an SRAM-based FPGA as an array of SBs only. To simplify the experiments, we assume that the number of tracks in the vertical and horizontal directions in the SBs is one. We model the SB configuration as a disjoint type. That is, each SB consists of only one PIP with six pass transistors. We also assume that the labels for the numberlink problem can only be placed on IOBs. By modeling an SRAM-based FPGA under these conditions, it looks like Fig. 7. In Fig. 7, the red circles are nodes that represent IOBs, the white circles are nodes that represent wire segments connecting adjacent PIPs, and the lines represent pass transistors that make up the PIPs. In an actual SRAM-based FPGA, there are wire segments between SBs (PIPs), but these are omitted in Fig. 7.

B. Designing state variables for constructing ZDDs

The above numberlink problem can be considered as a combination problem of $6mn$ pass transistors, where the array size is $m \times n$. To enumerate all the solutions to this problem using ZDD, we decided to use $6mn$ ZDD variables, each of which corresponds to a pass transistor. Our solver makes the binary decision (on or off) of pass transistors one by one in the order from the upper rows (and from the left columns if in the same row, and from the number 0 transistor in Fig. 7 if in the same PIP).

In the algorithm for a general numberlink problem in [3], the state of one row to be searched is stored in the state variable $mate[]$, whose number of elements is n, the number of PIPs in one row. In our algorithm for SRAM-based FPGAs, the state of one row to be searched is also stored in the state variable $mate[]$, but its number of elements is $7n + 2$. It consists of three parts: (1) intra-row connectivity ($3n + 1$ elements), (2) label information ($3n + 1$ elements), and (3) connectivity through rows above (n elements).

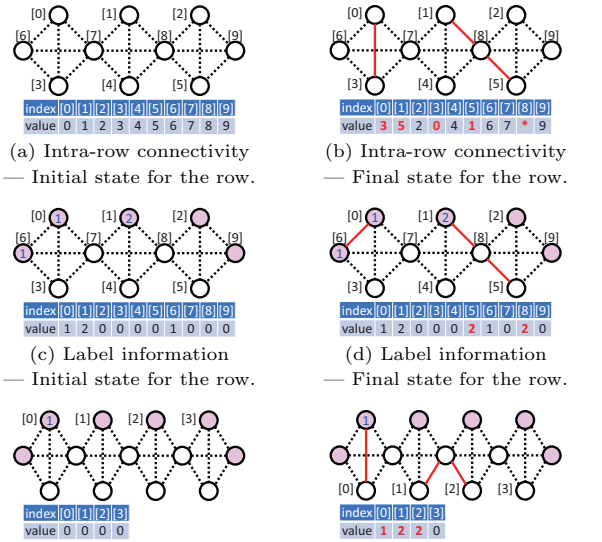
The connection information tracks the node number of the other end of a path within the same row. It is initialized with the node number of the node itself when the processing of the first node in a new row begins, as shown in Fig. 8(a). After that, each time a pass transistor is selected, the nodes at both ends of the path exchange numbers (Fig. 8(b)).

The label information is attached to a node when it is connected to another node with a label (a number given in the problem). The label information is initialized with that from the input file, as shown in Fig. 8(c), when the processing of the first node in a new row begins. After that, each time a pass transistor is selected, information is copied from the node with label information to the node without label information (Fig. 8(d)). When processing of a row is finished, the label information of the nodes at the boundary with the next row ([3] to [5] in Fig. 8(d)) is inherited by the corresponding nodes in the next row ([0] to [2] in Fig. 8(d)).

The connectivity through rows above indicates the presence of a connection to a node in the row above, or a connection via the row above, for the n nodes on the boundary with the row above. When processing of the top row of the "board" begins, there is no row above, so it is initialized to 0 (Fig, 8(e)). When processing of a row is finished, if the node on the boundary with the next row is an open end of a path, a unique number is assigned to each path and inherited by the next row (Fig.8(f)).

C. Constraints for state variables

When constructing a ZDD, combinations that violate the PIP constraints of the SRAM-based FPGA or the



(a) Intra-row connectivity — Initial state for the row.
(b) Intra-row connectivity — Final state for the row.
(c) Label information — Initial state for the row.
(d) Label information — Final state for the row.
(e) Connectivity thru rows above — Initial state for the row.
(f) Connectivity thru rows above — Initial state for the next row.

Fig. 8. State variable *mate*[] for early SRAM-based FPGAs.

(a) Legal pattern for path A-B. (b) Illegal pattern for path A-B.

Fig. 9. PIP constraints.

numberlink constraints must be pruned.

The PIP constraint is that selecting multiple pass transistors connected to one node of the same PIP is illegal. For example, Fig. 9(a) is a legal pattern that connects nodes A and B of a PIP, while Fig. 9(b) is illegal.

The constraints of numberlink are as follows: (1) Nodes with different labels are not mutually connected. (2) There is no route that does not reach label. (3) There is no node with label that does not have connection. (4) No route forms a cycle.

Violation of (1) can be avoided if a pass transistor between nodes with different label information is deselected. Violations of (2) and (3) can be pruned by detecting the open end of a path that does not reach a labelled node or the boundary node to the row below. Violation of (4) can be avoided if a pass transistor is deselected between nodes with connectivity through rows above.

The set of combinations that satisfy both constraints forms the solution set of the numberlink problem modeled after the SRAM-based FPGA. Figure 10 shows all solutions for an example problem satisfying both constraints.

V. Experimental Results

This section presents experimental results of the solvers proposed in Sections III and IV.

The programs for the proposed solvers were described in C++ using the C++ library TdZdd 1.1[4], and compiled using gcc-9.4.0 with option '-O3'. The CPU used in the experiments was an Intel(R) Xeon(R) Gold 6140 CPU@2.30GHz, with 128GB of main memory, and Ubuntu 20.04.6 as the OS.

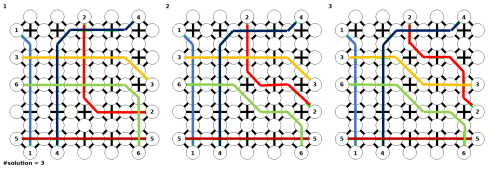
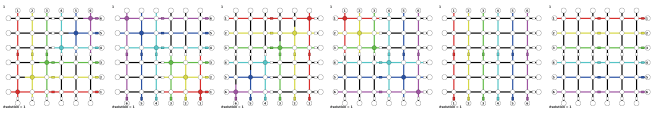
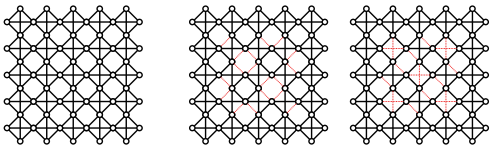


Fig. 10. Solutions to a sample problem with three solutions ($m = n = 5$).



(a) Q 1 (b) Q 2 (c) Q 3 (d) Q 4 (e) Q 5 (f) Q 6

Fig. 11. Test problems for solver for via-switch FPGAs.



(a) No depopulation (b) Diagonal only (c) Mixed

Fig. 12. SRAM-based FPGA model with depopulated transistors.

A. Results for the solver for SRAM-based FPGAs

The "board" size ($m \times n$) for the numberlink problem was set to 5×5 or 6×6. The number of labels was set to 6 pairs, and problems were generated by randomly placing them on the IOB nodes (Fig. 7). By generating problems randomly and discarding solutionless (unroutable) problems, we prepared 100 problems for each "board" size.

The experimental results are shown in Table I. The experimental results show that while the number of solutions increases by approximately 1200 times from a "board" size of 5×5 to 6×6, the increases in CPU time and memory usage are kept to about 100 times.

As a use case of the proposed method for architecture exploration, we investigate the change in the number of solutions when pass transistors are depopulated. Figures 12(b) and (c) both show PIP arrays with 18 pass transistors removed from Fig. 12(a). Table II shows the results of applying the same 100 problems as in the previous experiment to these. These results show that even when removing the same number of transistors, it is better to leave ones in the orthogonal directions. This can be considered a reaffirmation of the usefulness of long wires in the orthogonal directions (e.g., double-length wire).

TABLE I
RESULT OF SOLVER FOR SRAM-BASED FPGAS.

"board" size	#solution avg.	#solution min.	#solution max.	minimum total wire length avg.	minimum total wire length min.	minimum total wire length max.	CPU time (s) avg.	CPU time (s) min.	CPU time (s) max.	memory space (MB) avg.	memory space (MB) min.	memory space (MB) max.
5×5	47929	43	1197493	20.9	8	35	0.34	0.01	2.22	38.8	5	243
6×6	7.91E7	379	2.08E9	29.5	15	43	42.08	2.09	201.23	3557.5	166	22240

TABLE II
RESULTS FOR THE TRANSISTOR DEPOPULATION.

depopulation	routable problems	avg. #solution	avg. minimum total wire length
(a) No depopulation	100	47929	26.85
(b) Diagonal only	99	1264	28.16
(c) Mixed	66	978	28.85

TABLE III
MEMORY USAGE OF THE TEST PROBLEMS FOR VIA-SWITCH FPGAS.

Quiz	6×6	9×9	12×12	15×15
1	4	1653	—	—
2	43	—	—	—
3	5	880	—	—
4	4	8	6778	—
5	4	4	4	4
6	22	—	—	—

TABLE IV
IMPACT OF XB GRANURARITY IN VIA-SWITCH FPGAS.

min. $m_{\mathrm{xb}}, n_{\mathrm{xb}}$ with sol.	#prob		$m_{\mathrm{xb}}, n_{\mathrm{xb}}$ 1×1	2×2	3×3	6×6
1×1	477	avg. #sol	42.6	1215.4	10756.7	229753.6
		min. #sw	9.3	13.5	17.7	29.8
2×2	1455	avg. #sol	—	975.5	9948.5	241239.1
		min. #sw	—	15.7	19.2	30.9
3×3	65	avg. #sol	—	—	4378.6	123249.0
		min. #sw	—	—	22.2	34.3
6×6	3	avg. #sol	—	—	—	20007.0
		min. #sw	—	—	—	40.7

B. Results for the solver for via-switch FPGAs

As a test problem for numberlink for via-switch FPGAs, we used six problem patterns shown in Fig. 11. In these problems, n labels are placed in order on each of the two edges of a 'board' of size $n \times n$, and there is only one solution for each. Table III presents the relationship between the required memory capacity and the board sizes, ranging from 6×6 to 15×15. Due to the influence of the search order (variable order of the ZDD), we can see that the ZDD becomes more compact when there are more labels on the top edge.

We generated 2000 problems in which four pairs of labels were randomly arranged on a 'board' with $m = n = 6$. We investigated how the number of solutions (#sol) and the minimum number of switches (#sw) were affected when $m_{\mathrm{xb}} = n_{\mathrm{xb}} =$1, 2, 3, and 6. The results are shown in Table IV. Larger $m_{\mathrm{xb}}, n_{\mathrm{xb}}$ offer a larger number of problems with solutions and a larger number of alternative solutions, but require a larger number of switches. Interestingly, we also found a strong correlation between the number of solutions and the minimum number of switches (Fig. 13, for $m_{\mathrm{xb}} = n_{\mathrm{xb}} = 6$).

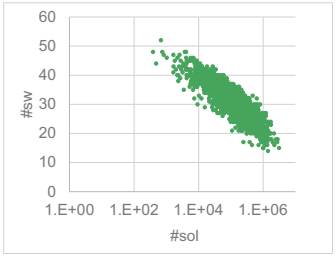


Fig. 13. Correlation between #sol and #sw

VI. CONCLUSION

In this paper, we modeled the routing architectures of via-switch FPGAs and early SRAM-based FPGAs as the numberlink problem. We developed a solver to determine the number of solutions to these numberlink problems and examined their applicability. There is still room for improvement in the selection of ZDD variables, the *mate*[] data structure, and the ZDD construction algorithm used in this paper. It is believed that this approach will apply to larger architecture in the future. The number of solutions to the numberlink problem is expected to be utilized as a new metric to evaluate the amount of resources and flexibility of a routing architecture.

REFERENCES

[1] Nikoli, "Nikoli official page". http://www.nikoli.co.jp/

[2] K. Kawamura, T. Shindo, T. Shibuya, H. Miwatari, and Y. Ohki, "Touch and cross router," ICCAD'90, pp.56–59, 1990.

[3] T. Toda, T. Saitoh, H. Iwashita, J. Kawahara, and S. Minato, "ZDDs and enumeration problems: state-of-the-art techniques and programming tool," Computer Software, vol.34, no.3, pp.3_97–3_120, 2017. in Japanese.

[4] H. Iwashita, "Tdzdd 1.1, a top-down/breadth-first decision diagram manipulation framework". https://github.com/kunisura/TdZdd

[5] X. Li, T. Imagawa, and H. Ochi, "Finding all solutions of multi-terminal numberlink problem utilizing top-down ZDD construction," ASICON 2023, pp.1–4, Oct. 2023.

[6] S. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," DAC'93, pp.272–277, 1993.

[7] A.Boutros and V.Betz, "FPGA Architecture: Principles and Progression," IEEE Circuits and Systems Magazine, vol.21, no.2, pp.4–29, 2021.

[8] H.-C. Hsieh, W.S. Carter, J. Ja, et al., "Third-generation architecture boosts speed and density of field-programmable gate arrays," CICC'90, pp.31.2/1–31.2/7, 1990.

[9] H. Ochi, K. Yamaguchi, T. Fujimoto, et al., "Via-switch FPGA: Highly dense mixed-grained reconfigurable architecture with overlay via-switch crossbars," IEEE T-VLSI, vol.26, no.12, pp.2723–2736, 2018.

[10] M. Hashimoto, X. Bai, N. Banno, et al., "Via-switch FPGA with transistor-free programmability enabling energy-efficient near-memory parallel computation," Jpn. J. Appl. Phys., vol.61, no.SM0804, pp.1–7, 2022.