Partial Connection-Aware Topology Synthesis for On-Chip Cascaded Crossbar Network

Minje Jun, *Member*, *IEEE*, Deumji Woo, and Eui-Young Chung, *Member*, *IEEE*

Abstract—The crossbar (also called bus matrix) solution is known as one of the most effective communication architectures for modern high-performance embedded systems. To make it even more effective, several topology synthesis methods have been proposed. They mostly generate a crossbar network in a cascaded fashion under the assumption that each crossbar switch is fully connected (i.e., each input has a connection to every output). This assumption often limits optimizing the area efficiency and/or performance of the network due to the unnecessary connections inside the crossbar switches. Some existing methods marginally improve their synthesis results by eliminating the unnecessary connections after the synthesis step. Such postprocessing approaches make sense since considering partially connected crossbar switches earlier in the synthesis flow can greatly increase the optimal topology search space, thereby increasing the runtime. However, the result from these postprocessing techniques is typically far inferior to that from the exhaustive search. In this work, we tackle such limitations of previous methods by introducing a heuristic method based on iterative switch merging. To the best of authors' knowledge, none of previous methods consider the partial connection of crossbar switches in the middle of the topology synthesis. Our experimental results prove the effectiveness of the proposed method by showing up to 30.35 percent of area saving against those methods that consider the partial connection only in a postprocess. The results also show the superiority of the proposed method against the existing topology synthesis methods, showing up to 49.09 percent area saving and synthesis time reduction by several orders of magnitude.

Index Terms—System-on-Chip (SoC), crossbar, topology synthesis, partial connection, on-chip interconnection network.

1 Introduction

The advance of process technology has leveraged the billion-transistor era, enabling huge integration of transistors into a single chip. In this era, System-on-Chip (SoC) is a viable solution to implement complex applications, even aiming at handheld devices. Many complex applications available today are mostly data-intensive; hence, the amount of communications among functional blocks critically determines the overall system performance. In other words, deciding which on-chip communication architecture to use is one of the most critical design steps affecting the design quality in terms of performance, area, and power consumption. The design complexity of an on-chip communication architecture also increases as the SoC design complexity increases. For this reason, both academia and industry have struggled for the design automation and performance improvements of on-chip communication architectures. The history of automatic communication architecture design is closely tied with the history of on-chip communication architecture itself, and we survey related work from both perspectives.

The shared bus design is a traditional solution for an on-chip communication architecture; hence, many previous approaches focused on improving the shared bus architecture itself, especially for its arbitration schemes. Thus, several design methods for arbitration schemes were proposed in [1], [2], [3]. However, the shared-bus architecture has become a performance bottleneck in many data-intensive SoC designs due to the large amount of on-chip data communication traffic. The multilayered bus architecture, an extended version of shared bus architecture, is to overcome the performance limitation of the shared-bus architecture. However, this multilayered bus architecture raised a new problem which was not considered in the single shared-bus architecture. More precisely, the topology of a network (i.e., how to shape a network) became a key parameter to determine the overall performance of a communication network. To deal with this problem, topology synthesis methods for the multilayered bus architecture were introduced to determine a network topology in an automated fashion [4]. However, the multilayered bus architecture inherently has a scalability issue which becomes worse as the number of components in a single chip increases. For this reason, the lifetime of these synthesis methods is heading toward the end since their target architecture is going to be used rarely in high-performance SoC designs shortly.

To overcome such limitations, the bus-matrix (also called crossbar)-based communication architecture has received a large attention to keep pace with the rapid increase of SoC design complexity. The crossbar architecture increases the overall communication bandwidth such that multiple master-slave pairs can communicate in parallel. Note that the crossbar architecture is not new and many previous

- *M. Jun and E.-Y. Chung are with the School of Electrical and Electronic Engineering, Yonsei University, 134 Sinchon-dong, Seodaemun-gu, Seoul 120-749, Korea. E-mail: minje.jun@dtl.yonsei.ac.kr, eychung@yonsei.ac.kr.*
- *D. Woo is with the Inter-University Semiconductor Research Center 306-4, School of Electrical Engineering and Computer Science, Seoul National University, San 56-1, Shillim-dong, Kwanak-ku, Seoul 151-744, Korea. E-mail: nanucu@capp.snu.ac.kr.*

Manuscript received 18 May 2009; revised 14 Feb. 2010; accepted 9 Sept. 2010; published online 15 Oct. 2010.
Recommended for acceptance by R. Marculescu.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-05-0211.
Digital Object Identifier no. 10.1109/TC.2010.211.

approaches can be found in [5], and it is known that the crossbar architecture generally outperforms the shared bus architecture [6], [7]. However, the adoption of the crossbar architecture in modern SoC design has recently been accelerated thanks to several companies who have productized crossbar-based solutions, such as AMBA Designer (from ARM) [8] and SonicsMX (from Sonics Inc.) [9]. Even though these vendors provide efficient crossbar switches with a system-level performance analysis tool, they did not mention the topology synthesis of crossbar networks in an automated fashion.

On the other hand, several research groups in academia focused on the topology synthesis of an on-chip crossbar network, which was rarely studied in the past. The aim of these approaches is to optimize important on-chip properties, such as area, power, and clock frequency, since the increase of parallel communications significantly sacrifices area, power, and clock frequency as the crossbar switch becomes larger (i.e., the number of master and slave ports increases). There have been several approaches to optimize the crossbar for the given communication characteristics of an application. One approach is to maximize the resource sharing by clustering masters and slaves into several local shared buses and connect them to a central crossbar so that the central crossbar can be minimized [10]. Another approach is to exploit the partial connection of a crossbar in addition to the clustering of masters and slaves [11]. The authors in [12], [13], and [14] improved the design further by considering other design parameters, such as an arbitration scheme, the sizes of buffers, memory allocation, and dynamic voltage and frequency scaling (DVFS). These approaches deal with the network topology problem, but they only aim at a single, central crossbar-based network, and therefore, they eventually face frequency limitations as the number of masters and slaves increases.

To address this issue, the cascaded crossbar network architecture and the corresponding topology synthesis methods were proposed to improve the scalability issue which is critical as the number of masters and slaves increases [15], [16], [17], [18]. In a cascaded crossbar network, masters and slaves can be connected to different crossbars, and the decentralized crossbars are connected to each other from the topological point of view. It was demonstrated that the cascaded crossbar solution can support larger systems than the single crossbar-based solution from the problem scalability perspective [18]. However, the existing topology synthesis methods for the cascaded crossbar network do not consider the partial connections of crossbars during the synthesis phase since the problem size becomes larger and the problem scalability becomes severer. To avoid the synthesis time explosion, the work in [18] considered the partial connections of crossbars at a postprocessing step after the topology of a subnetwork is determined. It obviously improves the design quality (such as area, delay, and power consumption) by eliminating unused paths inside crossbars, but such postprocessing does not allow the opportunity to appreciate better topological choices which may be explored if partial connection is considered in conjunction with the topology determination. Irregular topology synthesis methods in the Network-on-Chip (NoC) did not consider the partial connection, either [19], [20], [21].

In this paper, we tackle such major drawback of previous topology synthesis methods for the cascaded on-chip crossbar network. More precisely, we propose a topology synthesis method for crossbar-based on-chip communication network, where partially connected crossbar switches are considered during the topology determination process for maximizing its area efficiency and/or performance. Also, our method is based on a greedy heuristic technique called iterative switch merging which efficiently explores the optimal topology search space drastically enlarged by the consideration of partially connected crossbar switches. Hence, this feature prevents the runtime explosion issue which is very important as the design complexity of SoCs increases.

In Sections 2 and 3, we provide a motivating example of our method and the problem definition to be tackled in this work, respectively. In Section 4, we address the details of the proposed method. Finally, we demonstrate the experimental results to show the effectiveness of our work in Section 5 followed by a conclusion in Section 6.

2 Motivation

In most of previous methods [15], [16], a fully connected crossbar switch is used as a basic unit of a communication resource for building a cascaded crossbar network. However, these approaches waste the area by keeping unnecessary connections inside the crossbar switches.

In [18], the authors proposed to eliminate unnecessary connections in each crossbar switch at a postprocessing step, but it only shows marginal improvement in area efficiency since the topology itself is not altered. For this reason, it is necessary to consider the partially connected crossbar switches during synthesis phase to achieve higher area efficiency.

Fig. 1 clearly compares the area efficiency of the aforementioned three approaches. In this motivating example, we used one of our test cases under 90 nm process technology. Also, the clock frequency constraint was set by the maximum bandwidth on a link times the channel width. Fig. 1a is the topology for the test case synthesized by fully connected crossbar based topology synthesis. Fig. 1b is the topology synthesis result for the same example when we additionally consider the postprocessing step for partial connection. Since the postprocessing eliminates the unnecessary inner connections of each crossbar, the area is surely reduced, but the topology is unchanged. From Figs. 1a and 1b, it is shown that the postprocessing for partial connection improves the area efficiency by 20.53 percent. Finally, by considering the partial connection of crossbars simultaneously when determining the topology (we call it ***in-process partial crossbar***), we can find a better solution as shown in Fig. 1c which was obtained by our method to be discussed later in detail. In this case, the area efficiency compared to Fig. 1a is improved by 44.84 percent which is about two times than that achieved in Fig. 1b.

This motivating example clearly shows why we need to consider ***in-process partial crossbar*** which is even capable of alternating the topology of the network for better area efficiency. However, ***in-process partial crossbar*** obviously expands the design space, and thus, increases the synthesis time. Therefore, we need to find a runtime-efficient synthesis algorithm to search the enlarged search space.

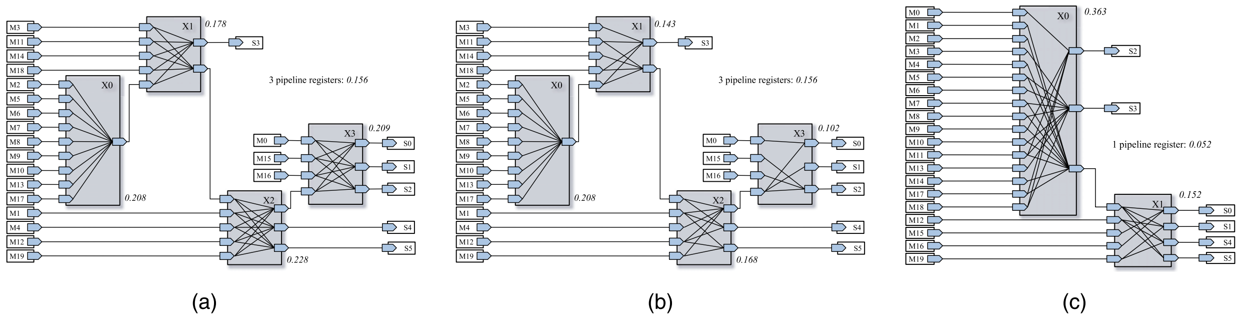

Fig. 1. The advantage of in-process consideration of partial connection of crossbar. ("M" : master, "S" : slave, "X" : crossbar.) (a) Fully connected crossbar, Area: $0.979(mm^2)$. (b) Postprocess consideration of partial crossbar, Area: $0.778(mm^2)$. (c) In-process consideration of partial crossbar, Area: $0.540(mm^2)$.

In the sequel, we call the topology synthesis with only fully connected crossbars ***no-partial crossbar*** for brevity. Similarly, we call the topology synthesis which considers partial connections at a postprocessing step ***postprocess partial crossbar*** in contrast to ***in-process partial crossbar***. We formally define the problem of ***in-process partial crossbar*** in Section 3 and describe the solution that we propose in Section 4.

3 Problem Definition

3.1 Definitions

We propose an application-specific on-chip network topology synthesis method. It takes a communication trace graph as input and delivers as output a synthesized topology satisfying given constraints.

Definition 1. *Communication Trace Graph $CTG = G(V_M, V_S, E)$ is a directed graph, where $v_m \in V_M$ and $v_s \in V_S$ are, respectively, a master node and a slave node, and $e_{m,s} \in E$ is the edge between v_m and v_s. The edge $e_{m,s}$ has two attributes, $w(e_{m,s})$ and $d(e_{m,s})$, representing the required bandwidth and the latency constraints of the edge at the hop-count level, respectively.*

In shared memory architectures, the memories are mapped to slave nodes while the initiators, such as CPUs and DMAs, are mapped to master nodes. In peer-to-peer communication architectures, each processing element can be mapped to either master node, slave node, or both depending on whether it is a producer, consumer, or both of the communication.

Definition 2. *Topology $T(X, L)$ represents how cores and crossbars are connected, where $x \in X$ denotes a crossbar and each $l \in L$ a physical link connecting a core and a crossbar or a crossbar and another crossbar. A link $l \in L$ is called* loose *if the latency constraints of the edges which are assigned to the link are greater than 1.*[1] *Otherwise, the link is called* tight. *C_k denotes the cost (such as area and power consumption, or weighted sum of them) of crossbar $x_k \in X$.*

Definition 3. *M is a 2D matrix in which each element $m_{i,j}$ is the merge cost, namely, the design-cost reduction when we merge x_i and x_j. Suppose that we merge x_i with x_j. $m_{i,j}$ is positive if the merging of x_i and x_j reduces the design cost (i.e., improves the design) and negative if it increases the design cost. Also, $m_{i,j}$ is a real value and its magnitude corresponds to the amount of cost reduction or penalty depending on its polarity. M is naturally a symmetric diagonal matrix (i.e., $m_{i,j} = m_{j,i}$ and $m_{i,i} = 0$) since merging x_i to x_j is same to merging x_j to x_i from the topological point of view. Also, the diagonal elements of M are all zeros since merging a crossbar to itself does not make sense. Therefore, we only compute the elements in the upper half of M. The detail of computing $m_{i,j}$ will be introduced in Section 4.3.2.*

1. Its physical meaning is that one or more crossbars can be inserted between the two terminals of the link.

3.2 Problem Definition

In this paper, we concern the problem of synthesizing the topology $T(X, L)$ that satisfies all the conditions given in the CTG and that yields the best cost. The cost can be the clock frequency, the area, or the power consumption of the network. The design space includes the partial connection of each crossbar.

To avoid any potential deadlock and to focus on the network topology synthesis itself, we assume 1) a single path between a master and its slave and 2) a single protocol, a single channel width, and a single clock frequency of the network. That is, the bridges for protocols, addresses/data widths, and clock frequencies are not considered in the synthesis process, and the latency is defined using the hop-count concept. Lastly, we assume that pipeline stages are inserted only between the crossbars. These assumptions are valid in most practical cases due to the following reasons:

- The first assumption: In the application-specific on-chip network topology synthesis, it has been demonstrated that multiple paths marginally improve design quality improvements for the 10 benchmarks used in [22], even when the overhead of deadlock-free routing and in-order packet delivery was not considered. It is also notable that the results are obtained with considering only the fully connected switching components. In our approach, every unused path is removed from switches, and therefore, the chance of the multipath routing being beneficial is even more reduced.
- The second assumption: In this study, we are focusing on the backbone on-chip network into which hundreds of IP blocks are integrated. In such a large system, an IP can be a local system that communicates with other IPs through the backbone network interface, which is in charge of the compatibility of the protocol, address/data width, and

frequency used. Therefore, we assume that the roles of bridges are played by the network interfaces, which are not included in the backbone network.

- The third assumption: We assume that the pipeline stage can be inserted only between two crossbars, just as in [15], [16], [18]. That is, we do not consider modifying internal pipeline configuration of crossbars, and concentrate on finding an optimal topology composed of the given crossbars.

Under the above assumptions, the synthesis problem we try to address can be defined as follows:

Given CTG $G(V_M, V_S, E)$ and the crossbar library, **find** the topology $T(X, L)$ that minimizes the defined cost **such that** the bandwidth and latency constraints are satisfied.

4 Proposed Topology Synthesis

4.1 Overview of the Proposed Method

The proposed topology synthesis flow is shown in Fig. 2 and an example of applying this flow is given in Fig. 3. The synthesis process consists of two phases: 1) initial topology generation, and 2) iterative crossbar merging.

The first phase is to generate the topology composed of crossbars that are as small as possible, while satisfying the constraints. This phase starts with allocating a dedicated crossbar to each master node and slave node and then establishes links according to the connection information given in the input CTG (Fig. 3a). Next, new crossbars are iteratively inserted to the links while maintaining the latency and bandwidth constraints (Fig. 3b).

In the second phase, the crossbars generated from the first phase are iteratively merged until no more merge can improve the quality of the solution. For each iteration, the crossbar pair that can reduce cost the most is merged, and according to the demanded connections in the CTG, unnecessary connections in each crossbar are removed, while considering the partial connections for each crossbar (Figs. 3c, 3d, and 3e). Unlike our approach, the technique by [18] considers applying partial connections only after the topology (or subtopology) is determined. In order to find

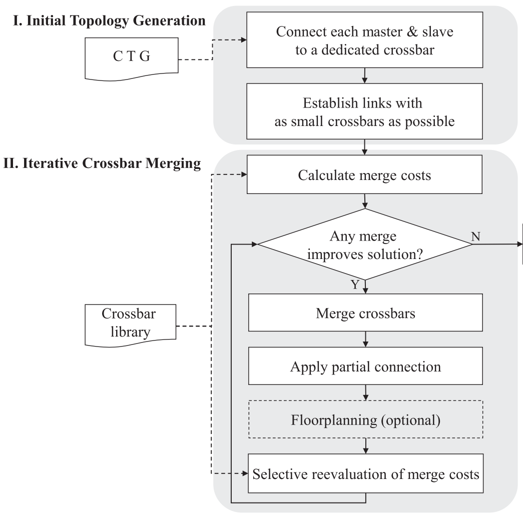
I. Initial Topology Generation
C T G
Connect each master & slave to a dedicated crossbar
Establish links with as small crossbars as possible
II. Iterative Crossbar Merging
Calculate merge costs
Any merge improves solution?
N
Y
Best solution
Crossbar library
Merge crossbars
Apply partial connection
Floorplanning (optional)
Selective reevaluation of merge costs

Fig. 2. Proposed topology synthesis flow.

the most cost-reducing crossbar pair, it is needed to obtain the merge cost of every existing crossbar pair for each iteration. It is obviously too inefficient to recalculate the merge cost of every crossbar pair whenever a merge occurs. Thus, we reevaluate only those crossbar pairs that have been affected by a merge. More details on this technique will be presented in Section 4.3.

In our crossbar network design, we adopt the floorplan consideration methodology where the floorplanner is invoked whenever the topology is updated. However, since the coconsideration of the floorplan and topology synthesis is not our main contribution, we take the floorplanning as an optional step in order to concentrate on the network topology synthesis.

4.2 Initial Topology Generation

The purpose of this phase is to obtain a topology with the smallest crossbars while satisfying all the constraints given in the input CTG, to prepare the iterative merging phases.

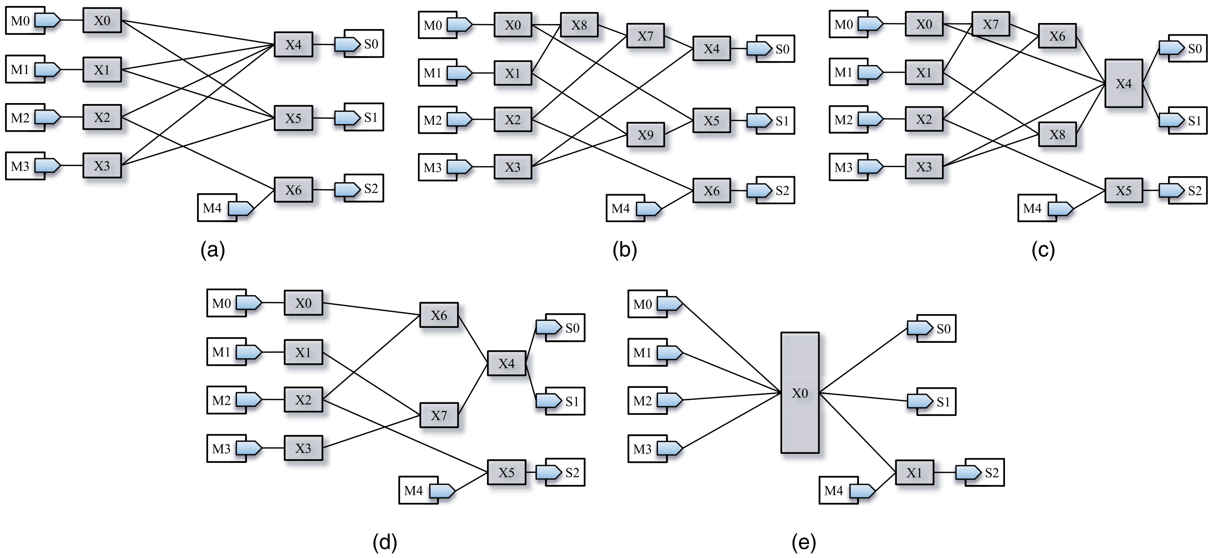

Fig. 3. Example of topology synthesis process. (“M” : master, “S” : slave, “X” : crossbar.) (a) Connect each master and slave to a dedicated crossbar. (b) Establish links with as small crossbar as possible. (c) Merge crossbars. (d) Remove redundant links. (e) Iterate crossbar merging and redundant link removal.

```
Algorithm 1 : Generate Initial Topology
Input: CTG = ($V_M, V_S, E$)
Output: initial topology $T$
 1:  for all $v_m \in V_M$ do
 2:    link $v_m$ with a crossbar $x_m$;
 3:  end for
 4:  for all $v_s \in V_S$ do
 5:    link $v_s$ with a crossbar $x_s$;
 6:  end for
 7:  for all $e_{m,s} \in E$ do
 8:    if $d(e_{m,s}) > 1$ do
 9:      link crossbar $x_m$ with crossbar $x_s$;
10:    else
11:      merge crossbar $x_m$ with crossbar $x_s$;
12:    end if
13:  end for
14:  for all $x_i \in X$ do
15:    if $x_i$ has more than 2 loose links do
16:      partition the loose links into two groups;
17:      assign new crossbars $x_j$, $x_k$ to each group;
18:      $X = X \cup \{x_j, x_k\}$;
19:    end if
20:  end for
21:  return $T$;
```

Fig. 4. Algorithm for initial topology generation.

The algorithm for this phase is shown in Fig. 4. This phase first initializes the network by connecting each master and slave to a dedicated crossbar (line 1-6), and making the links between crossbars to establish the connections given by the CTG (line 7-13). In this step, if $d(e_{m,s})$ is 1 (i.e., only one hop is allowed between master v_m and slave v_s), we merge the two crossbars to which the master v_m and slave v_s are connected so as to meet the latency constraint. Next, in lines 14-20, the crossbars are split into smaller ones iteratively. The process starts from selecting a crossbar which has more than two loose links and splits the crossbars into smaller ones by the following procedure:

- Step 1: At the selected crossbar node x_i, partition the loose links into two groups. If x_i has one or more tight links as well as loose links, all the loose links are partitioned into the same group.
- Step 2: Each group is assigned a dedicated crossbar so that x_i now becomes a 1×2 (or 2×1) crossbar if x_i has no tight link or has only one tight link. The tight link is remained the same since no crossbar can be inserted between its two terminals. The newly generated crossbars are $1{\times}k$ (or $k{\times}1$), where k is the number of loose links assigned to the corresponding group (line 17).
- Step 3: The newly generated crossbars are added to the crossbar set X (line 18), and the process is repeated until no crossbar is left which can be split.

The partitioning of the loose links in Step 1 is done such that the total bandwidth in each group is as close to each other as possible in order to balance the loads in the network. The splitting procedure is iteratively performed on the crossbars in descending order to the degree of the crossbars (i.e., the number of in/out links of the crossbars). The rationale behind the order is that the crossbar having more loose links will probably have more chances to be split into larger number of smaller crossbars.

```
Algorithm 2 : Iterative Crossbar Merging
Input: initial topology $T$
Output: topology $T$
 1:  initialize $P = \emptyset$;
 2:  for all $(x_i, x_j)$ where $x_i, x_j \in X$ do
 3:    $m_{i,j}$ = calculate_merge_cost($x_i, x_j$);
 4:    if $m_{i,j} > 0$ do
 5:      $P = P \cup \{m_{i,j}\}$;
 6:    end if
 7:  end for
 8:  while $P \neq \emptyset$ do
 9:    $(x_i, x_j)$ = crossbar pair having largest $m_{i,j} \in P$;
10:    $T$ = merge_switch_pair($x_i, x_j$);
11:    $S$ = selective_reevaluation($L$);
12:    for all $m_{i,j} \in S$ do
13:      $m_{i,j}$ = calculate_merge_cost($x_i, x_j$);
14:      if $m_{i,j} > 0$ do
15:        $P = P \cup \{m_{i,j}\}$;
16:      else $P = P \setminus \{m_{i,j}\}$;
17:      end if
18:    end for
19:  end while
20:  return $T$;
```

Fig. 5. Iterative crossbar merging algorithm.

4.3 Iterative Crossbar Merging

4.3.1 Overall Procedure

This phase improves the initial topology generated in the first phase by iteratively merging the crossbars in the topology, and the algorithm is presented in Fig. 5. In the algorithm, the set P is used to keep positive merge costs (i.e., those $m_{i,j}$'s that can improve the solution).

The algorithm starts with initializing P to null. Then, the algorithm calculates the merge cost $m_{i,j}$'s of all crossbar pairs by using the function `calculate_merge_cost` (line 3). To calculate the merge cost of each crossbar pair, we first examine whether or not the merge of these two switches yields multiple paths in order to hold the assumption given in Section 3.2 (i.e., the single-path assumption). If the merge holds the single-path assumption and $m_{i,j}$ is positive (i.e., merging x_i and x_j yields a feasible and better topology than the current one), $m_{i,j}$ is added to P (line 2-7) as a merging candidate. If $m_{i,j}$ is positive but produces multiple paths, we identify the appropriate links (or edges) to be eliminated for the single-path assumption and the link numbers are recorded into the attribute of $m_{i,j}$ called *removal_links*, a vector denoted as $r_{i,j}$ for $m_{i,j}$. If $r_{i,j}$ is not null, then $m_{i,j}$ is added to P, since the merge can hold the assumption by eliminating the links in $r_{i,j}$. We describe more details in Section 4.3.2.

In the while loop starting at line 8, the largest $m_{i,j}$ in P that yields the best quality improvement is selected, and the

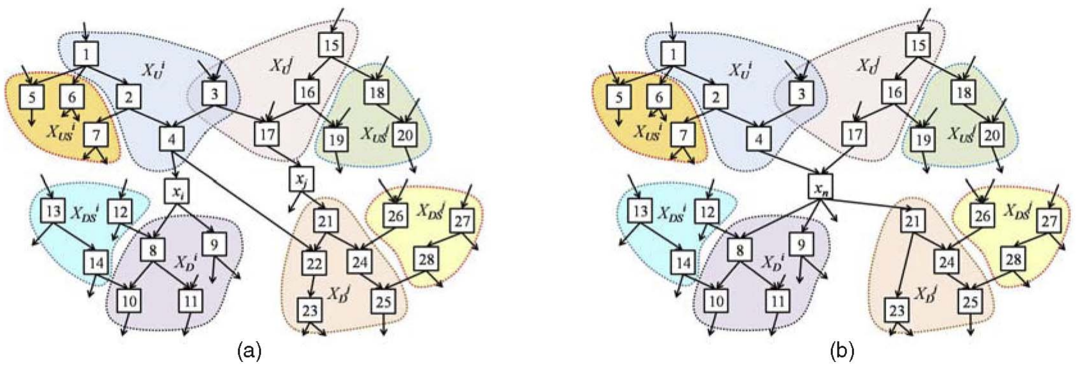

Fig. 6. Multipath manipulation in calculate_merge_cost. (a) Before virtual merge. (b) After virtual merge.

function `merge_switch_pair` improves the topology by merging x_i and x_j into x_n. If $r_{i,j}$ is not null, the corresponding links are removed for the single-path assumption, which will the discussed in Section 4.3.3.

After the merge, we need to update the merge costs affected by x_n for the next iteration (or next merge). In the worst case, all the merge costs should be recalculated with a large computation overhead. To mitigate this issue, we perform the function `selective_reevaluation` which selects $m_{i,j}$'s whose values are likely to be affected by the merge. The selected merge costs are denoted as a set S. The function `selective_reevaluation` also provides a way to trade off the synthesis quality and the computation time by a parameter called ***selection level*** L (line 11), which will be introduced in detail in Section 4.3.4.

Then, the function `calculate_merge_cost` recalculates the value of each element in S, and P is updated based on the new merge cost. This iteration is terminated when P becomes null. Finally, the algorithm returns the optimized topology.

4.3.2 Calculating Merge Cost

This function takes two crossbars as its inputs and outputs their merge cost. By its definition, the merge cost of x_i and x_j represents the cost reduction (in our objective, the network area) incurred by merging two crossbars. Unfortunately, it cannot be obtained by simply comparing the area sum of two crossbars with the area of the crossbar to be created by the merge. It is because other crossbars can also be modified by the merge due to the following reason: if merging x_i and x_j produces multiple paths in the resulting topology, some links should be removed to hold the single-path assumption. It is also necessary to remove the corresponding ports (and their internal connections) of some crossbars connected to the links since they are no longer useful due to the link removal.

There are two necessary and sufficient conditions to detect the multipaths when we merge x_i and x_j: 1) there exists at least a common ancestor or a successor of x_i and x_j since x_n, the new crossbar created by merging x_i and x_j, also becomes a common endpoint to the paths from the common ancestor or to the common successor. 2) At least one ancestor (successor) of x_i is connected to the successor (ancestor) of x_j. In this case, the ancestor can reach the successor through the path which already exists and also through the path including x_n.

To handle the multipath issue, we need to trace the topology created by merging two crossbars. However, we do not know which merge provides the largest merge cost at this time. For this reason, we ***virtually*** merge two crossbars in `calculate_merge_cost`, meaning that the crossbars and links to be modified are only marked with their attributes rather than actually merged. The actual merge is performed by the function `merge_switch_pair` in the next step. In the remainder of Section 4.3.2, "merge" or "removal" means setting the attributes of the corresponding crossbars or links.

For convenience, we call two common endpoints of a multipath *the start node* and *the end node*, respectively. That is, the node placed in the arrow tail is the start node and the node placed in the arrow head is the end node in a directed topology graph.

There are several options to eliminate multipaths. However, we limit the possible choices to the neighbors of the start node and the end node. In other words, we do not consider the links of the intermediate nodes forming the paths in order to reduce the computation overhead. First, we examine at the start node whether the sum of the traffics on all the fan-out links does not exceed the maximum link capacity.[2] If not, we select a link whose capacity is the largest and update its bandwidth attribute called $b_{i,j}$ by the sum of the traffics. In other words, the selected single link delivers all the traffics loaded on the start node. If the traffic sum exceeds the maximum link capacity, we examine at the end node in a similar manner. If the sum of traffics on all its fan-in links does not exceed its maximum input link capacity, we treat the input links of the end node as we do for the start node. Otherwise, we cancel the merge since the merge of these two crossbars incurs unallowable traffic congestions at the start node and/or end node.

Note that we do not eliminate the links in this step since a virtual merge is performed. Instead, the identified link numbers are stored in $r_{i,j}$ for later use if the two crossbars are actually merged.

The following example shows how we handle the multiple paths in this step.

Example 1. In Fig. 6a, the common ancestor of x_i and x_j is x_3 and their merge yields two paths—$x_3 \to x_4 \to x_n$ and

2. The maximum link capacity is the channel width multiplied by the clock frequency.

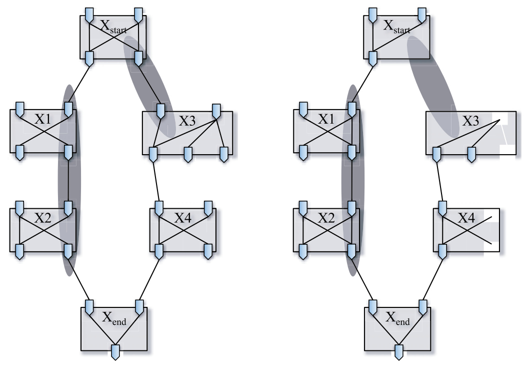

Fig. 7. Internal connection manipulation for multipath removal. (a) Before virtual link removal. (b) After virtual link removal.

$x_3 \rightarrow x_{17} \rightarrow x_n$, which correspond to the first multipath condition. We examine the fan-out links of x_3 to see whether the total amount of traffic of its fan-out links does not exceed x_3's fan-out link capacity. It is also examined the effect when the links between x_3 and x_4 (for convenience, $l_{3,4}$) and between x_3 and x_{17} (for convenience, $l_{3,17}$) are merged. They can be merged with redirecting the traffic through $l_{3,4}$ to $l_{3,17}$ and removing $l_{3,4}$ or vice versa. Between the two cases, the one which is better for the load balancing is selected. For example, suppose that $l_{3,4}$, $l_{3,17}$, $l_{4,n}$, and $l_{17,n}$ carry the bandwidth of 100, 50, 200, and 300 MB/s, respectively. If we remove $l_{3,4}$, the traffic through it will be rerouted to $l_{3,17}$, and thus, $l_{17,n}$ will carry 400 MB/s while the bandwidth through $l_{4,n}$ will be reduced to 100 MB/s. On the other hand, if $l_{3,17}$ is removed, $l_{4,n}$ and $l_{17,n}$ will carry the same amount of bandwidth, 250 MB/s. Therefore, $l_{3,17}$ is finally selected for the elimination through the virtual merge, as shown in Fig. 6b. The selected link number is added to $r_{i,j}$ to used when these two crossbars are actually merged.

The second multipath condition can be demonstrated by x_4 and x_{22} in Fig. 6a, where x_4 (an ancestor of x_i) has a path to x_{22} (an successor of x_j). By virtually merging x_i and x_j, we find two paths—$x_4 \rightarrow x_n \rightarrow x_{21} \rightarrow x_{22}$ and $x_4 \rightarrow x_{22}$. Similarly, the link from x_4 to x_{22} is selected for elimination and the link number is also added to $r_{i,j}$. Note that x_{22} will disappear when their actual merge occurs since it becomes a trivial 1×1 crossbar due to the link elimination.

Afterward, a task is required to ensure that the switches on the remaining path have appropriate internal connections to replace the paths broken by the virtual link elimination. The following example shows how to handle the task:

Example 2. Obviously, there are two paths from x_{start} to x_{end} in Fig. 7a. Suppose that the link between x_{start} and x_3 is virtually eliminated as shown in Fig. 7b. In that case, all the traffics from x_{start} to x_{end} should be delivered through x_1 and x_2; hence, the a new connections has to be added in each switch to construct a path from x_1 and x_2. On the other hand, the internal connections of the virtually removed link should be eliminated in x_3 since they are no longer useful due to the link removal.

Note that we neither create nor delete the internal connections since it is the result of virtual merge. Instead, we record the internal connections to be created or deleted to a vectored attributes, c_{x_i}, and the element of both attributes is represented as $<w, m, s>$, where m and s are the master port number and slave port number of x_i, respectively. Also, w is a boolean variable such that the connection from m to s should be created if $w = 1$. Otherwise, the connection should be deleted.

As shown in Examples 1 and 2, the merge of x_i and x_j modifies other crossbars and their changes should be considered in the merge cost of x_i and x_j. For this purpose, we define a set $X^{i,j}_{NEW}$. Its elements are the crossbars which are connected the links in $r_{i,j}$. In Example 1, $X^{i,j}_{NEW} = \{x_3, x_4, x_{17}, x_{22}\}$, while $X^{i,j}_{NEW} = \{x_{start}, x_1, x_2, x_3\}$ in Example 2. Also, note that such manipulation naturally directs the crossbars to have partial connections without unnecessary internal connections.

After resolving the multipath issue, the merge cost of x_i and x_j is calculated as follows (if not resolved, the merge cost is set to a negative value):

$$m_{i,j} = C_i + C_j - C_n + \sum_{x_k \in X^{(i,j)}_{NEW}} (C_k - C'_k), \tag{1}$$

where C_n is the cost of the resulting crossbar of the merge, and C'_k is the modified area of x_k due to the multipath removal. Note that the partial connection of each crossbar is taken into account in the merge cost calculation, and the details of obtaining the area and delay of a partial crossbar will be presented in Section 4.5. In addition, we set the merge cost $m_{i,j}$ to negative infinity if the merging results in a infeasible topology: 1) if a merging results in such a large crossbar that its maximum clock frequency cannot support the required bandwidth, and/or 2) if one or more latency constraints are violated after the multipath removal, we set the merge cost to negative infinity so that the topology evolves only to the feasible direction.

4.3.3 *Merging Switches*

The function `merge_switch_pair`(x_i, x_j) performs the merge of the two input switches and updates the topology T. For example, suppose that $m_{i,j}$ is the largest merge cost in Fig. 6a, and x_i and x_j are therefore selected for merge at line 9 in Fig. 5. Then, the merge creates x_n followed by removing the links recorded in $r_{i,j}$. The newly created x_n is added to X_{new}. After this process, the topology changes to a better one without violating given constraints and assumptions.

Prior to merge, another important task of this function is performed, which is called *switch classification*. It is a preprocessing to set up a data structure for the next step performed by the function `selective_reevaluation`, where some portion of merge costs is selected for their reevaluation in the next iteration, which can greatly reduce the computing time compared to the exhaustive reevaluation.

Switch classification is applied to each crossbar given as input of this function and then classifies the switches into four classes.

First, it identifies the ancestors of each input crossbar and defines a set of ancestors for each of them. We denote the set as X_U^i for x_i and X_U^j for x_j, as depicted in Fig. 6a. In general, the set of ancestors of switch x_i is defined as follows:

- $X_U^i \subset X$ is a set of the crossbars which have a path to x_i.

Based on the above definition, we compute X_U^i by performing the back-trace from x_i to sources (masters). Also, we back-trace from x_j to its sources to find its ancestors, in Fig. 6a.

Similarly, we can identify the successors of each input crossbar by performing forward-tracing from the crossbar to its sinks (slaves). We also define the set of successors of a switch x_i as follows:

- $X_D^i \subset X$ is a set of the crossbars which have a path from x_i.

In addition, we define two more classes for x_i as follows:

- $X_{US}^i \subset X$ is a set of the crossbars which have a path from $x \in X_U^n$, but are not on the path to x_i.
- $X_{DS}^i \subset X$ is a set of the crossbars which has a path to $x \in X_D^n$, but are not on the path from x_i

We can similarly define X_U^j, X_D^j, X_{US}^j, and X_{DS}^j for x_j. For simplicity in explanation, we use the four classes with respect to x_i, but note that the same principle holds for the four classes defined for x_j.

The switch classification is based on how much it will be affected when x_i is merged to other node. It is obvious that X_U^i and X_D^i will be affected more than the others since the merge effect of x_i can be propagated through the paths. Furthermore, these two classes may include common ancestors and/or common successors which may yield the modification of the topology for multiple path elimination. The other two classes, X_{US}^i and X_{DS}^i may also be affected by the modified switches in X_U^i and/or X_D^i. However, these two classes will be less affected since there are lower possibilities to have paths from/to the modified switches compared to the former case. Fig. 6a shows the classification of both x_i and x_j, and the classification is still maintained after the merge of x_i and x_j as shown in Fig. 6b.

Note that there is another switch class X_{NEW} which is already discussed in Section 4.3.2.

4.3.4 Selective Reevaluation

The key feature of this function, `selective_reevaluation`(L), is how to find those $m_{i,j}$'s that have been largely affected by a merging action, avoiding unnecessary computation of unchanged merge costs. The exact method is to examine the merge cost of every switch pair, but it is not desirable from the runtime perspective. The other extreme approach is to examine the merge costs of the switches paired with an element in X_{NEW}. This approach may miss many of affected merge costs, which degrades the quality of synthesis result. For this reason, we propose a method which is capable of trading off the quality of synthesis result and the search time by using the classes obtained from *switch classification*.

The trade-off can be achieved by defining four merge cost groups called *reevaluation groups*. *Reevaluation groups* are numbered from 1 to 4. The lower the number is, the size of the group is smaller, but its elements (merge costs) have higher chances to be changed.

Using the notations of classes introduced in Section 4.3.3, we can formally define *reevaluation groups*, when we merge two crossbars x_i and x_j in X into x_n.

- ***Reevaluation group 1:*** $m_{p,q}$, where
  - $x_p \in X_{NEW}$ for all $x_q \in X$.
- ***Reevaluation group 2:*** $m_{p,q}$, where
  - $x_p \in X_U^i$ and $x_q \in X_U^j \cup X_{US}^j \cup X_{DS}^j$.
  - $x_p \in X_U^j$ and $x_q \in X_U^i \cup X_{US}^i \cup X_{DS}^i$.
- ***Reevaluation group 3:*** $m_{p,q}$, where
  - $x_p \in X_D^i$ and $x_q \in X_D^j \cup X_{US}^j \cup X_{DS}^j$.
  - $x_p \in X_D^j$ and $x_q \in X_D^i \cup X_{US}^i \cup X_{DS}^i$.
- ***Reevaluation group 4:*** $m_{p,q}$, where
  - $x_p \in X_{US}^i$ and $x_q \in X_{DS}^j$.
  - $x_p \in X_{US}^j$ and $x_q \in X_{DS}^i$.

As far as *reevaluation group 2* is concerned, we ignore X_D^j when we consider x_q. This is because x_q in X_D^j with the switches in X_U^i will incur the second multipath necessary and sufficient condition given in Section 4.3.2, which is unhelpful in the cascaded crossbar network [16]. The same principle is applied to *reevaluation group 3*.

Based on the *reevaluation groups*, we implement a method to trade off the accuracy (synthesis result) and runtime at four different levels, depending upon the aggressiveness of selection. We expect that a lower level implementation provides higher speed-ups but detects smaller portion of the affected merge costs, whereas a higher level implementation detects more affected merge costs but is more time-consuming. The ***selection levels***, L, are defined as follows:

- Level 1: Reevaluate only *reevaluation group 1*.
- Level 2: Reevaluate *reevaluation groups 1, 2, and 3*.
- Level 3: Reevaluate *reevaluation groups 1, 2, 3, and 4*.
- Level 4: Reevaluate every merge cost.

Note that as a smaller portion of reevaluated merge costs is considered (i.e., a lower reevaluation level is used), a larger portion of merge costs is likely to be out of date (or inaccurate), thus producing a suboptimal merge. On the other hand, as the fraction of reevaluated merge costs increase (i.e., using higher reevaluation level), we can achieve better merges at each iteration at the cost of additional computation time. As will be shown in Section 5, we found that Level 3 can trade-off accuracy and runtime most effectively.

4.4 Complexity Analysis of Iterative Merging

Previous approaches avoided the in-process crossbar topology synthesis without performing any quantitative analysis of the algorithm complexity. In our work, we provide a complexity analysis of our method to appreciate whether the in-process crossbar topology synthesis is worthwhile from the runtime perspective.

As mentioned in Fig. 4, the initial topology from our method has N crossbars. For a given initial topology, there are total $\binom{N}{2}$ possible cases, if we select two switches for merging out of N switches. For each merging case, we compute the merging cost as shown in line 1 through line 7 in Fig. 5. The loop from line 8 to line 19 performs iterative merging and the merging strategy is determined by L, ***selection level*** parameter.

In this algorithm, the lower bound of its complexity can be found when $L = 1$, whereas its upper bound is when $L = 4$ as analyzed in the following. The analysis starts with calculating the bound of the loop iteration. The loop is iterated at most $N - 1$ since each iteration merges only two switches. However, in terms of complexity analysis, the best case is that no positive merge cost is in P for initial topology so the synthesis process is finished without merge, regardless of what the selection level is. In this case, the complexity is just $O(N)$ and it is not meaningful in showing the lower bound of our method.

On the other hand, in terms of upper bound, all the merge costs are reevaluated at each iteration when $L = 4$; hence, the number of reevaluation at the kth iteration is obviously $\binom{N-k}{2}$, and therefore, the number of reevaluations is at most $\sum_{k=1}^{N-2}\binom{N-k}{2} = (N^3 - 3N^2 + 5N)/6$. In other words, the worst-case algorithm complexity when $L = 4$ is $O(N^3)$. The worst-case algorithm complexity of other ***selection levels*** lies under the $O(N^3)$. Note that if the in-process partial crossbar is taken into account, the computational burden obtained for a single crossbar, discussed in Section 4.3.2, should be applied to all the crossbars for each iteration, raising the worst-case complexity to $O(N^3)$.

4.5 Partial Crossbar Characterization

For the full crossbars, we adopt the characterization approach in [16], [18], [15], where the RTL codes of crossbars of all concerned sizes are generated and synthesized. This approach is acceptable for the full crossbar characterization since it requires a couple of hundreds of synthesis which can be easily parallelized and done in a day with a high-performance machine. However, this characterization approach is not adequate for the partial crossbars since there are plenty of partial connection configurations for an $m \times n$ crossbar. Instead, we model the area, delay, and power of a partial crossbar using the synthesis results of the full crossbars and the input/output ports.

As shown in Fig. 8, a crossbar is composed of input ports, output ports, and their connections. Any configuration of full or partial crossbar can be assembled with a set of input and output ports and corresponding connections. Since no logic block is required between the input ports and output ports but only nets exist, the ports must be the dominant factor for area and power. Therefore, we model the area and power consumption of a partial crossbar as the sum of area and power, respectively, consumed by the ports. That is, the area of a partial crossbar is calculated as follow:

$$A^{PC} = \sum_i A_i^{port}, \tag{2}$$

where A^{PC} is the area of the partial crossbar, and A^{port} is the area of its port, either input or output.

We use the similar approach for power consumption, that is, the power consumption of a partial crossbar is obtained by summing the power consumed by the ports. For the power consumption of a port, P^{port}, we adopt the model used in [17], which is as follow:

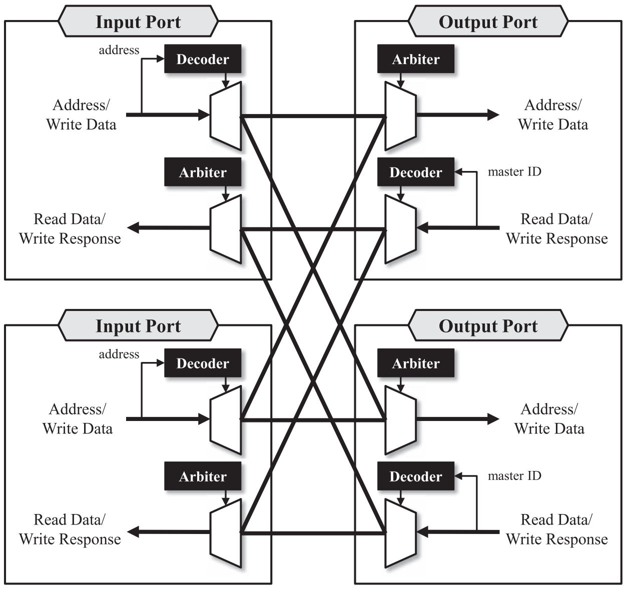

Fig. 8. 2×2 crossbar architecture.

$$P_i^{port} = leak_i + \alpha_i \cdot f_i + \beta_i \cdot \tau_i \cdot f_i, \tag{3}$$

where $leak_i$, f_i, and τ_i are leakage power (Watt), clock frequency (Hz), and transition activity (transitions/sec) of the port. The second term of the right-hand side of (3) is the traffic-independent but clock-dependent dynamic power (e.g., clock network), and the third is the traffic- and clock-dependent dynamic power (e.g., register). α and β have the units of Watt/Mhz and J/transition, respectively, and are measured for every size of input and output ports. Then, the power consumption of a partial crossbar, P^{PC}, is obtained as follow:

$$P^{PC} = \sum_i P_i^{port}. \tag{4}$$

Lastly, we model the delay of a partial crossbar as the delay of the full crossbar which has the same critical path delay. For example, suppose a 6×4 partial crossbar in which the critical path is the path between the master port having fan-out of three (i.e., the number of slave ports to which it is connected is three) and the slave port having fan-in of four (i.e., the number of master ports to which it is connected is four). Then, we model the delay of the partial crossbar as the delay of 4×3 full crossbar.[3]

To examine the accuracy of our partial crossbar model, we measured the characteristics of fifteen 5×5 partial crossbars and compared them with the values obtained by our model. Fig. 9 shows the absolute percentile errors of our models for area, delay, and power consumption. The 15 partial crossbars Partial_1 to Partial_15 have 10-24 buses inside. The result shows that the error is within eight percent for all the metrics. Specifically, the maximum errors for area, delay, and power consumption are 0.14, 7.95, and 6.55 percent, respectively. Based on this result, we believe that the proposed models for partial crossbar characterization are accurate enough to be

3. Although the clock frequency can be set differently for each path technically, single clock frequency is usually assumed for a switch [10], [15], [16], [18], [19], [20], [21].

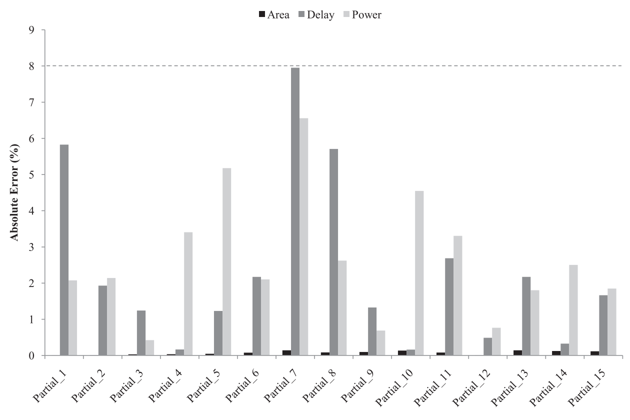

Fig. 9. Partial crossbar model accuracy.

used in our topology synthesis method. By using the models, we can avoid time-consuming synthesis of tens of thousands of partial crossbars, and only need to perform tens of synthesis for ports with various fan-outs and fan-ins.

5 Experiment

5.1 Settings and Overview of Experiments

We tested our method for nine benchmark applications, where five applications are real-world examples and the others are synthetic ones. The real-world examples are an MPEG4 decoder (G2) [26], a multimedia SoC (G3) [16], a mobile multimedia player (G4) [17], an mobile application processer (G5) [17], and a game SoC (G7) [17]. The synthetic benchmarks are generated based on the real-world ones. G1 is derived by removing two masters and two memories from the CTG of G2. G1 represents a relatively small system where all the masters share the only memory for their data communications. G6, G8, and G9 are generated by combining two or more real-world examples. They are used to evaluate the effectiveness and scalability of the proposed method in larger systems, which we may face in a near future. In combining two or more CTGs to make synthetic benchmarks G6, G7, and G9, the cores having similar functionality in different CTGs are mapped to the same core in the newly generated CTG and the corresponding edges are linked to the core.

The characteristics of the benchmarks is summarized in Table 1, where $|V_M|$, $|V_S|$, $|E|$, and Width are the numbers of masters, slaves, edges, and the address/data width of the network interface, respectively.

For crossbars and ports, we generated RTL codes of AMBA3 AXI crossbar and port for each size, synthesized them with Synopsys Design Compiler with 90 nm process library, and the prelayout area, delay, and power information are used. We used 0.19 ps/μm [24] and 0.6 fJ/bit/μm [27] for the wire delay and power calculation, respectively. The topology synthesis tool was implemented in C++.[4]

We first conducted a set of experiments to analyze the impact of the parameter and the option in our methods. More precisely, we measured the impact of the parameter

4. More detailed information for the crossbar switches and the benchmarks used in the experiment can be found at http://dtl.yonsei.ac.kr/doc/International_Journal/tc_partialaware_supplement.html.

TABLE 1
Benchmark Description

Name	$\|V_M\|$	$\|V_S\|$	$\|E\|$	Width	Description
G1	7	1	7	32	synthetic, derived from G2
G2	9	3	13	32	mpeg4 decoder
G3	12	4	21	32	multimedia SoC
G4	12	5	20	32	mobile multimedia player
G5	14	5	22	32	mobile application processor
G6	28	8	49	64	synthetic, G2+G4+G5
G7	38	8	88	64	game SoC
G8	49	11	110	64	synthetic, G5+G7
G9	63	12	136	64	synthetic, G2+G4+G5+G7

selection level which trades off the runtime and solution quality. In addition, we addressed the wire delay effect on the network topology by enabling the floorplanning step in the flow shown in Fig. 2 so that the physical validity of every candidate topology can be examined. In these experiments, we use the network area as the cost of the synthesis for simplicity in the comparison.

In the second set of experiments, we investigated the impact of ***in-process partial crossbar*** in terms of area and power saving over ***no partial crossbar*** and ***postprocess partial crossbar*** in Section 5.4. Also, we quantitatively analyzed the role of selective reevaluation step (also referred to as level 3) in ***in-process partial crossbar*** in the same section.

In the final set of experiments, we compared in Section 5.5 our ***in-process partial crossbar*** with the existing methods, namely, the traffic group encoding-based approach [15] (for short TGE), the MILP-based technique [16] (for short MILP), and MILP-based heuristic method [18] (for short MIRO)[5] in terms of area efficiency and runtime scalability.

5.2 Sensitivity Analysis

In our synthesis process, the ***selection level*** parameter plays a significant role for trading off efficiency and accuracy. Here, we examine the area and synthesis-time trade-off for the four ***selection levels*** introduced in Section 4.3. For all four levels, we compare the synthesis time, the selection accuracy, and the area of the resulting topology. The *selection accuracy* is defined as the ratio of the number of reevaluated merge costs to that of the merge costs actually affected by iterative merging.

Fig. 10 shows the result by averaging the comparison criteria over all benchmarks. In case of area, the values are normalized to the results from using level 1. On the other hand, the selection accuracy and synthesis time are normalized to the results from using level 4. The left y-axis is for the synthesis time and the selection accuracy values, while the right y-axis is for the area values. In order to concentrate on the topology synthesis, the floorplanning step in Fig. 2 is disabled in this experiment, and the effect of the floorplanning step will be shown in Section 5.3. As the ***selection level*** increases, the selection accuracy increases, and therefore, better crossbar pairs are merged at each iteration, yielding a smaller area as expected. However, the area

5. In fact, we extended MILP and MIRO from their original works such that they can handle the read and write traffic separately, just as the proposed method and TGE do.

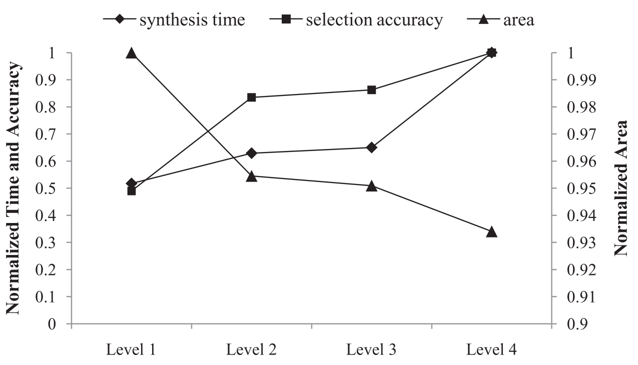

Fig. 10. Sensitivity on the selection level.

efficiency at level 4 is slightly better (1.79 percent) than that at level 3, while the synthesis time is increased by almost twice (1.79×). In fact, for most cases, level 3 yields a satisfactorily good solution in reasonable time.

Fig. 11 compares the empirical complexity of levels 1, 2, 3, and 4 over the benchmarks. The x- and y-axes represent the numbers of the crossbars in the initial topology and the computation time including ***in-process partial crossbar***, respectively. The nine points correspond to G1 through G9, and N (dotted line), N^2 (solid line), and N^3 (dashed line) are given for the sake of convenience in comparison. The result demonstrates that the empirical complexity is much lower than that for all practical cases. The complexity of level 1 and level 4 lie between $O(N)$ to $O(N^3)$.

5.3 Considering Floorplan in Topology Synthesis

Next, we compare the synthesis results with and without floorplanning step. Whenever the topology is changed by merging, the floorplanner Parquet [23] is invoked to check if the topology can be implemented on the die without any unacceptably long wire. More specifically, the floorplanner checks if every wire is within the length that the signals can be transmitted in one clock cycle determined by the topology, and if not, the topology is discarded. The effect of the floorplanning consideration in the resulting topology is shown Fig. 12a. The numbers of the crossbars used are annotated above the corresponding bars. With the floorplanning enabled, the network topology can be substantially different from that without floorplanning, showing at most $6.13\times$ larger area (G7). The deviation is mainly due to wire

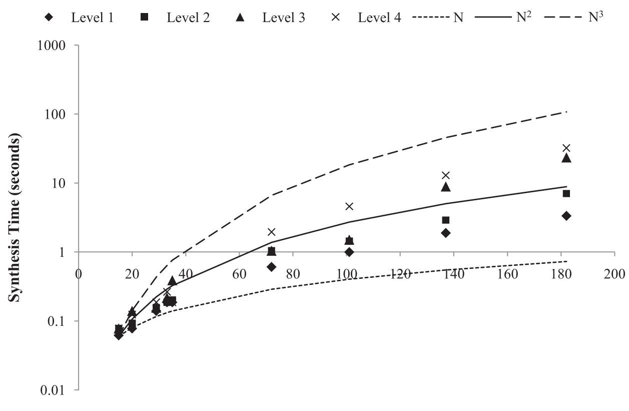

Fig. 11. Computation complexity.

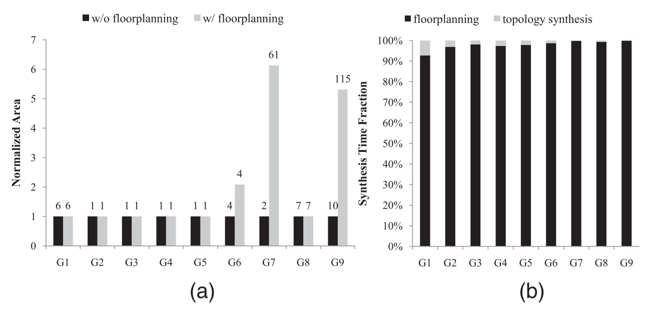

Fig. 12. The effect of considering floorplan. (a) Effect of the floorplanning step on the network topology. (b) Synthesis time fraction.

delay between the IPs and crossbars. As the iterative merging progresses, the number of crossbars decreases, making the relative distances of the IPs and crossbars increase. In fact, more crossbars are needed at most $11.50\times$ (for G9) and $5.96\times$ on average when the floorplan is considered.

Enabling the floorplanning step increases the synthesis time, as can be easily expected. In our experiment, the computation time of the floorplanning dominates that of the entire topology synthesis procedure, as shown in Fig. 12b. This inefficiency comes mainly from the lack of integrity of the topology synthesis and the floorplanning; their design spaces and exploration methods are poorly integrated such that the floorplanning should be done from the bottom for every topology even though the only change is the merging of two crossbars. Even though the topology synthesis with floorplanning is completed at most within 48 minutes (for G9) and it is a reasonable amount of time considering the problem size, the entire synthesis time can be reduced if more efficient floorplanner is used, such as that in [25].

5.4 Performance Analysis of In-Process Partial Crossbar Approach

To appreciate the impact of ***in-process partial crossbar***, we compared the three methods—***no partial crossbar***, ***postprocess partial crossbar***, and ***in-process partial crossbar***. For fair comparison, the same synthesis algorithm was used for all these methods. The only difference among these methods is when the partial crossbar switch is considered. Also, we used the area as the comparison metric for brevity. We used level 3 as the ***selection level*** for all three methods, and the floorplanning was turned off in this experiment in order to concentrate on the effect of partial connection consideration.

The normalized area comparison results are shown in Fig. 13, where, for simplicity, we denote these methods as "No_Partial," "Post_Partial," "In_Partial," respectively. Also, the number above each bar represents how many crossbar switches were used in each synthesized topology. As shown in Fig. 13, ***postprocess partial crossbar*** achieves area saving up to 30.31 percent (for G2) and 8.71 percent on average from ***no partial crossbar***. Since the removal of the unused internal buses is performed after the topology is determined, the number of crossbars used is the same as that of ***no partial crossbar***. ***In-process partial crossbar***, which is the proposed method, shows further reduction from ***postprocess partial crossbar***. Quantitatively, the area is further saved up to 30.35 percent (for G4) and 14.01 percent on average by ***in-process partial crossbar***.

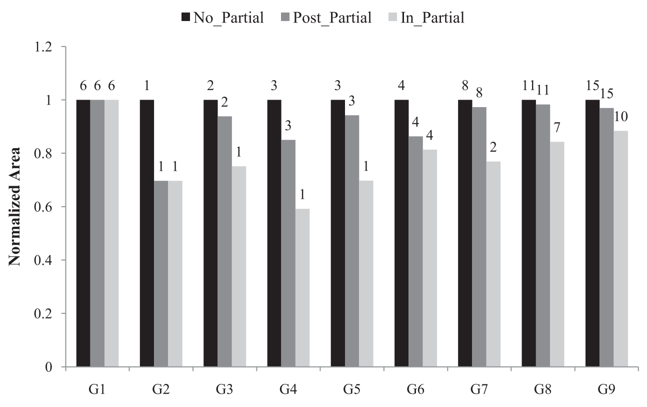

Fig. 13. The area comparison of different partial connection approaches (with L3).

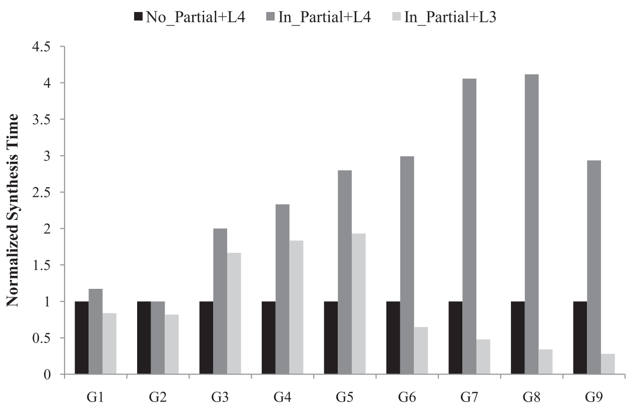

Fig. 15. The computation time variation of in-process partial crossbar with various selection levels.

Such savings are mainly thanks to the partial connection-aware topology synthesis since the resulting topology must have been discarded if we had considered only fully connected crossbar switches during the topology determination. In addition, it is worth noting that the results for G1 are the same for the three methods. It is because G1 has only one memory to which all the traffics are destined, and therefore, the resulting topology must be in shape of a single-rooted tree. In fact, the resulting topology of G1 has a shape of single-rooted tree composed of six 2×1 crossbars.

We additionally conducted an experiment to show that the effect of the cost function to the final solution. Fig. 14 shows the normalized power consumption of the resulting topology when the synthesis objectives are area minimization (i.e., the cost of a crossbar C_k is the area of the crossbar, denoted as obj-area) and power minimization (i.e., C_k is the power consumption of the crossbar, denoted as obj-power). When the objective is power minimization, the power consumption can be saved by up to 54.17 percent (for G6) and 14.74 percent on average compared to when the objective is area minimization.

Next, we measured the impact of the selective reevaluation step on ***in-process partial crossbar***. As mentioned in Section 4, this step can greatly reduce the synthesis time by pruning unnecessary cost-reduction evaluations, but its side effect is the degradation of area efficiency.

Fig. 15 shows the runtime variation of ***in-process partial crossbar*** with various ***selection levels*** used. In the legend, we denote the method with level 4 as "+L4," while the method with level 3 is denoted as "+L3." The runtime is normalized to the ***no partial crossbar*** approach without the selective reevaluation step (i.e., with level 4), which is the first bar of each test case in the figure. As expected from Section 2, ***in-process partial crossbar*** at level 4 drastically increases the synthesis time by up to $4.12\times$ (in case of G8), and the runtime overhead tends to become larger as the problem size increases due to the enlarged search space. However, when ***in-process partial crossbar*** is set to level 3, the synthesis time is greatly reduced by up to 91.70 percent (in case of G8) and 51.59 percent on average against the ***in-process partial crossbar*** at level 4. More interestingly, the ***in-process partial crossbar*** at level 3 shows faster synthesis time even than ***no partial crossbar*** at level 4 from runtime perspectives in many of the test cases. This may be an unexpected result since ***no partial crossbar*** deals with smaller search space by ignoring partially connected crossbars. The result can be justified by understanding the basic principle of our method. It inherently evaluates the cost reduction of all possible crossbar pairs to find the best merging candidates in an iterative manner, and the evaluation hence dominates the overall runtime, which is greatly reduced by the evaluation pruning done at the selective reevaluation step.

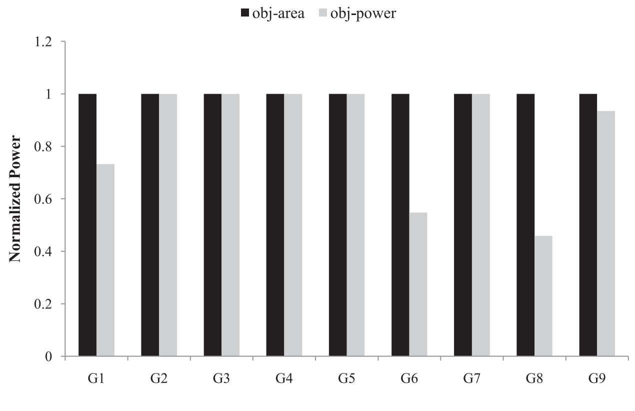

Fig. 14. The effect of the synthesis objective.

5.5 Comparison with Existing Methods

To appreciate the overall impact of our method, we compared it with three previous methods—TGE, MILP, and MIRO. In this comparison, the ***selection level*** of our ***in-process partial crossbar*** method was set to level 3. The floorplanning was also turned off in this experiment and the area is used as the cost since TGE and MILP are not considering the floorplanning and the power consumption. For all the methods, the time-out deadline is set to 48 hours.

The results are shown in Table 2. At the first glance, MIRO and the proposed method outperform the others thanks to consideration of partial connection. Compared to MIRO, the proposed method finds the same solutions for G1 to G5, in which the single partial crossbar is the best solution. However, for larger problems, G6-G9, the proposed method shows better area saving than MIRO, except for G9 where MIRO fails to find a feasible solution. It is mostly thanks to the fact that MIRO applies the partial connection after the topology of the subnetwork is determined, while our method considers it in the calculation of all the merge costs.

TABLE 2
The Comparison of the Synthesis Quality and Time of TGE, MILP, MIRO, and the Proposed Method

	TGE		MILP		MIRO		Proposed		Improv. TGE	Improv. MILP	Improv. MIRO
App	Area	Time	Area	Time	Area	Time	Area	Time	%	%	%
G1	0.67	33.39	0.67	0.26	0.67	1.36	0.67	0.08	0.00	0.00	0.00
G2	0.39	41.32	0.38	837.70	0.27	8.36	0.27	0.14	30.31	28.96	0.00
G3	0.58	71.56	0.51	51159.67	0.38	28.19	0.38	0.39	33.57	24.88	0.00
G4	0.60	60.21	0.53	61426.78	0.40	77.36	0.40	0.34	33.79	25.25	0.00
G5	0.63	111.59	0.60	50311.45	0.43	38.14	0.43	0.45	30.93	28.04	0.00
G6	1.61	372.84	fail	t.o.	1.31	231.34	1.21	1.06	24.68	n/a	7.47
G7	2.00	1132.16	fail	t.o.	1.42	956.46	1.39	1.56	30.27	n/a	2.03
G8	4.11	3601.67	fail	t.o.	2.97	2192.77	2.09	8.94	49.09	n/a	29.49
G9	fail	1238.38	fail	t.o.	fail	289.15	3.05	23.59	n/a	n/a	n/a

Units: mm^2 for Area, seconds for Time.

Another benefit of our method is much faster synthesis time compared to the counter parts. TGE failed to find a feasible solution for the largest problem due to sticking at the local optimum during the simulated annealing. The synthesis time of MILP shot up as the problem size grows and reached the time-out deadline without finding any feasible solution for G6-G9. MIRO showed better solution quality and shorter synthesis time than TGE and MILP for G1-G8, but failed to find a solution for G9 even though the time-out deadline was not reached. It shows the apparent limitation of MIRO on the scalability; MIRO compresses the input communication graph by merging the master or slave nodes, and the merge makes the bandwidth and the latency constrains more and more tight since the bandwidth is accumulated and the latency constraint is taken as the minimum.

Unlike the other methods, our method found the best solutions for all the test cases in much shorter time. Quantitatively speaking, the area is saved by up to 49.09, 28.96, and 29.49 percent compared to TGE, MILP, and MIRO, respectively, even with the failed cases excluded. The synthesis time reduction is more than $400\times$, $85{,}000\times$, and $200\times$, on average, compared to TGE, MILP, and MIRO, respectively.

6 Conclusion

In this paper, we have proposed a synthesis method for crossbar-based networks, in which the partial connection of a crossbar is considered in the middle of the topology determination step. In our synthesis process, which is based on repetitive merging of crossbars, the proposed selective reevaluation feature enables more efficient design space exploration. The experimental result we presented proves that considering the partial connection of crossbars in a cascaded crossbar network results in a great synergy effect, improving the design quality by more than 30 percent compared with the postprocess consideration. Our method achieves impressive quality and scalability improvements over existing topology synthesis methods, such as MILP, TGE, and MIRO, showing up to 29.49 percent area saving and more than $200\times$ synthesis time reduction compared to MIRO which outperforms the other two methods.

Acknowledgments

This work was supported in part by Technology Innovation Program (Industrial Strategic technology development program, KI002168, Development of Configurable Device & SW Environment) funded by the Ministry of Knowledge Economy (MKE, Korea), by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0025423), and by the IDEC (IC Design Education Center). D. Woo was with the School of Electrical and Electronic Engineering, Yonsei University, 134 Sinchon-dong, Seodaemun-gu, Seoul 120-749, Korea.

References

[1] M. Jun, K. Bang, H.J. Lee, N. Chang, and E.Y. Chung, "Slack-Based Bus Arbitration Scheme for Soft Real-Time Constrained Embedded Systems," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '07),* pp. 159-164, Jan. 2007.
[2] K. Lahiri, A. Raghunathan, and G. Lakshminarayana, "LOTTERYBUS: A New High-Performance Communication Architecture for System-On-Chip Designs," *Proc. Ann. Design Automation Conf. (DAC '01),* pp. 15-20, June 2001.
[3] B.C. Lin, G.W. Lee, J.D. Huang, and J.Y. Jou, "A Precise Bandwidth Control Arbitration Algorithm for Hard Real-Time SoC Buses," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '07),* pp. 165-170, Jan. 2007.
[4] M. Drinic, D. Kirovski, S. Megerian, and M. Potkonjak, "Latency-Guided On-Chip Bus Network Design," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 12, pp. 2663-2673, Dec. 2006.
[5] W. Dally and B. Towels, *Principles and Practices of Interconnection Networks.* Morgan Kaufmann Publishers Inc., 2003.
[6] M. Loghi, F. Angiolini, D. Bertozzi, and L. Benini, "Analyzing On-Chip Communication in a MPSoC Environment," *Proc. Conf. Design, Automation and Test in Europe (DATE '04),* pp. 752-757, Feb. 2004.
[7] K.K. Ryu, E. Shin, and V.J. Mooney, "A Comparison of Five Different Multiprocessor SoC Bus Architectures," *Proc. Euromicro Symp. Digital Systems Design '01,* pp. 202-209, Sept. 2001.
[8] AMBA Designer User Guides, http://www.arm.com/products/solutions/AMBA_Designer.html, 2010.
[9] SonicsMX SMART Interconnect Solution, http://www.sonicsinc.com/sonicsMX.htm, 2010.
[10] S. Murali, L. Benini, and G. De Micheli, "An Application-Specific Design Methodology for On-Chip Crossbar Generation," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 26, no. 7, pp. 1283-1296, July. 2007.
[11] S. Murali and G. De Micheli, "An Application-Specific Design Methodology for STbus Crossbar Generation," *Proc. Conf. Design, Automation and Test in Europe (DATE '04),* pp. 1176-1181, Mar. 2005.
[12] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Constraint-Driven Bus Matrix Synthesis for MPSoC," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '06),* pp. 30-35, Jan. 2006.

[13] S. Pasricha and N. Dutt, "COSMECA: Application Specific Co-Synthesis of Memory and Communication Architectures for MPSoC," *Proc. Conf. Design, Automation and Test in Europe (DATE '06)*, pp. 700-705, Mar. 2006.

[14] S. Pasricha, N. Dutt, and F.J. Kurdahi, "Dynamically Reconfigurable On-Chip Communication Architectures for Multi Use-Case Chip Multiprocessor Applications," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '09)*, pp. 25-30, Feb. 2009.

[15] J. Yoo, S. Yoo, and K. Choi, "Communication Architecture Synthesis of Cascaded Bus Matrix," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '07)*, pp. 171-177, Jan. 2007.

[16] M. Jun, S. Yoo, and E.Y. Chung, "Mixed Integer Linear Programming-Based Optimal Topology Synthesis of Cascaded Crossbar Switches," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '08)*, pp. 583-588, Jan. 2008.

[17] J. Yoo, S. Yoo, and K. Choi, "Topology/Floorplan/Pipeline Co-Design of Cascaded Crossbar Bus," *IEEE Trans. Very Large Scale Integration Systems*, vol. 17, no. 8, pp. 1034-1047, Aug. 2009.

[18] M. Jun, S. Yoo, and E.Y. Chung, "Topology Synthesis of Cascaded Crossbar Switches," *IEEE Trans. Computers-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 6, pp. 926-930, June 2009.

[19] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo, "Designing Application-Specific Networks on Chips with Floorplan Information," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '06)*, pp. 355-362, Jan. 2006.

[20] K. Srinivasan, K.S. Chatha, and G. Konjevod, "Linear-Programming-Based Techniques for Synthesis of Network-On-Chip Architectures," *IEEE Trans. Very Large Scale Integration Syntems*, vol. 14, no. 4, pp. 407-420, Apr. 2006.

[21] K. Srinivasan, K.S. Chatha, and G. Konjevod, "An Automated Technique for Topology and Route Generation of Application Specific On-Chip Interconnection Networks," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '05)*, pp. 231-237, Nov. 2005.

[22] K.S. Chatha, K. Srinivasan, and G. Konjevod, "Automated Techniques for Synthesis of Application-Specific Network-On-Chip Architectures," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 8, pp. 1425-1438, Aug. 2008.

[23] S.N. Adya and I.L. Markov, "Fixed-Outline Floorplanning: Enabling Hierarchical Design," *IEEE Trans. Very Large Scale Integration Systems*, vol. 11, no. 6, pp. 1120-1135, Dec. 2003.

[24] L. Carloni, A.B. Kahng, S. Muddu, A. Pinto, K. Samadi, and P. Shama, "Interconnect Modeling for Improved System-Level Design Optimization," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '08)*, pp. 258-264, 2008.

[25] J.Z. Yan and C. Chu, "DeFer: Deferred Decision Making Enabled Fixed-Outline Floorplanner," *Proc. Ann. Design Automation Conf. (DAC '08)*, pp. 167-172, 2008.

[26] S. Murali and G. De Micheli, "SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs," *Proc. Ann. Design Automation Conf. (DAC '04)*, pp. 914-919, June 2004.

[27] S. Yan and B. Lin, "Application-Specific Network-On-Chip Architecture Synthesis Based on Set Partitions and Steiner Trees," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '08)*, pp. 277-282, 2008.

Minje Jun (M '08) received the BS and MS degrees in electrical and electronic engineering in 2006 and 2008, respectively, from the Yonsei University, Seoul, Korea, where he is currently working toward the PhD degree in electrical and electronic engineering. His research interests include System-on-Chip architecture and Network-on-Chip with the special emphasis on their design automation. He is a member of the IEEE.

Deumji Woo received the BS degree in electrical and electronic engineering from the Yonsei University, Seoul, Korea, in 2009, and he is currently working toward the MS degree in the School of Electrical Engineering and Computer Science, Seoul National University, Korea. His research interests include computer and System-on-Chip architecture, Network-on-Chip, and design methodology for large-scale multiprocessor system-on-chip (MPSoC) and on-chip bus architectures.

Eui-Young Chung (SM '99-M '06) received the BS and MS degrees in electronics and computer engineering from the Korea University, Seoul, Korea, in 1988 and 1990, respectively, and the PhD degree in electrical engineering from the Stanford University, California, in 2002. From 1990 to 2005, he was a principal engineer with SoC R&D Center, Samsung Electronics, Yongin, Korea. He is currently an associate professor in the School of Electrical and Electronic Engineering, Yonsei University, Seoul. His research interests include system architecture and VLSI design, including all aspects of computer-aided design with the special emphasis on low-power applications and flash memory applications. He is a member of the IEEE.