Fault-Tolerant Application Specific Network-on-Chip Design

Parth Shah*, Abhishek Kanniganti†, Soumya J‡
Department of Electrical and Electronics Engineering,
Birla Institute of Technology and Science - Pilani, Hyderabad Campus,
Hyderabad, India
fo Email: *parthjshah95@gmail.com, †nmabhi42@gmail.com, ‡soumyatkgp@gmail.com

***Abstract*—Network-on-Chip (NoC) has been introduced to address the communication problems associated with the traditional bus based System-on-Chip (SoC) architectures. NoC can be designed either using regular or irregular architectures. Even though many regular architectures have been proposed in the literature, there is a mismatch between the application requirements and the design. Application specific NoC designs have been proposed to match the requirements of the applications, which are irregular in nature. Due to the heavy integration of the components on the chip, designs that are vulnerable to faults in links can render the chip unusable. This paper first sets the benchmark of minimum possible communication cost and thereafter proposes a greedy algorithm to develop link fault-tolerant application specific topology for the given application core graph which meets that benchmark.**

***Index Terms*—Link fault tolerance, greedy algorithm, application specific network on chip, topology design, graph theory**

I. Introduction

Recent advances in technology have enabled to integrate millions of transistors on a single chip, so much so that multiple components can now be integrated on a single chip, leading to the development of System-on-chip (SoC). Traditional SoC designs used bus topology for the communication between the components. But this can lead to saturation very fast and is not scalable. Network-on-Chip (NoC) [1] has been introduced to overcome the problems associated with bus based communication. For generic NoC, symmetric topologies like mesh are widely used, but there is a mismatch between the regular topologies and the application requirements. However, for Application Specific NoC (ASNoC), irregular topologies can reduce hardware costs and improve performance. A survey of ASNoC design techniques has been presented in [2]–[4]. They enumerate the advantages of custom topologies over standard ones for ASNoC.

The network components which connect various cores in the NoC are prone to failure and can result in the complete breakdown of the system. Hence, it is essential to design NoC topologies with fault tolerance, which can be attained by adding more links and routers to the existing topology and create alternate paths between routers and cores. The proposed solution in this paper is a graph theoretic greedy algorithm that generates a link fault tolerant topology from the given application specifications with minimum communication cost.

This work is partially supported by the research project No. ECR/2016/001389 dt. 06/03/2017, sponsored by the SERB, Govt. of India

II. Related Work

A fault-tolerant irregular topology generation method has been presented in [3] for Application Specific NoC designs. They discuss how designed NoC topology allows different routing paths if there is a link failure on the default routing path. The solution proposed in this paper generates a fault tolerant topology by first generating random irregular non-fault tolerant topologies and adding additional links to make them fault tolerant. They compare fault-tolerant topologies with non-fault-tolerant application-specific irregular topologies on energy consumption, performance, and area using multimedia benchmarks and custom-generated graphs. They have used simulated annealing for the mapping process. However, the mapping has been considered after the topology is generated instead of considering the application requirements while synthesising the topology.

A fault-tolerant application-specific NoC architecture has been presented in [6] to tolerate permanent router failures. They have implemented the architecture in VHDL and synthesized in Xilinx ISE. They have proposed spare router and a link-interface in the network by considering 3-dimensional NoC and shown the results in 2-D environment as well. However, they have considered Through-Silicon-Vias (TSVs) in the process. To apply the same techniques to 2-D, the assumptions to be made must be different than that of 3-D, which has not been addressed. The same group has presented a fault-tolerant application-specific NoC design in [7]. However, they have shown the simulation results only on mesh architectures instead of application-specific topology.

In [8] algorithms from three different domains for the construction of fault-tolerant graphs are evaluated. Classical graph-theoretic algorithms, optimization and bio-inspired ap-

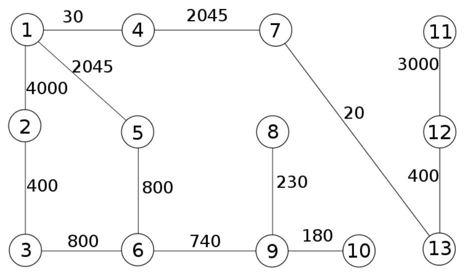

Fig. 1. Sample input core flow graph

proaches are compared regarding the quality of the generated graphs as well as concerning the runtime requirements.

Fault-tolerant application-specific NoC design has been implemented in [9] using FPGA. However, the implementation methods have not been given in detail and they have not explored the architectural optimizations while implementing on FPGA.

[11] demonstrates a fault tolerant topology where its dual-port network interface (NI) makes the key PE have two links with network at least, which can ensure high reliability of the system. An adaptive fault tolerant routing algorithm is also developed that can sense congestion on the network to work around the issues effectively.

In [10], they use the generalized binary de Bruijn's (GBDB) graph as a scalable and efficient network topology for an on-chip communication network. The experimental results show that the latency and energy consumption of generalized de Bruijn's graph are much less compared to Mesh and Torus, the two common NoC architectures in the literature. Hence, this paper compares the De Bruijn's topology with the one presented here.

III. Problem Definition

Input to our approach is an application which gives information such as the number of cores and the communication bandwidth requirement between each pair of cores. Our objective is to create topologies which have:

1) cores communicating with each other through at least two distinct paths,
2) minimum latency and energy consumption and
3) less hardware overheads.

To achieve this, each link must be such that even if it is removed from the topology, the two routers which were connected would still be able to have connection using an alternate path. Thus, even if the link fails, the routers will not get cut off from each other. Also, latency and power consumption are assumed to be directly proportional to the total communication cost defined in section III-B.

A. Core Flow Graph and Topology Graph

The application information is given with the help of a core flow graph (CFG) as defined in [3]:

> A core flow graph (CFG) is a graph $G(N, E)$ where each vertex $n_i \in N$ represents a core (i.e., a node) in the application, and each edge $e_{i,j} \in E$ represents a dependency between two cores n_i and n_j. The amount of data transfer between n_i and n_j is represented by the weight $w_{i,j}$ for all $e_{i,j}$ and is given in bits per second.

A sample CFG is shown in fig. 1. The output is a network of routers or topology where each router is mapped to a core. This can be represented by the topology graph (TG) defined in [3]:

> A topology graph (TG) is a connected graph $T(R, L)$ where R represents the set of routers and L represents the set of links connecting the routers.

B. Communication cost

In the CFG, for every edge $e_{i,j}$, the communication cost $c_{i,j}$ associated with it is the product of its weight $w_{i,j}$ and the number of hops from r_i to r_j (denoted by $h_{i,j}$). The total communication cost can be defined as a sum of costs $c_{i,j}$ associated with all the edges. This can be succinctly described in the two following equations:

$$c_{i,j} = h_{i,j} * w_{i,j} \tag{1}$$

$$C = \sum c_{i,j} \quad (\forall e_{i,j} \in E) \tag{2}$$

C. Link Fault Tolerance

The primary goal of this paper is to create link fault tolerant topologies. Here, we use the following definition of fault tolerance:

The link $l_{i,j}$ is said to be fault tolerant if the routers r_i and r_j have an alternate path for the communication other than the link $l_{i,j}$. The link fault tolerance of a topology is defined as the percentage of such links in the topology.

D. Degree

The Degree of a router is the number of links connected to it.

E. Assumption

In all the following algorithms, we assume that each core n_i in N in the CFG is mapped to a single router r_i in R in the TG.

IV. Native Topology

The simplest technique to build a topology $T(R, L)$ is to replace all nodes with routers and edges with links in the CFG, $C(N, E)$. The algorithm 1 does exactly that.

This simple topology has the lowest communication cost theoretically possible, since $h_{i,j} = 1$ for all edges $e_{i,j}$. Hence, this sets a benchmark for the minimum communication cost. Any topology that includes this as a subgraph will have the same communication cost. Hence, this topology is intended to be the basis for the algorithm 2. When this technique is applied to the input graph in fig. 1, the resultant topology is illustrated in fig. 2.

Algorithm 1 Native Topology

```
Input: CFG $G(N, E)$
Output: TG $T(R, L)$
1: for all cores $n_i \in N$ in $G(N, E)$ do
2:     add $r_i$ to $R$ in $T(R, L)$
3: end for
4: for each edge $e_{i,j} \in E$ do
5:     create a link $l_{i,j}$ connecting routers $r_i$ and $r_j$ and add it to $T$ in $T(R, L)$.
6: end for
7: return $T(R, L)$
```

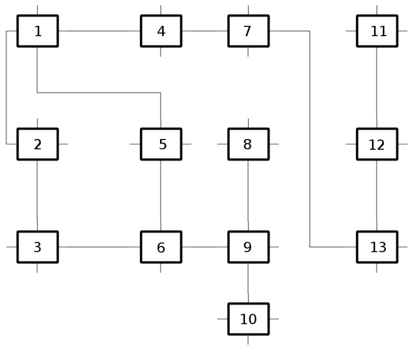

Fig. 2. Native topology

V. Poorest Neighbour Algorithm

The native topology has the lowest possible communication cost, as mentioned in section IV. However, it might not be link fault tolerant. To achieve that, we add some more links. But, we still include the entire native topology so as to keep communication cost to the minimum. In the following algorithm, we test whether each link $l_{i,j}$ in the native topology has an alternate path connecting routers r_i and r_j (going in the descending order of the corresponding weights $w_{i,j}$). If not, we connect it to a neighbouring router with the lowest degree (defined in section III-D). The rationale behind this algorithm is that the edges with highest weights should have the lowest possible hops in order to reduce the total communication cost and that the routers should have similar loads (and therefore similar degrees). This is the 'poorest neighbour' algorithm (algorithm 2). When this algorithm is applied to the CFG in fig. 1, result looks like fig. 3.

VI. De-Bruijn's Algorithm

De Bruijn's graph is a popular algorithm which is widely used in the field of Bioinformatics. Works such as [10] suggest De Bruijn's graph as an alternate topology and compared its energy consumption with popular regular topologies like Mesh and Torus. This graph is inherently link fault tolerant and has lower energy consumption than regular topologies like Mesh as established by [10]. It may also contain more than one alternate path for a pair of nodes. The algorithm 3 is also explained in [5]. In the later sections of the paper,

Algorithm 2 Poorest Neighbour Algorithm

```
Input: CFG $G(N, E)$
Output: TG $T(R, L)$
1: Generate the native topology $T(R, L)$ using algorithm 1.
2: for all links $l_{i,j}$ in $L$ do
3:     if there exists an alternate path between $r_i$ and $r_j$ then
4:         continue
5:     else
6:         let $r_l$ be the router with the lower degree from routers $r_i$ and $r_j$
7:         let $r_h$ be the router with the higher degree from routers $r_i$ and $r_j$
8:         from the list of all the neighbouring routers of $r_h$, let $r_p$ be the router with the lowest degree. {$r_p$ is the 'poorest' neighbour of $r_h$.}
9:         create link $l_{p,h}$ connecting routers $r_p$ and $r_h$.
10:        add $l_{p,h}$ to $L$ in $T(R, L)$.
11:    end if
12: end for
13: return $T(R, L)$
```

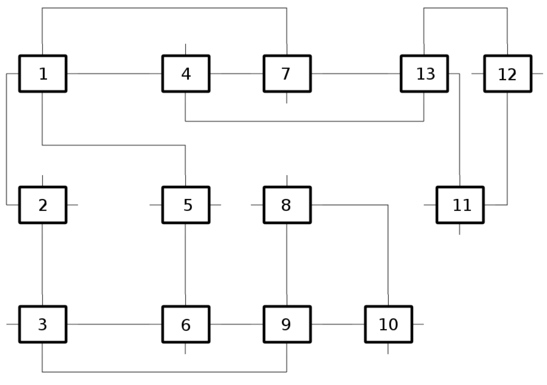

Fig. 3. Result of poorest neighbour algorithm

its performance is compared against the poorest neighbour algorithm (algorithm 2). When applied to the CFG in fig. 1, the result looks like fig. 4

VII. Observations

The main objective of this paper is to create a topology with 100% link fault tolerance. However, as seen in table I, the poorest neighbour algorithm does not always provide 100% link fault tolerance. We investigate the cause and find a solution for this problem in this section. With the solution proposed here, the poorest neighbour algorithm provides 100% link fault tolerance in all tested cases.

A. *Fail Case: Isolated Pair*

Consider a disconnected graph like the one shown in fig. 5. The poorest neighbour algorithm first checks if each link has an alternate path for connecting the two end routers of that link. If yes, then it does nothing. If no, then the algorithm seeks to connect one of the routers with a neighbour of the

Algorithm 3 De-Bruijn's Algorithm

```
Input: CFG $G(N, E)$
Output: TG $T(R, L)$
1: let $R = N$ {the number of routers in TG and cores in CFG is the same since there is a one to one mapping}
2: Connect routers $r_1$ and $r_2$ with link $l_{1,2}$ and add it to $L$ in $TG(R, L)$.
3: let $p = 2$ and $c = p + 1 = 3$.
4: while $c < R$ do
5:     Connect routers $r_c$ and $r_p$ with link $l_{c,p}$ and add it to $L$ in $TG(R, L)$.
6:     $c = c + 1$.
7:     if $c$ is odd then
8:         $p = p + 1$.
9:     end if
10: end while
11: let $p = R$ and $c = p - 1$.
12: let $e = N \mod 2$
13: while $c > 0$ do
14:     Connect routers $r_c$ and $r_p$ with link $l_{c,p}$ and add it to $L$ in $TG(R, L)$.
15:     $c = c - 1$.
16:     if $c \mod 2 = e$ then
17:         $p = p - 1$.
18:     end if
19: end while
20: return $T(R, L)$
```

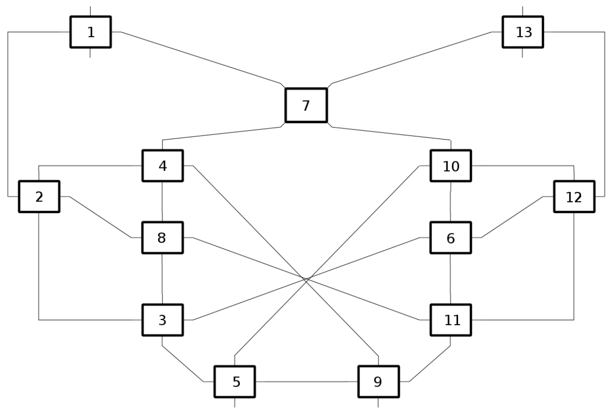

Fig. 4. De Bruijn's graph

TABLE I
LINK FAULT TOLERANCE

Number of cores	Link fault tolerance		
	Poorest neighbour (without solution)	*Poorest neighbour (with solution)*	*De Bruijn's graph*
8	100	100	100
24	100	100	100
30	86.20	100	100
64	100	100	100
64	99.0	100	100
128	100	100	100

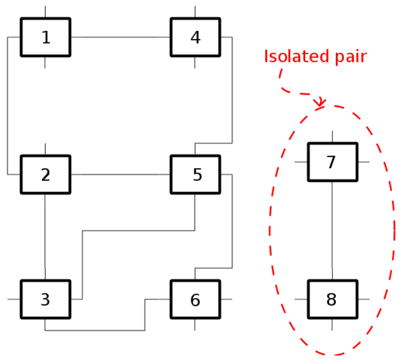

Fig. 5. Graph with an isolated pair

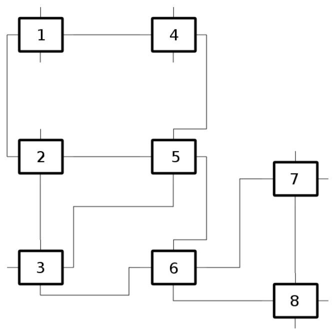

Fig. 6. proposed solution

other router. But in this graph, consider the routers r_7 and r_8. None of these routers have another neighbour to connect to. No connection is thus made, and the resulting topology does not provide 100% link fault tolerance.

B. Proposed solutions

In this case, the proposed solution is to modify the algorithm to connect the routers to the router (in the rest of the topology) with the lowest degree. Here, the routers r_1 and r_6 are of degree 2. Thus, if we choose router r_6, the resulting topology would look something like fig. 6. Similarly, if we choose router r_1, the links would be established between r_7, r_8 and r_1. This procedure is illustrated in algorithm 4 and is to be performed after the completion of poorest neighbour algorithm (2).

Algorithm 4 Solution to isolated pair scenario

```
Input: TG $T(R, L)$
Output: TG $T(R, L)$
1: for all $l_{i,j}$ such that $degree(r_i) = 1$ and $degree(r_j) = 1$ do
2:     from the rest of the topology, find $r_p$ with lowest degree. {if there are more than one routers with the lowest degree, choose one arbitrarily.}
3:     connect both routers $r_i$ and $r_j$ to $r_p$.
4: end for
5: return $T(R, L)$
```

VIII. Results

In this section, the comparison is performed for all the topology generation methods. The mentioned algorithms were implemented in JAVA and simulated by generating different networks used in the paper. Every algorithm was evaluated by considering overall communication cost and the required number of links. The results are available in [12]. The binaries of implementations and input data are available in [13]. Table II compares the communication cost of the topologies generated by native, poorest neighbour and De Bruijn's algorithms against the number of cores in the input CFG. It is clear from this comparison that the communication cost of the topology generated by poorest neighbour always matches the communication cost of the native topology irrespective of the number of cores contained in the CFG. Hence, poorest neighbour algorithm is highly scalable for fault tolerant topology generation. Also, we can infer that for a given input graph, the communication cost of the topology generated by poorest neighbour is very less compared to the topology generated by De Bruijn's algorithm. Thus, DeBruijn's algorithm is not very scalable in comparison. Table III compares the number of links required for the topologies generated by the poorest neighbour and De Bruijn's algorithms. Lesser the number of links, lower the hardware cost. In this aspect, the poorest neighbour algorithm is marginally better than De Bruijn's algorithm.

TABLE II
Communication cost

Number of cores	**Communication cost**		
	Native	***Poorest neighbour***	***De Bruijn's graph***
8	576	576	704
24	131	131	401
30	88	88	364
64	24,608	24,608	423,165
64	6,063	6,063	93,454
128	55,404	55,404	1,777,704
128	11,963	11,963	380,437
128	26,281	26,281	892,604
128	163,837	163,837	5,437,165
128	137,777	137,777	4,667,287

TABLE III
Number of links

Number of cores	**Number of links**		
	Native	***Poorest neighbour***	***De Bruijn's graph***
8	8	9	13
24	21	33	44
30	24	37	56
64	95	108	125
64	59	103	125
128	207	234	253
128	127	211	253
128	180	202	253
128	236	265	253
128	192	229	253

IX. Conclusion

The native topology set the benchmark for communication cost. Both De Bruijn's and poorest neighbour algorithms (with the additional procedure mentioned in section VII-B) provide 100% link fault tolerance in all the tested scenarios. However, the latter outperforms the former with significantly lower communication cost as well as marginally lower hardware costs.

References

[1] Benini, L., Micheli, G. D. "Network on chips: a new SoC paradigm," IEEE Computer, Vol. 35, No. 1, pp.70-78, 2002.

[2] Fathollah Karimi Koupaei, Ahmad Khademzadeh, Majid Janidarmian, "Fault tolerant application specific network-on-chip," Proceedings of the World Congress on Engineering and Computer Science 2011, Vol II, WCECS 2011, October 2011

[3] Suleyman Tosun, Vahid B. Ajabshir, Ozge Mercanoglu, and Ozcan Ozturk, "Fault tolerant topology generation method for application specific network-on-chips," IEEE Transactions On Computer - Aided Design Of Integrated Circuits And Systems, vol. 34, no. 9, September 2015

[4] G. Ascia, V. Catania, M. Palesi, "Multi-objective mapping for mesh-based NoC architectures," in: ODES SSS 2004. International Conference in Hardware/ Software Codesign and System Synthesis, 8-10 Sept. 2004, p. 182-187.

[5] Sharma, Tharesh. "Fault Tolerant Network on chips Topologies." Reliable Networks-On-Chip in the Many Core Era. Universitt Stuttgart - SS09, Stuttgart. July 7, 2009. Seminar Report, p. 14 - 17.

[6] Hosseinzadeh, F., Bagherzadeh, N., Khademzadeh, A., Janidarmianm, M. (2014) "Fault tolerant optimization for application specific network-on-chip architecture," LNEE, Vol. 247, No. 2014, pp. 363-381

[7] Koupaei, F. K., Khademzadeh, A., Janidarmian, M. (2011) "fault-tolerant application-specific network-on-chip," In Proceedings of the World Congress on Engineering and Computer Science.

[8] Becker M., Krmker M., Szczerbicka H. (2015) "Evaluating Heuristic Optimization, Bio-Inspired and Graph-Theoretic Algorithms for the Generation of Fault-Tolerant Graphs with Minimal Costs,". In: Kim K. (eds) Information Science and Applications. Lecture Notes in Electrical Engineering, vol 339. Springer, Berlin, Heidelberg, pp 1033-1041

[9] Yesil, S., Tosun S., Ozturk, O. (2016) "FPGA implementation of a fault-tolerant application-specific NoC design,"2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS), Istanbul, pp. 1-6.

[10] M. Hosseinabady, M. R. Kakoee, J. Mathew and D. K. Pradhan, "Reliable network-on-chip based on generalized de Bruijn graph," 2007 IEEE International High Level Design Validation and Test Workshop, Irvine, CA, 2007, pp. 3-10.

[11] Pengfei Yang, Quan Wang, Wei Li, Zhibin Yu, Hongwei Ye, "A Fault Tolerance NoC Topology and Adaptive Routing Algorithm," 2016 13th International Conference on Embedded Software and Systems (ICESS), pp. 42-47

[12] Shah, Parth; Kanniganti, Abhishek (2017): Simulation results. figshare. https://doi.org/10.6084/m9.figshare.5590015.v2. Retrieved: 07:47, Nov 12, 2017 (GMT)

[13] Shah, Parth; Kanniganti, Abhishek (2017): PoorestNeighbourAlgorithm. figshare. https://doi.org/10.6084/m9.figshare.5593468.v1 Retrieved: 10:10, Nov 12, 2017 (GMT)