Fault Tolerant Topologies in Application Specific Network-on-Chip

*Abstract*--In recent times the number of transistors in a chip is growing at a rapid rate. Hence the communication between the modules in a System on Chip (SoC) cannot be handled effectively with the help of the traditional bus architecture. To overcome this difficulty the Network-on-Chip (NoC) design was introduced. Application specific network on chip designs are asymmetric. This design is vulnerable to faults that can render the chip unusable. This project aims at developing fault tolerant topologies for this network. We present two algorithms for creating fault tolerant irregular topologies and then compare them on the basis of communication cost, increase in the same with faults, hardware cost overheads and fault tolerance. We also set the theoretical lower limits for communication cost and the number of links and routers required for any topology to be designed for a given input application specifications.

*Keywords*--Communication cost, Fault tolerance, Application specific Network on Chip, topology design.

# Introduction

Recent advances in technology have enabled to integrate millions of transistors on a single chip, so much so that various components of a computer can now be integrated on a single chip. This is called a System-on-chip (SoC). For communication between the components, conventionally a bus topology has been used. But this can lead to saturation very fast and is not scalable. The alternative is called network-on-chip (NoC) where the bus is replaced by several smaller interconnections. For generic NoC, symmetric topologies like the mesh are widely used. 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 [4]. It enumerates 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 additional links and routers to the existing topology and create alternate paths between routers and cores. The proposed solution in this paper generates a fault tolerant topology from the given application specific NoC with minimum latency and hardware overhead.

The paper is organized as follows. The section 2 of this paper lists down related literature. The section 3 gives formal definition to the input, output and all the terms and metrics used. Sections 4 and 5 describes some non-fault tolerant topologies for setting benchmarks. Section 6 describes De Bruijn’s graph. Section 7 explains the deterministic algorithm (called poorest neighbour algorithm) for fault tolerant topologies. In sections 8 and 9 the various tests, observations and results are tabled. Finally, section 10 concludes the paper.

# Related Work

[2] presents an algorithm for generation of a irregular link fault tolerant topology for Application Specific NoC designs. They discuss how designed NoC topology allows different routing path 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 a random irregular nonfault tolerant topologies and adding additional links to make them fault tolerant. The paper uses Ring topology as their baseline metric for network resource consumption. A theoretical maximum and minimum are calculated for number of links and routers. The algorithm runs for multiple iterations where in every iteration, a non-fault tolerant topology is randomly generated and links are added to it to make fault tolerant. After a series of iterations, a set of topologies which also includes Ring topology, are generated and the topology with minimum communication cost is chosen as the final topology. 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. The algorithm proposed in this paper is a probabilistic algorithm where a fault tolerant topology is not always guaranteed to be generated.

The algorithm proposed in this paper is a deterministic algorithm which always generates a fault tolerant topology for a given input topology.

In [3], 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 with 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.

# Problem Definition

Our aim is given the information of the application such as the number of cores and the communication bandwidth requirement between each pair of cores, to create topologies which have 1) cores communicating with each other through at least two distinct paths, 2) have minimum latency and energy consumption and 3) minimum 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 at the ends of it would be connected through some alternate path. Thus, even if the link fails, the routers will not get cut off from each other. Also, each core must have access



Fig. 1. Sample input graph

to at least two routers. Thus, even if one router fails, the core will not get cut off from the rest of the network.

The application specific information required is given with the help of a Core Flow Graph as defined in [2].

**Definition 1.** A Core Flow Graph (CFG) is a weighted graph G(*N, E*), where each vertex *ni* ∈ *N* represents a core (i.e., a node) in the application, and each edge *eij* ∈ *E* represents a dependency between two cores *ni* and *nj*. The amount of data transfer between *ni* and *nj* is represented by weight *wij* for all *eij* and is given in bits per second.

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.

**Definition 2.** *Topology graph* (TG) is a connected graph T(*R, L*) where *ri* ∈ *R* represent the routers and *lij* ∈ *L* represent the links between the routers. Here, we assume that each router *r* is mapped to node *ni*. Here, we are assuming there is only one core attached to each router, since the size of a core is very large compared to that of a router.

The performance in of a topology in terms of latency and energy consumption depends on the communication cost.



Fig. 2. A sample topology

**Definition 3.** Communication cost: In the Core Flow Graph, for every edge *eij*, the communication cost *cij* associated with it is the product of its weight *wij* and the number of hops from *ri* to *rj* (denoted by *hij*). The total communication can then be defined as a sum of costs *cij* associated with all the edges. This can be succinctly described in the two following equations:

The primary goal of this paper is to create fault tolerant topologies. Here, we give the formal definition of fault tolerance.

**Definition 4.** Link fault tolerance: The link *lij* is said to be fault tolerant if the routers *ri* and *rj* have an alternate path of communication other than the link *lij*. The link fault tolerance of a topology is defined as the percentage of such links in the topology.

**Definition 5.** The Degree of a router is the number of links connected to it.

# Minimum Spanning Tree

A strictly nonfault tolerant topology is generated here, for setting a benchmark for the minimum number of links and routers needed. Here, there are no alternate paths for any two given routers. Since the objective here is to minimize the total communication cost, we try to minimize the number of hops for the edges with highest weights (or bandwidth). The algorithm proposed to achieve this is as follows:

**Step 1:** From input graph G(*N, E*), sort all edges eij in the descending order of the weight *wij* associated with the respective edges to create the list L(*eij*).

**Step 2:** Add a router *ri* in the topology graph T(*R, L*) for each node *ni*.

**Step 3:** For the first edge *eij*in L(*eij*), check if a path exists which connects routers *ri* & *rj*.

**Step 4:** If yes, proceed to the next step. Otherwise, connect the routers with a new link *lij*.

**Step 5:** Remove eij from L(*eij*) and if L(*eij*) is not yet empty, repeat from **Step 2.**

**Step 6:** Return the resultant topology T(*R, L*).

This algorithm produces topologies that look like a tree with multiple branches. For example, when applied to the graph in fig 1, the resultant topology is shown in fig 3.

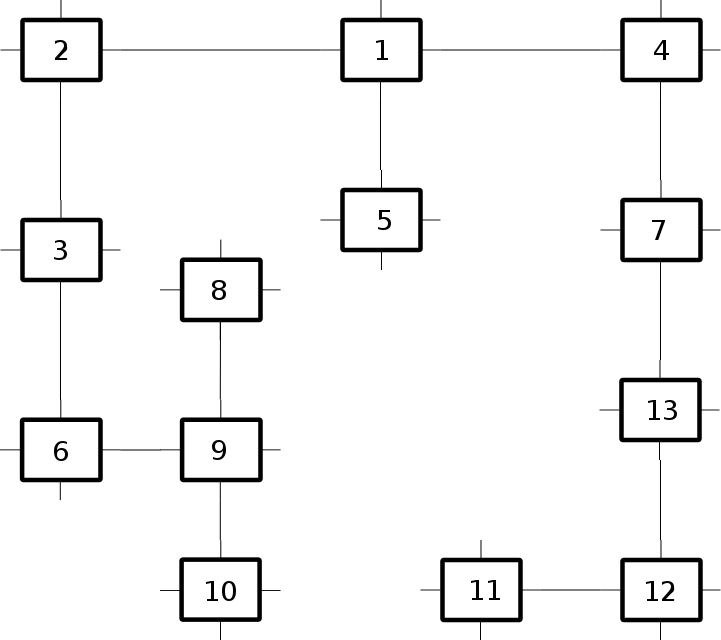


Fig. 3. Minimum spanning tree

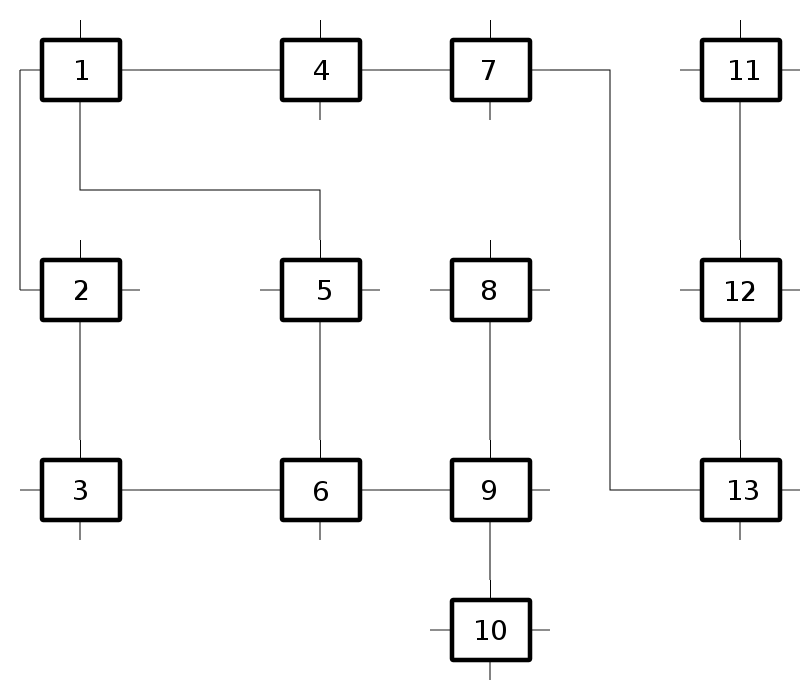


Fig. 4: Example of the native algorithm

This topology is strictly nonfault tolerant and minimizes the number of routers and links needed for the given application requirements, thus reducing hardware cost. That is why, we will use this topology as a benchmark for the link and router hardware overheads while evaluating the future topology generation algorithms.

# 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 core flow graph C(*N, E).* The following algorithm does exactly that:

**Step 1:** Add a router *ri* for each node *ni*.

**Step 2:** For each edge *eij* create a link *lij* connecting routers *ri* & *rj*.

**Step 3:** Return the resultant topology.

Apart from being simple, this topology also has the lowest communication cost theoretically possible, since *hij* = 1 for all edges *eij*. Hence, this sets a benchmark for the minimum communication cost. Any topology that has this as a part of it, will have the same communication cost. Hence, this topology is intended to be the basis for the next

algorithm.

When applied to the graph in fig 1, the topology looks like in fig 4.

# De-Bruijn’s algorithm

De Bruijn’s graph is a popular algorithm which is widely used in the field of Bioinformatics. Papers such as [3] suggest De Bruijn’s graph as an alternate topology and compared its energy consumption with popular regular topologies like mesh and torus. De Bruijn’s graph is inherently link fault tolerant and has lower energy consumption than regular topologies like mesh as established by [3]. The generated De Bruijn’s graph is link fault tolerant and may also contain more than one alternate paths for a pair of nodes. In the later sections of the paper, De Bruijn’s performance is compared against the Poorest neighbour and other algorithms which are proposed in other sections. The algorithm looks like this:

**Step 1:** The number of nodes in core flow graph G(*N, E*) is *N*, hence add *R* = *N* routers in the topology graph T(*R, L*). Connect routers *r1* and *r2*.

**Step 2:** Set variables *p* = 2 and *c* = *p* + 1 = 3.

**Step 3:** Connect routers *rp* and *rc*. Increment *c*.

**Step 4:** If *c* is odd, increment *p*. Repeat step 3 if *c* < *R*.

**Step 5:** Set *p* = *R* and *c* = *p* - 1. Set variable *e* = *N* % 2.

**Step 6:** Connect routers *rp* and *rc*. Decrement *c*.

**Step 7:** If *c* % 2 = *e*, decrement *p*. Repeat step 3 if *c* > 0.

When this algorithm is applied to the graph in fig 1, the topology looks like in fig 5.

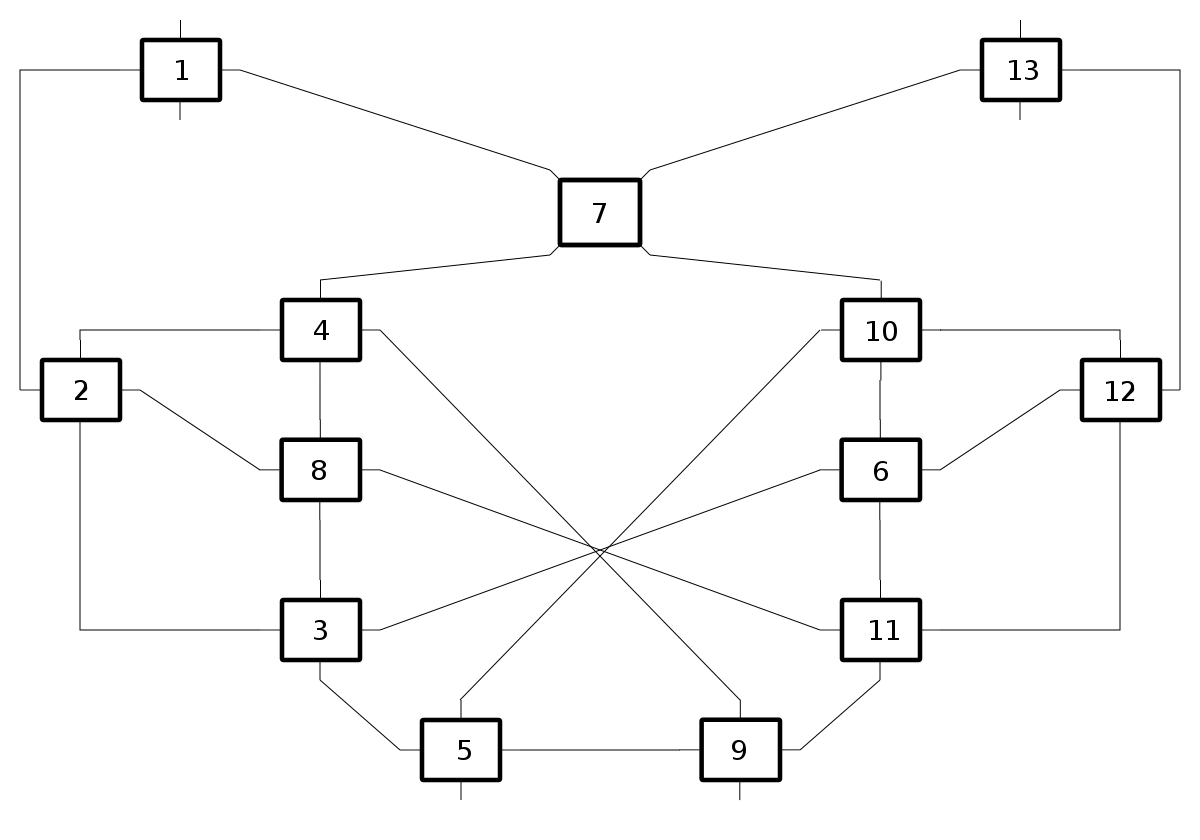


Fig. 5. Example of De Bruijn’s topology

# Poorest Neighbour Algorithm

This algorithm aims to provide link fault tolerance. As mentioned earlier, it starts from the native topology described above. The algorithm is as follows:

**Step 1:** Generate the native topology *T(R, L)*.

**Step 2:** From input graph *G(N, E)*, sort all edges *eij* in the descending order of the weight *wij* associated with the respective edges to create the list *L(eij)*.

**Step 3:** For each link *lij* corresponding to *eij* in *L(eij)*, check if there is an alternate path between the routers *ri* and *rj*.

**Step 4:** If yes, jump to Step 2 for the next link.

**Step 5:** If no, compare the degree of routers *ri* and *rj*. Now we refer the router with the smaller degree as *rs* and the larger degree as *rl*.

**Step 6:** Consider the router *rl*. From the list of all the Neighbouring routers of *rl*, select the router *rn* of the lowest degree.

**Step 7:** Connect routers *rn* and *rs* with link *lns*.

**Step 8:** Jump to Step 2 for the next link.

**Step 9:** If no links are left, return the resultant topology.

The aim of this algorithm is to provide 100% link fault tolerance with the minimum communication cost. Since this topology contains the native topology as a part of it, the communication cost is identical to that of the native topology, which is the theoretical minimum. When this

algorithm is applied to the graph in fig 1, the resultant topology is shown in fig 6.

1. Fault Tolerance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Serial No.** | **Number of Nodes** | **Fault Tolerance** | | |
| ***Poorest Neighbour (without solution)*** | ***Poorest Neighbour***  ***(with solution)*** | ***De Bruijn’s*** |
| 1 | 8 | 100 | 100 | 100 |
| 2 | 24 | 100 | 100 | 100 |
| 3 | 30 | 86.20689392 | 100 | 100 |
| 4 | 64 | 100 | 100 | 100 |
| 5 | 64 | 99.00990295 | 100 | 100 |
| 6 | 128 | 100 | 100 | 100 |

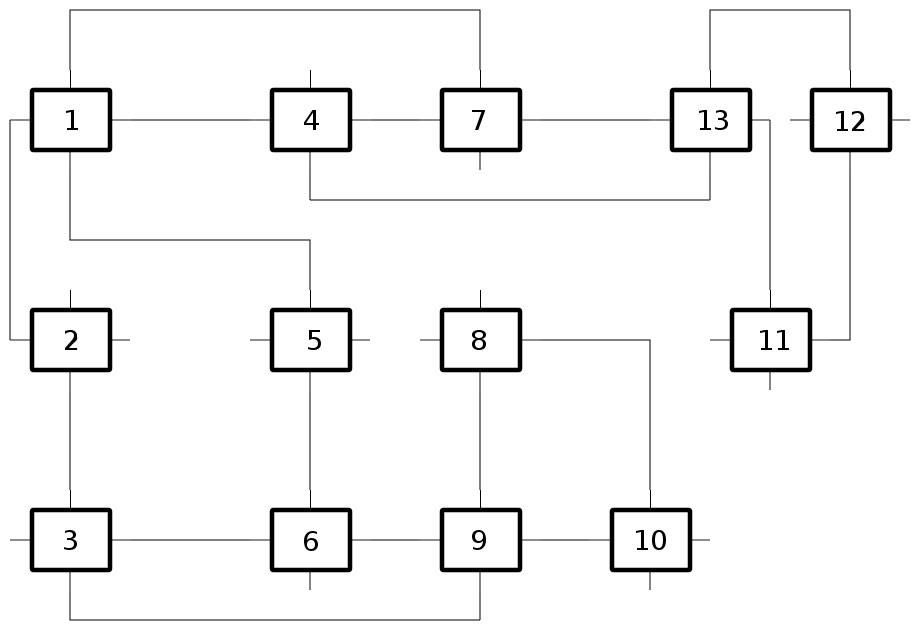


Fig. 6. Example of the poorest neighbour algorithm

# Observations

The main objective of this paper is to create a topology with 100% fault tolerance. However, as seen in table 1, in some cases, the poorest neighbour algorithm does not provide 100% fault tolerance. We investigate the cause and a solution for this problem in this section. With the solution proposed here, the poorest neighbour algorithm provides 100% fault tolerance in all cases.

*A. Fail Case: Isolated Pair*

The poorest neighbour algorithm fails to provide 100% fault tolerance in the following scenario.

Consider a disconnected graph like the one shown above. Suppose this is the input to our algorithm. The ‘poorest neighbour’ algorithm first checks if the particular link has an alternate path for connecting the two end nodes of that link. If yes, then it does nothing. If no, then the algorithm seeks to connect one of the nodes with a neighbour of the other node. But in the above graph, consider the edge connecting nodes *n7* and *n8*. None of these nodes have a neighbour. Hence, there is no one to connect either of the concerned nodes. No connection is thus made, and the resulting topology is not fault tolerance.

*B. Proposed solutions*

In this case, the proposed solution is to modify the algorithm to connect the nodes to the nodes in the rest of the graph with the lowest degree. Here, the nodes *n1* and *n6* are of degree 2. Thus, if we choose node n6, the resulting topology would look something like fig. 8.

However, this increases the number of ports of the

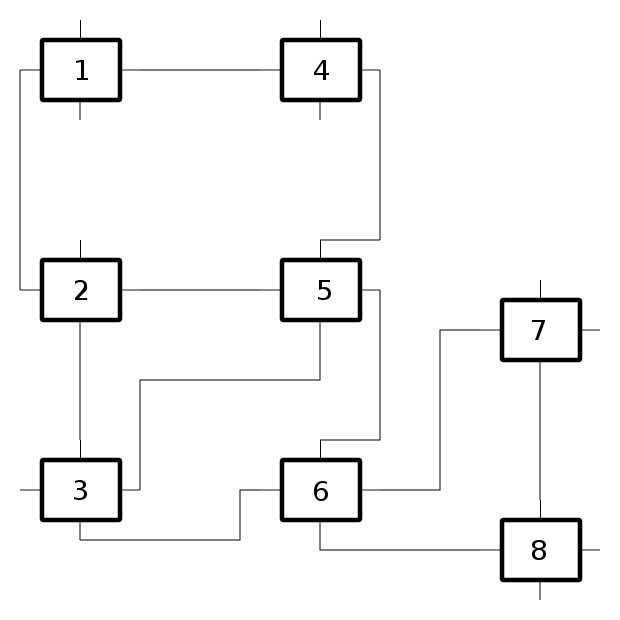


Fig. 8. Proposed solution

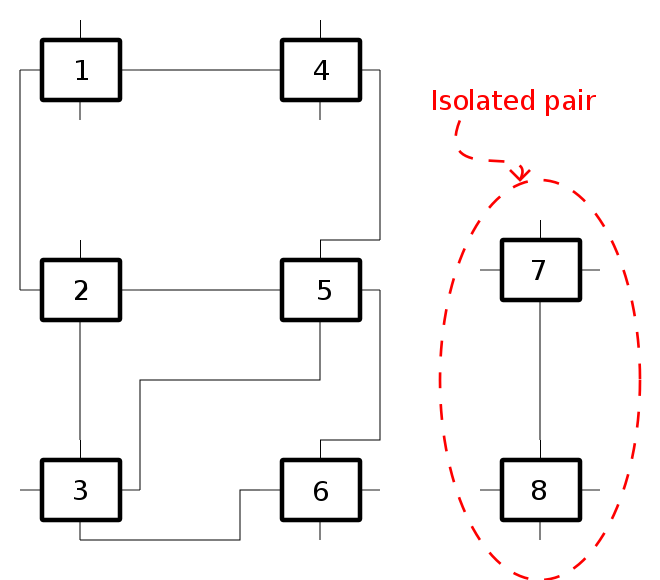


Fig. 7. Graph with an isolated pair.

chosen node by 2. Thus, the degree of node *n6* now becomes 4. This would become a problem if the degree goes beyond the maximum number of ports of a router. For example, if the minimum degree in the above example was 3 instead of 2 and the maximum number of ports a router has is 4, it would look like fig. 9.

Here, we can connect only one node to *n6*. One solution would be to connect the other node to the poorest neighbour of *n6* which refers to the node *n3*. Thus, the resulting fault tolerant topology would look like shown in fig. 10.

# Test results

Now, let us compare all the topology generation methods. Every algorithm is evaluated over the following criteria:

1. Communication cost
2. Number of links

The table 2 compares the communication cost of the topologies generated by Poorest neighbour, native and De Bruijn’s algorithms against the number of nodes in the input graph.

**Inferences:** From columns 3 and 4, it is clear 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 nodes contained in the core input graph. Hence, poorest neighbour algorithm is highly scalable for fault tolerant topology generation.

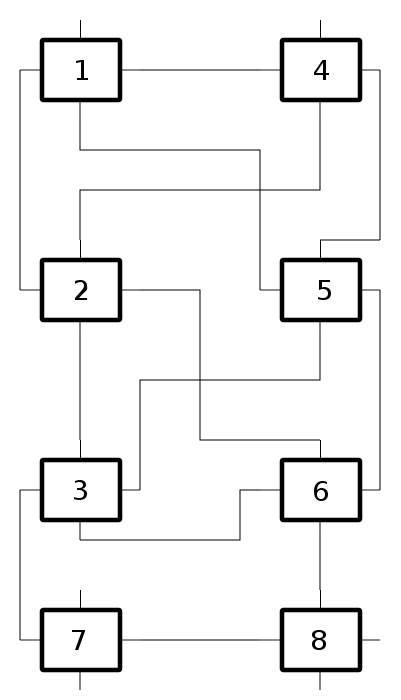


Fig. 10. Proposed solution to the alternate scenario

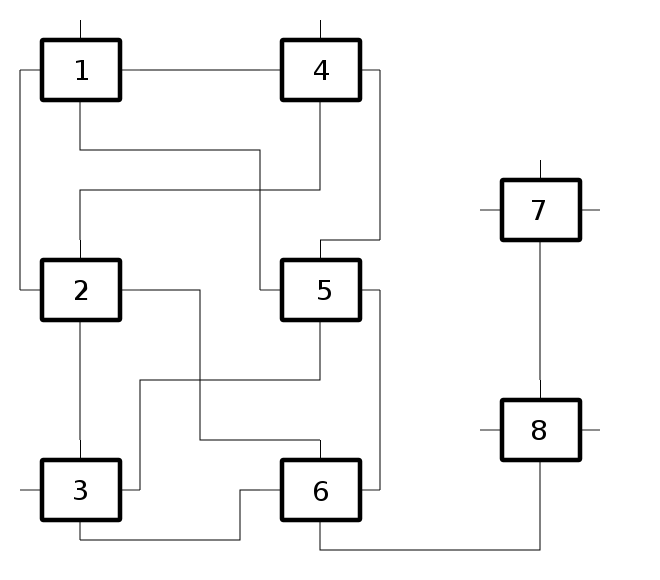


Fig. 9. Alternate scenario

1. Hardware required – number of links

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Reference No.** | **Number of nodes** | **Number of links** | | |
| ***Minimum Spanning Tree*** | ***Poorest Neighbour*** | ***De Bruijn’s*** |
| 1 | 8 | 7 | 9 | 13 |
| 2 | 24 | 20 | 33 | 44 |
| 3 | 30 | 21 | 37 | 56 |
| 4 | 64 | 63 | 108 | 125 |
| 5 | 64 | 59 | 103 | 125 |
| 6 | 128 | 127 | 234 | 253 |
| 7 | 128 | 127 | 211 | 253 |
| 8 | 128 | 127 | 202 | 253 |
| 9 | 128 | 126 | 265 | 253 |
| 10 | 128 | 126 | 229 | 253 |

1. Communication Cost

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Reference No.** | **Number of nodes in CFG** | **Communication Cost** | | |
| ***Native*** | ***Poorest Neighbour*** | ***De Bruijn’s*** |
| 1 | 8 | 576 | 576 | 704 |
| 2 | 24 | 131 | 131 | 401 |
| 3 | 30 | 88 | 88 | 364 |
| 4 | 64 | 24608 | 24608 | 423165 |
| 5 | 64 | 6063 | 6063 | 93454 |
| 6 | 128 | 55404 | 55404 | 1777704 |
| 7 | 128 | 11963 | 11963 | 380437 |
| 8 | 128 | 26281 | 26281 | 892604 |
| 9 | 128 | 163837 | 163837 | 5437165 |
| 10 | 128 | 137777 | 137777 | 4667287 |

From column 4 and 5 of the table, we can clearly infer that for a given input graph, the communication cost of topology generated by poorest neighbour is very less compared to the topology generated by De Bruijn’s algorithm. Also with the increase in the number of nodes, the communication cost of the De Bruijn’s topology is increasing as compared to Poorest Neighbour. Hence, it is not very scalable as compared to Poorest neighbour.

The table 3 compares the number of links required for the topologies generated by nonfault Tolerant Tree, Poorest neighbour and De Bruijn’s algorithms.

**Inferences**: The number of links required is least for the Non-fault tolerant tree topology. Even with increase in the number of nodes there is no significant increase in number of links in minimum spanning tree topology. Lesser the number of links, lower the hardware cost. Hence, minimum spanning tree topology requires the least hardware requirements amongst the three algorithms. The number of links required for topologies generated by De Bruijn’s and Poorest neighbour is more or less the same.

# Conclusion

The non-fault tolerant topology generation algorithm described in section 4 had the minimum hardware cost. The native topology set the benchmark for communication cost. The Poorest Neighbour algorithm provided link fault tolerance while meeting the communication cost benchmark set by the native topology and is found to be better than De Bruijn’s algorithm in this aspect. The hardware costs of Poorest Neighbour algorithm are on an average lower than that of the De Bruijn’s algorithm. However, they cannot match the cost effectiveness of the minimum spanning tree topology.

# References

1. 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
2. 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
3. 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.
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. Universität Stuttgart - SS09, Stuttgart. July 7, 2009. Seminar Report, p. 12 - 17.