Average Packet Latency and Link Fault Tolerant Routing Algorithms in Tree-Like Interconnection Networks

Wang, Yaodong

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
17
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2022-03-24
(URL)
https://doi.org/10.15002/00025257

Average Packet Latency and Link Fault Tolerant Routing Algorithms in Tree-Like Interconnection Networks

Yaodong Wang
Graduate School of CIS
Hosei University
yaodong.wang.7y@stu.hosei.ac.jp

***Abstract*—In the first part of this research, we propose two hybrid topologies named k-Cube k-Ary n-Tree (CAT) and Mirrored k-Cube k-Ary n-Tree (MiCAT), based on fat-trees and hypercubes. We evaluate the path diversity, cost, performance, and average packet latency of CAT and MiCAT. The results show that CAT and MiCAT can save up to 87% switches and 80% links in a large-scale parallel system, for example, if $k = n = 8$, compared to fat-trees, and meanwhile, both CAT and MiCAT have higher path diversities than fat-trees. The second part gives four link fault tolerant routing algorithms in Mirrored k-ary n-tree (MiKANT) interconnection networks and evaluates their performance through simulations. In addition, the performance of the combined algorithms is also evaluated.**

1. Introduction

Nowadays, more and more compute nodes are used in supercomputers and data centers that require a large-scale interconnection network. Thus the trade-off of the cost and the performance for the interconnection networks becomes a more and more important issue.

In order to find a balance between the hardware cost and performance, a wide variety of interconnection topologies were proposed. Among these topologies, fat-tree [1] is one of the most popular topologies of the interconnection networks in current large-scale supercomputers. In a k-ary n-tree [3], a special case of fat trees, the switch radix is $2k$, and the number of switch levels is n. In order to reduce the hardware cost of the switches and links and improve the communication performance of the fat-tree networks, a Mirrored K-Ary N-Tree (MiKANT) network [2] was proposed. It is a variant of fat-trees aimed at reducing hardware cost and packet latency. As the scale of MiKANT becomes large, the probability of switch or link failure increases. Fault tolerance is one of the important issues of the interconnection networks for large-scale parallel computers. [6] gave four link fault tolerant routing algorithms in Mirrored k-ary n-tree (MiKANT) interconnection networks and evaluated their performance through simulations.

However, in each leaf switch of the k-ary n-tree or MiKANT, there are only k compute nodes connected to it. A large-scale parallel computing system, which consists of a huge number of compute nodes, requires a large-scale interconnection network that consists of a huge number of switches and links. In such a case, we propose two hybrid topologies: k-Cube k-Ary n-Tree (CAT) and Mirrored k-Cube k-Ary n-Tree (MiCAT) [5] that are based on a fat-tree and small size hypercubes [4].

* Supervisor: Prof. Yamin Li

For a CAT/MiCAT, the parameter k represents both the dimension of the hypercube and the switch arity to ensure that the radix of this architecture will be equal to $2k$. There are $2^k - 1$ hypercube switches connected to each of the leaf switches. And k compute nodes are connected to each of the hypercube switches. We analyze the hardware cost, performance, path diversity, and packet latency of CAT and MiCAT. The results show that CAT and MiCAT achieve higher path diversity at a lower hardware cost than the classical fat-trees.

2. CAT and MiCAT Topologies

This section introduces the CAT and MiCAT topologies.

2.1. K-Cube K-Ary N-Tree

The CAT is constructed based on hypercube and k-ary n-tree, denoted as CAT(k, n). A CAT(k, n) can be constructed by replacing k compute nodes connected to each switch in level 0 with a k-cube and each switch of k-cube connects k compute nodes. That is, $2^k - 1$ switches and $k(2^k - 1)$ compute nodes are connected to each switch in level 0. Each switch of CAT is labeled as $\langle L, D, C\rangle$; Where L indicates level, D indicates the fat-tree ID, and C indicates the hypercube ID. In a CAT(k, n), we call the level 0 switch *transit switch*. Each switch of k-cube part is labeled as $\langle 0, D, C\rangle$, where $C = C_{k-1}, C_{k-2}, \ldots, C_1, C_0$ with $C_i \in \langle 0, 1\rangle$ and $C \neq 0, \ldots, 0$, a transit switch is labeled as $\langle 0, D, 0\rangle$. Suppose two switches $U = \langle 0, D_U, C_U\rangle$ and $V = \langle 0, D_V, C_V\rangle$ are connected to a same transit switch, then $D_U = D_V$. Because a transit switch $\langle 0, D_{n-2}, \ldots, D_0, 0, \ldots, 0\rangle$ has a k-cube address $C = 0, \ldots, 0$, it connects to switches $\langle 0, D_{n-2}, \ldots, D_0, C_{k-1}, \ldots, C_0\rangle$ with $C_{k-1} + \cdots + C_0 = 1$. In the routing within a k-cube part, the label D will not change. Similarly, the label C will not change when routing in the fat-tree part.

Fig. 1 shows a CAT(3, 3). There are $k^{n-1} = 9$ transit switches and each transit switch connects a 3-cube. In a 3-cube, there are $2^k = 8$ switches (including the transit switch). Except for the transit switch, each 3-cube switch connects 3 compute nodes. Totally there are $9 \times (8-1) \times 3$, or 189 compute nodes.

2.2. Mirrored K-Cube K-Ary N-Tree

The MiCAT, denoted as MiCAT(k, n), is constructed based on MiKANT(k, n) and k-cube. Each switch of a MiCAT(k, n) is labeled as $\langle G, L, D, C\rangle$, where G indicates a group, L, D, c is the same as CAT. For example, a

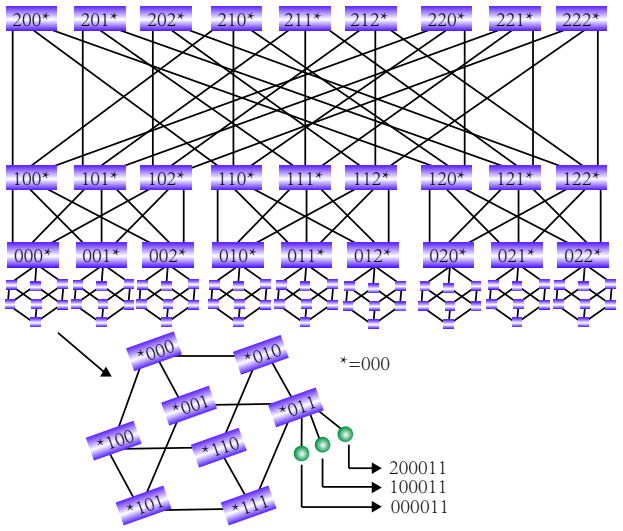

Figure 1. A 3-cube 3-ary 3-tree

MiCAT$(4,3)$ has $2k^{n-1} = 32$ transit switches. The group 0 and the stage 1 switches of group 1 form a CAT$(4,3)$; and the group 1 and the stage 1 switches of group 0 form another CAT$(4,3)$. A switch W

$$\langle G, L, D_{n-2}, \ldots, D_{L+1}, D_L, D_{L-1}, \ldots, D_0, C_{k-1}, \ldots, C_0 \rangle$$

connects to switches

$$\langle G, L{+}1, D_{n-2}, \ldots, D_{L+1}, *', D_{L-1}, \ldots, D_0, C_{k-1}, \ldots, C_0 \rangle$$

if $0 \le L < n-2$; otherwise $(L = n-2)$ to switches

$$\langle \overline{G}, L, *', D_{n-3}, \ldots, D_1, D_0, C_{k-1}, \ldots, C_0 \rangle$$

where $\overline{G}$ is the bit-inversion of G, $*'$ is any value for $*' \in \{0, 1, \ldots, k-1\}$. When the source node and destination node are of a same group (group 0 or 1), MiCAT(k,n) acts as the same as CAT(k,n).

Table 1 summarizes the topological properties of k-cube, k-ary n-tree, MiKANT(k,n), CAT(k,n), and MiCAT(k,n). From the table, we can see that both the CAT and MiCAT can connect more compute nodes with fewer switches and links.

3. Performance Evaluation

In this section, we evaluate the cost, performance, and path diversity of CAT(k,n) and MiCAT(k,n).

3.1. Cost Ratio

The k-ary n-tree has k^n nodes and nk^{n-1} switches. That is, one node requires nk^{n-1}/k^n switches. A CAT(k,n) has $(2^k-1)k^n$ nodes and $(n+2^k-1)k^{n-1}$ switches. The switch cost ratio of a CAT(k,n) to the k-ary n-tree is

$$\frac{(n+2^k-1)k^{n-1}/(2^k-1)k^n}{nk^{n-1}/k^n} = \frac{n+2^k-1}{n(2^k-1)}$$

Similarly, the link cost ratio of a CAT(k,n) to the k-ary n-tree is

$$\frac{(3\times 2^{k-1}+n-2)k^n/(2^k-1)k^n}{nk^n/k^n} = \frac{3\times 2^{k-1}+n-2}{n(2^k-1)}$$

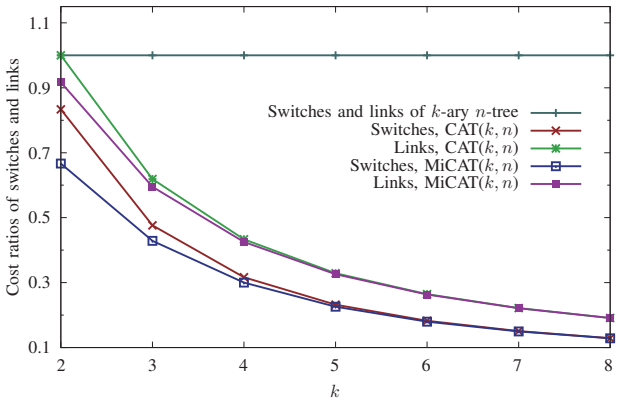

Figure 2. Cost ratios of switches and links to k-ary n-tree

We plot the switch and link cost ratios of the CAT(k,n) to the k-ary n-tree in Fig. 2. For convenience, we set $n = k$ in both figures. We can see that when $n = k \ge 3$, both the CAT(k,n) and MiCAT(k,n) have a lower cost of the switches and links than the k-ary n-tree. For $k = n = 8$, the CAT saves 87.11% switches and 80.88% links compared to k-ary n-tree. The MiCAT saves 87.16% switches and 80.91% links compared to k-ary n-tree.

3.2. Relative Cost Performance

In order to make a good tradeoff between the cost and performance for CAT(k,n) and MiCAT(k,n), we define a *relative cost performance* (RCP) to the hypercube as below:

$$RCP_1 = \frac{2k \times (2n+2k)}{(\log_2(N_1/p)+p) \times (\log_2(N_1/p)+2)}$$

$$RCP_2 = \frac{2k \times (2n+2k)}{(\log_2(N_2/p)+p) \times (\log_2(N_2/p)+2)}$$

where the RCP_1 and RCP_2 means the RCP of the CAT(k,n) and RCP of the MiCAT(k,n), respectively; $2k$ is the radix of the CAT(k,n) and MiCAT(k,n) which affects the hardware cost; $2n+2k$ is the diameter of the CAT(k,n) and MiCAT(k,n) which affects the communication performance; $(\log_2 N/p)$ is the dimension of the hypercube; and p indicates the number of ports in a router for connecting compute nodes. Both the radix and diameter of k-cube are k, but in real implementations, there are p ports in a router for connecting compute nodes. That is, the real router radix is $k+p$. The diameter of CAT(k,n) and MiCAT(k,n) contains the links that connect compute nodes to switches. To make a fair comparison, we let the diameter of k-cube be $k+2$.

Fig. 3 and Fig. 4 illustrate the RCPs of CAT(k,n) and MiCAT(k,n), respectively, to the n-cube for $2 \le n \le 7$ with $p = 1$. For a given n, we change the value of k to implement the system with different sizes. The lower values in the curves mean that the systems achieve higher performance at lower hardware costs. For example, when we build a 491,520-node system with MiCAT$(4,7)$, the RCP is 0.4676. For a given size of a system, we can select suitable k and n based on the figures so that the system will have a lower RCP.

TABLE 1. COMPARISON OF NETWORK TOPOLOGICAL PROPERTIES

Parameters	HC(k)	k-ary n-tree	MiKANT(k,n)	CAT(k,n)	MiCAT(k,n)
Nodes	2^k	k^n	$2k^n$	$(2^k-1)k^n$	$2(2^k-1)k^n$
Switches	2^k	nk^{n-1}	$(2n-2)k^{n-1}$	$(n+2^k-1)k^{n-1}$	$(2n-4+2^{k+1})k^{n-1}$
Links	$k2^{k-1}$	nk^n	$(2n-1)k^n$	$(3\times 2^{k-1}+n-2)k^n$	$(3\times 2^k+2n-5)k^n$
Radix/Degree	k	$2k$	$2k$	$2k$	$2k$
Diameter	k	$2n$	$2n$	$2n+2k$	$2n+2k$
Average distance	$\frac{k}{2}$	$2n-\frac{2}{k-1}$ $+\frac{2}{(k-1)k^n}$	$2n-\frac{1}{k-1}$ $+\frac{1}{(k-1)k^n}-\frac{1}{2}$	$k+2n+\frac{2}{k-1}$ $-\frac{1}{k^{n-1}}(\frac{k}{k-1}+\frac{k}{2}+2)$	$k+2n-\frac{1}{2}+\frac{4}{k-1}$ $-\frac{1}{2k^{n-1}}(\frac{k}{k-1}+\frac{k}{2}+2)$

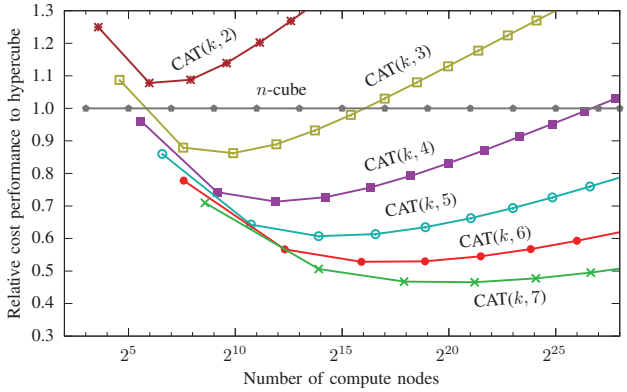

Figure 3. RCP comparison of CAT(k,n)

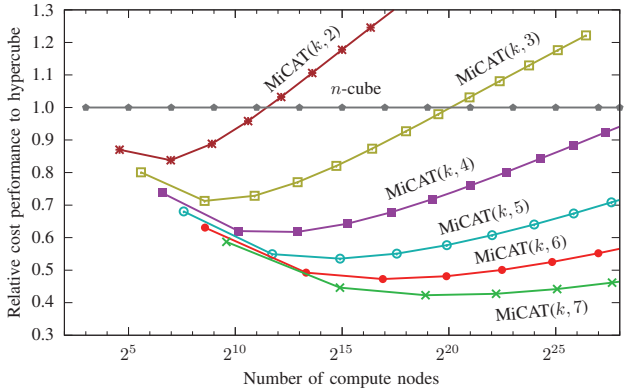

Figure 4. RCP comparison of MiCAT(k,n)

3.3. Path Diversities

Path diversity is an important attribute of a topology. High path diversity means that there are many paths between the source and destination nodes for sending packets. This section defines a method to calculate the path diversity of the CAT(k,n) and MiCAT(k,n), and compares it to that of k-ary n-tree and MiKANT(k,n). The path diversity (PD) of a network is defined as below.

$$PD = \frac{\tilde{P}}{N}$$

where $\tilde{P}$ is the average number of shortest paths and N is the number of nodes in the system. The path diversity

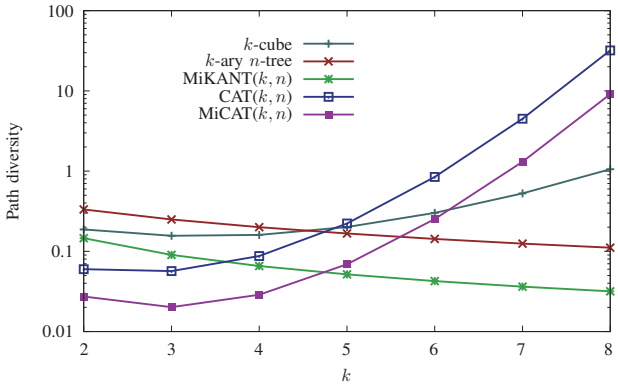

Figure 5. Path diversities

PD of a k-cube is

$$PD_{cube} = \frac{\tilde{P}_{cube}}{N_{cube}} = \sum_{i=1}^{k} \frac{k!}{i! \times 4^k}$$

And, the path diversity PD of a k-ary n-tree is

$$PD_{kant} = \frac{\tilde{P}_{kant}}{N_{kant}} = \frac{1-1/k^{2n-2}}{k+1}$$

Similarly, The path diversity PD of a MiKANT(k,n) is

$$PD_{mikant} = \frac{\tilde{P}_{mikant}}{N_{mikant}} = \frac{1-1/k^{2n-2}}{4(k+1)} + \frac{1}{4k^2}$$

The path diversity of CAT can be written by the path diversity of the hypercube and k-ary n-tree. The path diversity PD of a CAT(k,n) is

$$PD = \frac{\tilde{P}_{cat}}{N_{cat}} = \frac{\tilde{P}_{cube}}{k^{n-1}}[\tilde{P}_{cube}\tilde{P}_{kant}(k^{n-1}-1)+1]/N_{cat}$$

Similarly, the path diversity of MiCAT can be written by the path diversity of the hypercube and MiKANT(k,n). The path diversity PD of a MiCAT(k,n) is

$$PD = \frac{\tilde{P}_{cube}}{2k^{n-1}}[\tilde{P}_{cube}\tilde{P}_{mikant}(2k^{n-1}-1)+1]/N_{micat}$$

Fig. 5 plots the path diversities of the k-cube, k-ary n-tree, MiKANT(k,n), CAT(k,n), and MiCAT(k,n) with $n=8$. We can see that the path diversities of the CAT(k,n) and MiCAT(k,n) are better than that of k-ary n-tree and MiKANT(k,n) when $k \geq 5$.

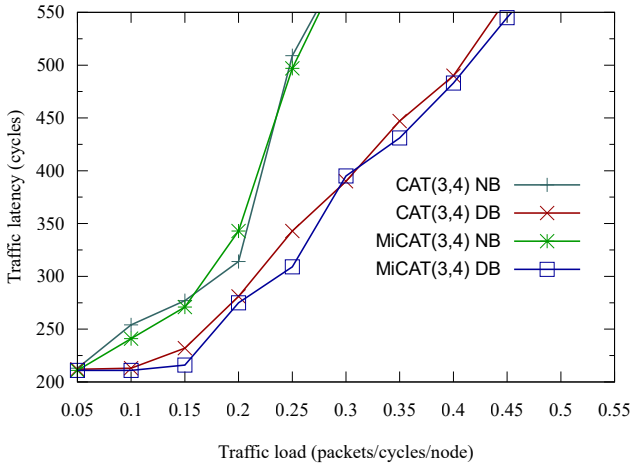

Figure 6. Average packet latencies

3.4. Packet Latency

We have evaluated the average packet latencies of CAT and MiCAT with $k = 3$ and $n = 4$ through simulation. CAT and MiCAT have the number of nodes which is $(2^k - 1)k^n = 567$ and $2(2^k - 1)k^n = 1134$, respectively. The reason why use $k = 3$ and $n = 4$ is that in CAT and MiCAT topologies, the bottleneck of the packet latency always happens in the hypercube part, we need to use the same size of the hypercube part to make them fair; furthermore, a large scale interconnection network require to use a big amount of memory and we did it in an ordinary personal computer.

We evaluate the simulation on a clock cycle-by-cycle by uniform pattern. In uniform traffic, packet destination addresses are randomly assigned. For each packet in one clock cycle, the packet can be sent to another switch/node if the switch has a usable buffer, otherwise the packet will wait for one clock cycle. We set traffic load λ in the range of 0.05 and 0.55, stepped by 0.05. That is, in each clock cycle, there are $N \times \lambda$ compute nodes that will send the packet to their destination nodes where N is the number of nodes in the system. The simulation terminates when each destination node has received an average of 200 packets.

Fig. 6 illustrates the packet latencies of CAT$(3, 4)$ and MiCAT$(3, 4)$, respectively. The vertical axis represents the average packet latency in clock cycles and the horizontal axis represents the traffic load λ. The NB represents the normal buffer, the normal buffer means that each switch has the same buffer depth as 8; DB represents the double buffers, double buffer means that the level 0 switches have the double buffer depth of other switches which is 16. We can see that the double buffer depth has lower average packet latencies than normal buffer depth in both CAT and MiCAT obviously.

4. Link Fault Tolerance in MiKANT

The purpose of fault tolerant routing is to enable a system to continue operating properly in a high probability with switch or link faulty. A link faulty means that the ports of the switches connected by the link cannot be used for passing the packet. In this section, we give four link fault tolerant routing algorithms.

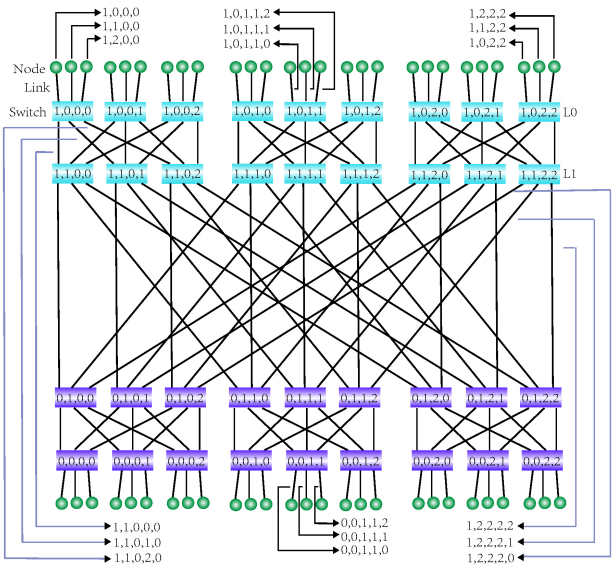

Figure 7. A Mirrored 3-ary 3-tree

Algorithm 1 MiKANT_Routing (*packet*)

Input: $packet = \langle T, data \rangle$; /* received packet which will be sent to T */
$W = \langle G_W, L_W, W_{n-2}, ..., W_1, W_0 \rangle$; /* my switch ID */
$T = \langle G_T, T_{n-1}, T_{n-2}, ..., T_1, T_0 \rangle$; /* destination node ID */
if $(G_W \neq G_T)$ /* W, T: different groups */
  send *packet* to $T^+_{L_W}$ port; /* increasing level */
else /* W, T: same group */
  if $(W_{n-2}, ..., W_{L_W} \neq T_{n-2}, ..., T_{L_W})$ /* going to NCA */
    send *packet* to $T^+_{L_W}$ port; /* increasing level */
  else /* going to destination from NCA */
    if $(L_W > 0)$ /* not a level 0 switch */
      send *packet* to $T^-_{L_W - 1}$ port; /* decreasing level */
    else /* a level 0 switch */
      send *packet* to T^-_{n-1} port; /* to destination node */
    endif
  endif
endif

4.1. Routing Algorithm in MiKANT

The routing algorithm of MiKANT is based on the destination compute node ID and switch IDs. The output port in each switch is selected based on the current switch ID and destination node ID. Each switch has $2k$ ports in a MiKANT(k, n). In each group, the ports on the side near to compute nodes are labeled with $0, 1, \ldots, k-1$; the ports on the other side are labeled with $k, k+1, \ldots, 2k-1$. We use $W = \langle G_W, L_W, W_{n-2}, ..., W_1, W_0 \rangle$ to denote the current switch ID. The packet received by W contains the destination node ID $T = \langle G_T, T_{n-1}, T_{n-2}, ..., T_1, T_0 \rangle$. Based on T, W selects a port and sends the packet through the selected port. These routings are in an *upward phase* in which the packet is sent from the source switch to a nearest common ancestor (NCA) of both the source and destination nodes. Because there are more than one NCA, we can select other ports to send the packet to different NCAs. This is helpful for fault tolerant routing.

The routing algorithm is formally given in **Algorithm 1**, where $T^+_x = T_x + k$ and $T^-_x = T_x$.

4.2. Fault Tolerant Routing Algorithms

The routing algorithm given in **Algorithm 1** finds a shortest path between the source and destination compute nodes. If a link in the path is faulty while the path to the destination node is determined, the shortest path routing algorithm will fail. For example, the current switch is $\langle 0,1,0,0\rangle$ in a MiKANT$(3,3)$ and the destination node is $\langle 1,0,0,0\rangle$. In this case, if the link $\langle 1,2,0,0,0\rangle$ is faulty, the current switch cannot reach the switch $\langle 1,1,0,0\rangle$ in the shortest path. Based on the shortest path routing algorithm, we give four link fault tolerant routing algorithms that select other links for the routing if a faulty link is encountered.

4.2.1. Go-Neighbor-Switch Algorithm. In the previous example, the link $\langle 1,2,0,0,0\rangle$ is faulty. We can send the packet to the neighbor switch $\langle 1,1,0,1\rangle$ or $\langle 1,1,0,2\rangle$ of switch $\langle 1,1,0,0\rangle$. In this way, the current switch can send the packet to the destination node $\langle 1,0,0,0\rangle$ via the lower level switch $\langle 0,1,0,1\rangle$ or $\langle 0,1,0,2\rangle$. We call this the "Go-Neighbor-Switch" algorithm. The current switch can go to the lower level and back to the current level to change to the neighbor switch. The current switch is $\langle 0,1,0,0\rangle$ and destination node is $\langle 1,0,0,0\rangle$. We can select the path like $\langle 0,1,0,0\rangle \rightarrow \langle 0,0,0,1\rangle \rightarrow \langle 0,1,0,1\rangle \rightarrow \langle 1,1,0,1\rangle \rightarrow \langle 1,0,0,0\rangle$ or $\langle 0,1,0,0\rangle \rightarrow \langle 0,0,0,0\rangle \rightarrow \langle 0,1,0,2\rangle \rightarrow \langle 1,1,0,2\rangle \rightarrow \langle 1,0,0,0\rangle$. Generally, for $W = \langle G_W, L_W, W_{n-2}, \ldots, W_1, W_0\rangle$ and $T = \langle \overline{G}_W, T_{n-1}, T_{n-2}, \ldots, T_1, T_0\rangle$, suppose that the link $\langle 1, L_W+1, L_{n-2}, \ldots, L_1, L_0, P\rangle$ is faulty. If $G_W = 1$, P equals T_{n-2} and $(L_{n-2}, \ldots, L_1, L_0 = W_{n-2}, \ldots, W_1, W_0)$. Otherwise, P equals W_{n-2} and $(L_{n-2}, \ldots, L_1, L_0 = T_{n-2}, \ldots, T_1, W_0)$. W will send a packet to $\langle G_W, L_W - 1, W_{n-2}, *, \ldots, W_1, W_0\rangle$, then to $\langle G_W, L_W, W_{n-2}, * - W_{n-3}, \ldots, W_1, W_0\rangle$, and next to $\langle \overline{G}_W, L_W, T_{n-2}, * - W_{n-3}, \ldots, W_1, W_0\rangle$, where $* - W_{n-3}$ are all the element $\in \{0,1,\ldots,k-1\}$ except W_{n-3}.

4.2.2. Go-Down-Level Algorithm. In the downward phase, the path to the destination node is determined. For example, $W = \langle 0,1,0,0\rangle$ in a MiKANT$(3,3)$ wants to send a packet to the destination node $\langle 0,2,0,0\rangle$ via switch $\langle 0,0,0,0\rangle$. If the link $\langle 0,1,0,0,0\rangle$ is faulty, the shortest path algorithm will fail. But W can use other ways to get to the switch $\langle 0,0,0,0\rangle$. For example, $\langle 0,1,0,0\rangle \rightarrow \langle 0,0,0,1\rangle \rightarrow \langle 0,1,0,1\rangle \rightarrow \langle 0,0,0,0\rangle$ or $\langle 0,1,0,0\rangle \rightarrow \langle 0,0,0,2\rangle \rightarrow \langle 0,1,0,2\rangle \rightarrow \langle 0,0,0,0\rangle$. We call this "Go-Down-Level" algorithm. Generally, for $W = \langle G_W, L_W, T_{n-2}, \ldots, T_1, *\rangle$, $T = \langle G_T, *, T_{n-2}, \ldots, T_1, T_0\rangle$, $G_W = G_T$, and $L_W = 1$, if $\langle G_W, L_W, T_{n-2}, \ldots, T_1, *, T_0\rangle$ is a faulty link, W can send the packet to the $\langle G_W, 0, T_{n-2}, \ldots, * - T_0\rangle$, then to $\langle G_W, 1, T_{n-2}, \ldots, * - W_0\rangle$, next to $\langle G_W, 0, T_{n-2}, \ldots, T_1, T_0\rangle$, and finally to the destination node.

4.2.3. X-Turns Algorithm. In the Go-Neighbor-Switch algorithm, when all the links connected to neighbor switches are faulty, W cannot send the packet to the destination node via neighbor switches. But there are still other ways to get to the destination node. We call this "X-Turns" algorithm. Generally, for $W = \langle G_W, L_W, W_{n-2}, \ldots, W_1, W_0\rangle$ and destination node $T = \langle \overline{G}_W, T_{n-1}, T_{n-2}, \ldots, T_1, T_0\rangle$. $\langle 1, L_W+1, L_{n-2}, \ldots, L_1, L_0, P\rangle$ are faulty links. If $G_W = 1$, $(L_{n-2}, \ldots, L_1, L_0 = W_{n-2}, \ldots, W_1, *)$, P will equal T_{n-2}. Otherwise P will equal W_{n-2}. And $(L_{n-2}, \ldots, L_{L_W}, \ldots L_1, L_0 = W_{n-2}, \ldots, *, \ldots, T_1, T_0)$. The current switch can send a packet to $\langle \overline{G}_W, L_W^+, * - T_{n-2}, \ldots, W_1, W_0\rangle$, then to $\langle \overline{G}_W, L_W^+, * - T_{n-2}, \ldots, W_1, * - W_0\rangle$, next to $\langle G_W, L_W^+, * - W_{n-2}, \ldots, W_1, * - W_0\rangle$, and final to the destination node via switch $\langle \overline{G}_W, L_W^+, T_{n-2}, \ldots, W_1, * - W_0\rangle$ where $L_W^+ = n-1$.

Algorithm 2 Go-Neighbor-Switch_and_X-Turns

```
W = ⟨G_W, L_W, W_{n-2}, ..., W_1, W_0⟩;          /* my switch ID */
T = ⟨G_T, T_{n-1}, T_{n-2}, ..., T_1, T_0⟩;      /* destination node ID */
L = ⟨G_L, L_L, L_{n-2}, ..., L_1, L_0, P⟩;       /* link ID */
if (G_W ≠ G_T)                                   /* W, T: different groups */
    send packet to T^+_{L_W} via link L;         /* increasing level */
    if (L is faulty link)
        send packet to W_N via link L;           /* Go-Neighbor-Switch */
        if (L is faulty link)
            send packet to N(T^+_{L_W-(n-2)}) via link L;/* X-Turns */
            if (L is faulty link)
                routing fails;
            endif
        else
            if (current switch in N(T^+_{L_W-(n-2)})
                send packet to T̄^+_{L_W} via link L;   /* Back to */
                if (L is faulty link)
                    routing fails;
                endif
            else
                if (current switch in T̄^+_{L_W})
                    send packet to T^+_{L_W} via link L;
                    if (L is faulty link)
                        routing fails;
                    endif
                endif
            endif
        endif                                    /* X-Turns end */
    else
        if (current switch in W_N)
            send packet to T^+_{L_W} via link L;/* Go to T^+_{L_W} via W_N */
            if (L is faulty link)
                routing fails;
            endif
        endif
    endif
endif
```

X-Turns algorithm must be used together with Go-Neighbor-Switch algorithm, see **Algorithm 2**, where W_N means the neighbor switch of W, $N(T^+_{LW-(n-2)})$ means W has the same group ID as T and in level $L_W-(n-2) = 0$.

X-Turns algorithm is quite complex and may cause deadlock. In order to avoid the deadlock, we prepare two parameters to save the values of W_{n-2} and W_0 for each packet, and avoid selecting those paths that were visited before when the packet goes back to the switch.

4.2.4. Three-Turn Algorithm. We find another method to send the packet to the different group of current switch when the Go-Neighbor Switch algorithm cannot send the packet to the destination node. We call it the "Three-Turn" algorithm. Generally, for $W = \langle G_W, L_W, W_{n-2}, \ldots, W_1, W_0\rangle$ and destination node $T = \langle \overline{G}_W, T_{n-1}, T_{n-2}, \ldots, T_1, T_0\rangle$, sup-

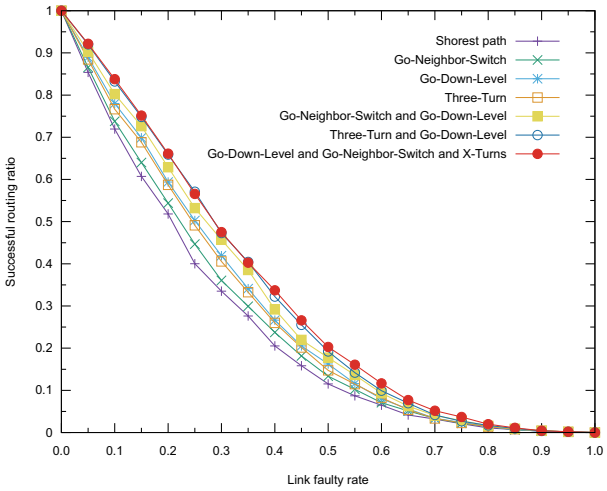

Figure 8. Successful routing ratio on link faulty

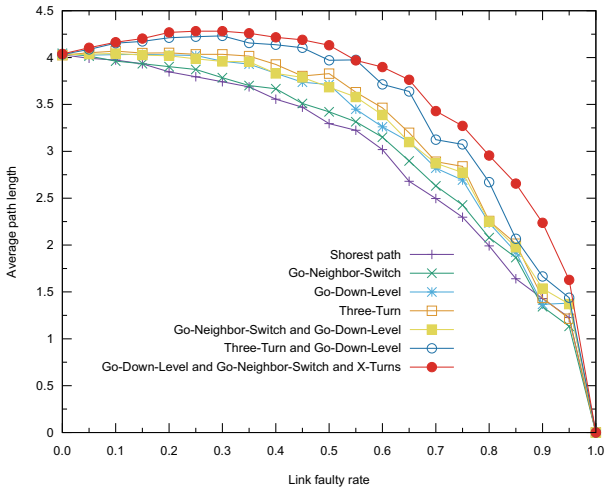

Figure 9. Average path length on link faulty

pose that $\langle 1, L_W + 1, L_{n-2}, \ldots, L_1, L_0, P\rangle$ is faulty. If $G_W = 1$, P equals T_{n-2} and $(L_{n-2}, \ldots, L_1, L_0 = W_{n-2}, \ldots, W_1, W_0)$. Otherwise, P equals W_{n-2} and $(L_{n-2}, \ldots, L_1, L_0 = T_{n-2}, \ldots, T_1, W_0)$. W will send a packet to $\langle \overline{G}_W, L_W, * - T_{n-2}, \ldots, W_1, W_0\rangle$, then to $\langle G_W, L_W, * - W_{n-2}, \ldots, W_1, W_0\rangle$, and next to $\langle \overline{G}_W, L_W, T_{n-2}, \ldots, W_1, W_0\rangle$.

In order to avoid the deadlock, we need a parameter to save the value of the W_{n-2} before W sends a packet to $\langle \overline{G}_W, L_W, * - T_{n-2}, \ldots, W_1, W_0\rangle$, and avoid selecting switch $\langle G_W, L_W, W_{n-2}, \ldots, W_1, W_0\rangle$ when the packet goes back from $\langle \overline{G}_W, L_W, * - T_{n-2}, \ldots, W_1, W_0\rangle$.

5. Evaluation of Algorithms

We have evaluated the performance of the shortest path, Go-Neighbor-Switch, Go-Down-Level, X-Turns, and Three-Turn algorithms and their combinations through simulations. We simulate the network MiKANT$(3, 3)$ with 135 links. MiKANT$(3, 3)$ is the smallest scale in which all the algorithms can be used and all the algorithms can be effective in a larger scale network of MiKANT(k, n).

We developed our own simulator. For a given number of faulty links, we simulate each algorithm 100,000 times. To configure the network for each time, we randomly assign the source and destination nodes. The faulty links are also assigned randomly. One configuration is used for the simulations of all the algorithms.

Fig. 8 shows the successful routing ratios of the algorithms with the link faulty rate ranging from 0% to 100%. The shortest path algorithm has the lowest successful routing ratio. The Go-Neighbor-Switch algorithm, Go-Down-Level algorithm, Three-Turn and their combination have better performance than the shortest path algorithm. As shown in **Algorithm 2**, the X-Turns algorithm must be used with the Go-Neighbor-Switch algorithm. We also applied the Go-Down-Level algorithm to the X-Turns and Three-Turn simulation,respectively. From the figure, we can know the impacts of the algorithms on the successful routing ratios.

Fig. 9 shows the average path length when the routings are successful with the link faulty rate ranging from 0% to 100%. Go-Down-Level algorithm, Three-Turn algorithm, and X-Turns algorithm increase path lengths obviously. Generally, the algorithm achieves a higher successful routing ratio and has a longer routing path. We can find that the path length increases and then decreases as the link faulty rate increases. This is because, when the link faulty rate is high, the routing can be successful only when the source node and destination node have a shorter distance between each other.

6. Conclusions

In this research, we proposed two new interconnection networks, CAT and MiCAT and evaluated their cost ratio, performance, path diversity and average packet latency, the results show that both CAT and MiCAT have lower hardware costs and higher path diversity compared to the fat-trees. Then, we gave four algorithms and their combinations which can find a path from the source node to the destination node at high probability in the MiKANT with faulty links. The future research may develop new algorithms to reduce the packet latencies of CAT and MiCAT.

References

[1] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, 1985.

[2] Y. Li and W. Chu. Mikant: A mirrored k-ary n-tree for reducing hardware cost and packet latency of fat-tree and clos networks. In *The 18th IEEE International Conference on Scalable Computing and Communications*, pages 1643–1650, Oct. 2018.

[3] F. Petrini and M. Vanneschi. k-ary n-trees: high performance networks for massively parallel architectures. In *Proceedings 11th International Parallel Processing Symposium*, pages 87–93, Apr. 1997.

[4] Y. Saad and M.H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37(7):867–872, 1988.

[5] Y. Wang and Y. Li. Hybrid interconnection networks for reducing hardware cost and improving path diversity based on fat-trees and hypercubes. In *2021 Ninth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 254–260, 2021.

[6] Y. Wang and Y. Li. Link fault tolerant routing algorithms in mirrored k-ary n-tree interconnection networks. *International Journal of Networking and Computing*, 11(2):140–153, 2021.