Adaptive and Deadlock-Free Routing for Irregular Faulty Patterns in Mesh Multicomputers

Ming-Jer Tsai and Sheng-De Wang, *Member, IEEE Computer Society*

Abstract—Message routing achieves the internode communication in parallel computers. A reliable routing is supposed to be deadlock-free and fault-tolerant. While many routing algorithms are able to tolerate a large number of faults enclosed by rectangular faulty blocks, there is no existing algorithm that is capable of handling irregular faulty patterns for wormhole networks. In this paper, a two-staged adaptive and deadlock-free routing algorithm called "Routing for Irregular Faulty Patterns" (RIFP) is proposed. It can tolerate irregular faulty patterns by transmitting messages from sources or to destinations within faulty blocks via multiple "intermediate nodes." A method employed by RIFP is first introduced to generate intermediate nodes using the local failure information. By its aid, two communicating nodes can always exchange their data or intermediate results if there is at least one path between them. RIFP needs two virtual channels per physical link in meshes.

Index Terms—Adaptive and deadlock-free routing, fault tolerance, wormhole switching, irregular faulty pattern, virtual channel, mesh multicomputer.

1 Introduction

In parallel computing, processors must communicate with each other to exchange data or intermediate results. This can be achieved by message routing via an interconnection network in a multicomputer. In the first generation multicomputers, the packet switching technique is used for controlling the flow of messages through the network [1], [16], [20], [22], [26], [29]. It regards each packet as an entity that is passed from node to node as it moves through the network. Thus, the network latency is proportional to the distance between two communicating nodes.

In concurrent multicomputers, cut-through switching techniques are used for reducing the network latency, among which wormhole switching [27], virtual cut-through switching [19] and pipelined circuit-switching [12] are among the most popular ones. In the cut-through switching model, each packet consists of a sequence of elementary flow control units called flits spreading over several successive nodes and links. As the header flit advances along a specified route, the subsequent flits follow up in a pipelined fashion. The network latency is thereby insensitive to the length of the message path. In wormhole and virtual cut-through techniques, data flits immediately follow the routing header. Whenever the header flit is blocked at an intermediate node, the remaining flits stop advancing and block all channels they occupied in wormhole switching; in contrast, they are buffered in the intermediate node and removed from the network in virtual cut-through switching. In pipelined circuit-switching, data flits do not immediately follow the routing header. Whenever the header flit is blocked at an intermediate node, it may backtrack and release previously reserved channels on the path and attempt an alternative path. After the header flit reaches the destination, an acknowledge flit returns to the source node. Then, data flits are routed to the destination via the previously reserved path. It is noted that:

- wormhole switching is susceptible to a deadlock [8], [9], [10],
- virtual cut-through switching, like packet switching, needs more buffers to prevent a deadlock [4], [13], [15], [17], [21], [24], [30], and
- pipelined circuit-switching needs larger path setup time.

In wormhole networks, Glass and Ni [14] have proposed a partially-adaptive routing algorithm to tolerate $n-1$ faults in an n-dimensional mesh based on the turn model. In an $N \times N$ 2D mesh, the routing algorithm, presented by Cunningham and Avresky in [6], can improve the performance and provide fault tolerance for up to $N-1$ faults. Besides, Hadas and Brandt [18] proposed an original-based routing algorithm to tolerate square faulty blocks. When a fault arises, their method has a hot-spot effect and will nonminimally route a message even if the message does not encounter any square faulty block.

• *The authors are with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, Republic of China. E-mail: sdwang@hpc.ee.ntu.edu.tw.*

Manuscript received 11 Mar. 1998; revised 24 Nov. 1998; accepted 6 July 1999.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 106494.

1045-9219/00/$10.00 © 2000 IEEE

Based on the concept of virtual interconnection networks, Linder and Harden [23] have proposed a fully-adaptive and fault-tolerant routing algorithm. However, it requires an exponential number of virtual channels per physical link and tolerates a small number of faults. By comparison, Dally and Aoki [7] have proposed a dynamic algorithm to remove cycles from a packet wait-for graph instead of a channel dependency graph. Thus, the virtual channel utilization can be considerably improved. In their algorithm, the number of faults tolerated and virtual channels used depends on the location of faults.

By enclosing faults with rectangular faulty blocks, many adaptive routing algorithms are able to tolerate a large number of faults using a certain number of virtual channels per physical link. Chien and Kim [5] have proposed a planar-adaptive routing algorithm, which requires three virtual channels per physical link to tolerate faulty blocks with a distance of two in at least one dimension. Using extra four virtual channels per physical link, Boppana and Chalasani [2] can enhance any fully-adaptive algorithm to tolerate faulty blocks with a distance of two in at least one dimension. Besides, using three virtual channels per physical link, the algorithm presented by Boura and Das [3] provides full-adaptivity and fault-tolerance. It can tolerate faulty blocks with a distance of three. In addition, Su and Shin [28] have proposed an adaptive routing algorithm to tolerate faulty blocks with a distance of three in at least one dimension using only two virtual channels per physical link. However, all these methods introduce rectangular faulty blocks within which all nodes are regarded as faulty ones. Thus, good nodes within faulty blocks are prohibited from exchanging messages with the other good nodes, resulting in considerably degraded node utilization.

This paper proposes a new routing method RIFP that is able to tolerate irregular faulty patterns. We first enclose faulty nodes with rectangular blocks, as do the above algorithms [2], [3], [5], [28]. Then, we develop a routing method capable of routing messages whose sources and/or destinations are within faulty blocks. Since messages need to be routed around faulty nodes, routing within faulty blocks is prone to a deadlock. Thus, the difficulty of routing messages within faulty blocks is how to prevent a deadlock. We introduce the usage of intermediate nodes. By its aid, messages are instructed to reach their destinations without coming to a deadlock.

The rest of this paper is organized as follows: In the next section, necessary notations and definitions are introduced. The node fault is assumed to be the basic fault element through this paper. In Section 3, a method employed by RIFP is developed to generate intermediate nodes. The time complexity of this method is also analyzed. In Section 4, the routing algorithm RIFP with two illustrative examples is given. Finally, we conclude this paper in Section 5.

2 Notations and Definitions

For ease of reference, some notations are summarized in Table 1. They are illustrated in the following example. After that, definitions used in this paper are clearly defined.

Example. See Fig. 1. $S(B)$ contains node (i,j) with $2 \le i \le 10, 2 \le j \le 10$. $SB(B)$ contains nodes $(1,1)$, $(2,1)$, $\cdots$, $(11,1)$, $(11,2)$, $\cdots$, $(11,11)$, $(10,11)$, $\cdots$, $(1,11)$, $(1,10)$, $\cdots$, $(1,2)$. $SE(B)$ contains nodes $(2,2)$, $(3,2)$, $\cdots$, $(10,2)$, $(10,3)$, $\cdots$, $(10,10)$, $(9,10)$, $\cdots$, $(2,10)$, $(2,9)$, $\cdots$, $(2,3)$. And, $NF(M_3) = 4$, $Dim(M_3) = 2$, S_3 contains node (i,j) with $2 \le i \le 4$, $6 \le j \le 10$. Besides, $(M_3.0.lb,\ M_3.0.ub) = (2,4)$, $(M_3.1.lb,\ M_3.1.ub) = (6,10)$, $(B.0.lb, B.0.ub) = (2,10)$, $(B.1.lb, B.1.ub) = (2,10)$.

Definition 1 (Safe/Unsafe Note). *[22] A good node is called an unsafe node if it neighbors at least two faulty/unsafe nodes, and is called a safe node otherwise.*

Definition 2 (Connected Mesh). *An expanded mesh M is connected if there exists at least one path between any two good nodes x and y in M.*

Definition 3 (Neighboring Meshes). *Two expanded meshes M_i and M_j are neighboring if there exists at least one link (x,y), where x and y are good nodes in M_i and M_j, respectively.*

Definition 4 (Node Address Ordering). *Let $Addr(x) = (x_0, x_1, \cdots, x_{n-1})$ and $Addr(y) = (y_0, y_1, \cdots, y_{n-1})$ be addresses of nodes x and y, respectively. Then, $Addr(x) < Addr(y)$ if there exists i such that:*

1. $x_i < y_i$, *and*
2. $\forall j > i, x_j = y_j$.

Example. See Fig. 1 for the detailed description of Definitions 1 through 4. In the relaxation process of node deactivation, nodes $(3,5)$, $(3,6)$, $(4,4)$, $(4,8)$, $(5,3)$, $(5,6)$, $(5,8)$, $(6,4)$, $(6,5)$, $(6,7)$, $(7,6)$, $(7,9)$,$(8,6)$, $(8,8)$, $(8,10)$, $(9,5)$ are first marked as unsafe nodes based on Definition 1. It then goes to mark nodes $(2,6)$, $(3,4)$, $(4,3)$, $(4,9)$, $(5,2)$, $(6,8)$, $(6,9)$, $(7,4)$, $(7,7)$, $(8,7)$, $(9,8)$, $(10,5)$ to be unsafe nodes. The relaxation process continues until the node neighbors are down to at least two unsafe/faulty nodes. After that, a faulty block B is finally formed, within which all nodes are unsafe/faulty nodes. Besides, suppose that faulty block B is divided into six expanded meshes $M_1, M_2, \cdots, M_6$. Then each of M_1, M_2, M_3, and M_5 is connected; however, each of M_4 and M_6 is not connected based on Definition 2. In addition, M_3 neighbors to M_1; however, M_2, M_4, M_5, and M_6 do not neighbor to M_1 based on Definition 3. Nevertheless, we

TABLE 1
Summary of Notation

VIN_i	the virtual interconnection network i ($i = 1, 2$).
$VC_{i,j}$	the virtual channel in dimension i of VIN_j.
$num(VC_{i,j})$	the channel number of virtual channel $VC_{i,j}$.
$S(B)$	a set consisting of all nodes within faulty block B.
$SB(B)$	a set consisting of all nodes located on the border of faulty block B. Specifically, $SB(B)$ contains every node $x \notin S(B)$ such that x is on the corner of block B or neighbors to exactly one node $y \in S(B)$.
$SE(B)$	a set consisting of all nodes located on the edge of block B. Specifically, $SE(B)$ contains every node $x \in S(B)$ such that x neighbors to at least one node $y \in SB(B)$.
$NF(M)$	the number of faulty nodes in expanded mesh M.
$Dim(M)$	the dimension of expanded mesh M.
S_i	a set consisting of all nodes in expanded mesh M_i.
$Addr(x)$	the address of node x.
$Addr(x).k$	the address value of node x in dimension k.
$M.k.lb$ ($M.k.ub$)	the smallest (largest) address of expanded mesh M in dimension k.
$B.k.lb$ ($B.k.ub$)	the smallest (largest) address of faulty block B in dimension k.

have $Addr((0,0)) < Addr((1,0)) < \cdots < Addr((16,0)) < Addr((0,1)) < Addr((1,1)) < \cdots < Addr((16,1)) < \cdots < Addr((0,16)) < Addr((1,16)) < \cdots < Addr((16,16))$ $Addr((16,16))$ based on Definition 4.

3 Establishing Intermediate Nodes

As was described before, the difficulty of routing a message within a faulty block is how to prevent a deadlock. Our basic idea is described as follows: First, we decompose a faulty block into several connected expanded meshes. Next, we find a routing method such that routing in each expanded mesh is deadlock-free and cycles cannot form between these expanded meshes.

We decompose a faulty block into several expanded meshes in each of which the number of faulty nodes is less than its dimension. There are two reasons. The first is that these expanded meshes are provably connected. The second is that Glass and Ni's algorithm [14] is deadlock-free in each of these expanded meshes.

However, how to prevent cycles between these expanded meshes? First, we construct a directed acyclic graph DAG for a faulty block. Each vertex V_i corresponds to an expanded mesh M_i. Each directed edge (V_i, V_j) represents that M_i neighbors M_j. Then, suppose that a message in M_k is destined for a node in M_t. If V_k is an ancestor of V_t in DAG, then VIN_1 is used to route the message. If V_t is an ancestor of V_k, then VIN_2 is used. Otherwise, if V_k and V_t have a common ancestor V_{ca}, then VIN_2 is first used until the message reaches M_{ca}, and VIN_1 is then used until the message reaches the destination. In our method, the message will traverse several expanded meshes in order before it reaches the destination. For example, assume that the path from V_k to V_t is $P : V_k, V_a, V_b, \cdots, V_t$. Then, the message in M_k will first send to M_a, and then to M_b, and finally to the destination in M_t.

However, how does a message traverse expanded meshes in order? Our method introduces the usage of intermediate nodes. First, we find the pair of neighboring nodes in M_i and M_j, say (n_i, n_j), for each directed edge (V_i, V_j) in DAG. Node n_i is a good node in M_i and node n_j is a good node in M_j. Then, n_i is the intermediate node that instructs the message to enter M_j, and n_j is the intermediate node that instructs the message to enter M_i. In the following subsection, the method of establishing intermediate nodes is described more in detail. This method is analyzed in Section 3.2.

3.1 Algorithm Establish_IN

In this section, Algorithm Establish_IN is used for generating the intermediate nodes, in which each good node $x \in (S(B) \cup SB(B))$ for some faulty block B is able to get the failure information of each node $y \in S(B)$ is assumed. (See Appendix A). It consists of three procedures: Divide_FB Construct_DAG, and Discover_IN. Divide_FB is used for dividing a faulty block into several connected expanded

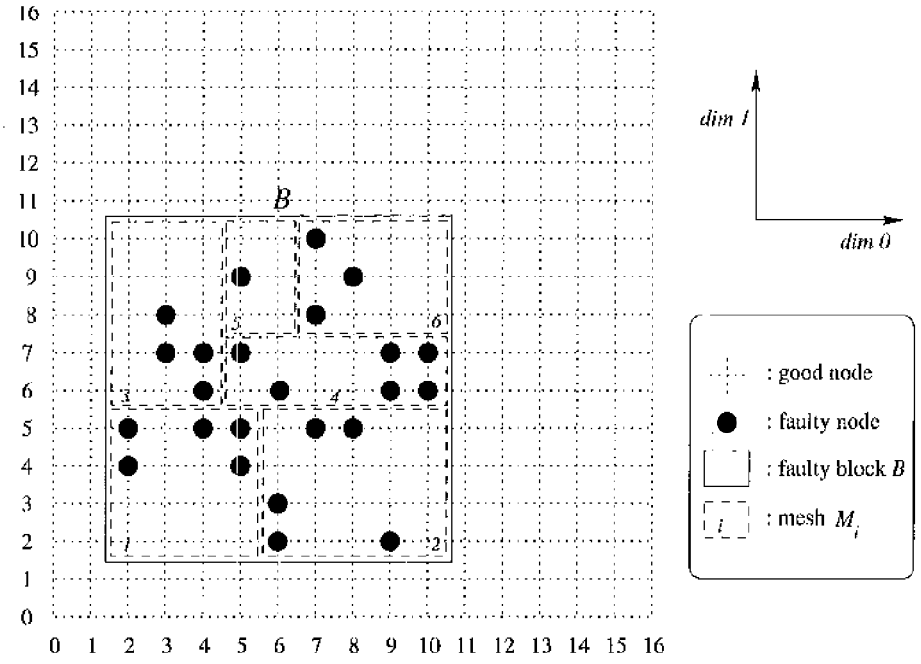


Fig. 1. A 17 × 17 injured mesh M, where one faulty block B is formed and is divided into six expanded meshes $M_1, M_2, \cdots, M_6$.

meshes. Construct_DAG is used for building the directed acyclic graph. Discover_IN is used for generating the intermediate nodes. To simplify our presentation, we will refer to M_0 as the network that interconnects all nodes not within faulty blocks.

Procedure Divide_FB

1. Set j to 1.

2. Select a good node $x \in S(B)$, $x \notin S_1 \cup S_2 \cup \cdots \cup S_{j-1}$, by the following order: /* $S_1 \cup S_2 \cup \cdots \cup S_{j-1} = \emptyset$ if $j < 2$ */
   2.1. $x \in SE(B)$ such that $\forall y \in SE(B), y \neq x$, $y \notin S_1 \cup S_2 \cup \cdots \cup S_{j-1} \Rightarrow Addr(x) < Addr(y)$.
   2.2. x neighbors to $M_{p_1}, M_{p_2}, \cdots, M_{p_a}$ $(1 \leq p_1 < p_2 < \cdots < p_a \leq j-1)$ such that $\forall y \in S(B), y \neq x, y \notin S_1 \cup S_2 \cup \cdots \cup S_{j-1}, y$ neighbors to $M_{q_1}, M_{q_2}, \cdots, M_{q_b}$ $(1 \leq q_1 < q_2 < \cdots < q_b \leq j-1) \Rightarrow (p_1 < q_1)$ or $((p_1 = q_1)$ and $(Addr(x) < Addr(y)))$.

3. For $k = n-1$ downto 0 do
   3.1 Set $M_j.k.lb$ and $M_j.k.ub$ to $Addr(x).k$

4. Decrease $M_j.k.lb$ by 1.

5. If $(M_j.k.lb < B.k.lb)$ or $(S_j \cap (S_1 \cup S_2 \cup \cdots \cup S_{j-1}) \neq \emptyset)$ or $(NF(M_j) \geq Dim(M_j))$, then increase $M_j.k.lb$ by 1. /* M_j fails to expand in dimension $-k$. */

6. Else goto Step 4. /* M_j continues to expand in dimension $-k$. */

7. Increase $M_j.k.ub$ by 1.

8. If $(M_j.k.ub > B.k.ub)$ or $(S_j \cap (S_1 \cup S_2 \cup \cdots \cup S_{j-1}) \neq \emptyset)$ or $(NF(M_j) \geq Dim(M_j))$, then decrease $M_j.k.ub$ by 1. /* M_j fails to expand in dimension $+k$. */

9. Else goto Step 7. /* M_j continues to expand in dimension $+k$. */

10. Increase k by 1.

11. If $k < n$, then goto Step 4. /* M_j expands in higher dimension. */

12. Increase j by 1.

13. If there exists a good node $x \in S(B)$, $x \notin S_1 \cup S_2 \cup \cdots \cup S_{j-1}$, then goto Step 2. /* Divide_FB expands another mesh to enclose unselected good nodes. */

In Step 2, Divide_FB seeds M_j with an unselected good node x within faulty block B. It first selects node x with the smallest address value among all nodes located on the edge of B in Step 2.1. If there is no node located on the edge of this block, then it selects node x with the smallest address value among all nodes neighboring to M_{i_1} in Step 2.2. (Assuming that $M_{i_1}, M_{i_2}, \cdots, M_{i_t}$ $(i_1 < i_2 < \cdots < i_t)$ neighbor to at least one unselected good node within B.) Steps 4 through 11 are used for growing M_j. Divide_FB expands M_j in dimension -0, $+0$, -1, $+1$, $\cdots$, $-(n-1)$, $+(n-1)$ by order. Steps 4 through 6 are used

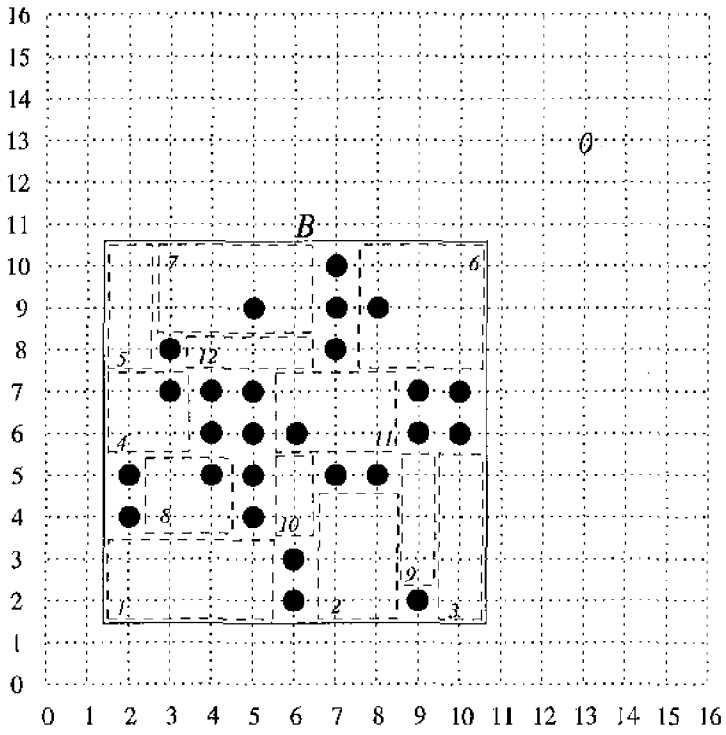


Fig. 2. A 17 × 17 injured mesh, where a faulty block B is divided into 12 expanded meshes $M_1, M_2, \cdots, M_{12}$ after the completion of Divide_FB.

to expand M_j in dimension $-k$. In Step 5, M_j fails to expand in dimension $-k$, if one of the following three conditions holds:

1. M_j exceeds the edge of B in dimension $-k$. That is, $M_j.k.lb < B.k.lb$.
2. M_j overlaps with one of $M_1, M_2, \cdots, M_{j-1}$. That is, $S_j \cap (S_1 \cup S_2 \cup \cdots \cup S_{j-1}) \neq \emptyset$.
3. The number of faulty nodes in M_j is greater than or equal to the dimension of M_j. That is, $NF(M_j) \geq Dim(M_j)$.

If M_j fails to expand in dimension $-k$, then it expands in dimension $+k$ as seen in Steps 7 through 9. In Step 11, M_j continues to expand in higher dimension. Finally, in Step 13, the execution continues to enclose unselected good nodes within B with another expanded mesh. Some features of Divide_FB are uncovered:

1. Each good node within faulty block B is included in exactly one expanded mesh.
2. The number of faults spreading over an expanded mesh M_j is less than the dimension of M_j. Thereby, each one of the expanded meshes is connected.
3. Suppose that there is a path between node n_j and n_0. n_j is a good node in M_j and n_0 is in M_0. Then, there exists

$$M_0 = M_{a_1}, M_{a_2}, \cdots, M_{a_b} = M_j$$

($0 = a_1 < a_2 < \cdots < a_b = j$) such that M_{a_c} neighbors to $M_{a_{c+1}}$ for $c = 1$ to $b - 1$. For example, see Fig. 2. For M_{12}, there exists M_0, M_7, M_{12} such that M_0 neighbors to M_7, and M_7 neighbors to M_{12}.

Example. See Fig. 2 for the description of Divide_FB. In the expansion of M_1, node $(2,2)$ is selected in Step 2, since its address is lowest among all nodes located on the edge of faulty block B. Then M_1 first expands toward dimension -0 during Steps 4 through 6. However, it fails, since M_1 exceeds the edge of B. After that, M_1 expands in dimension +0 during Steps 7 through 9 until it encounters faulty node $(6,2)$. M_1 cannot include node $(6,2)$, otherwise the number of faulty nodes within it is equal to its dimension. (It is noted at this time that $Dim(M_1) = 1$.) Thereafter, M_1 fails to expand in dimension -1, since it exceeds the edge of B. Thus, M_1 continues to expand in dimension +1 until it encounters nodes $(2,4)$, $(3,4)$, $(4,4)$, $(5,4)$. Again, M_1 cannot include these nodes, otherwise the number of faulty nodes within it is equal to its dimension. ($Dim(M_1) = 2$ at this time.) At Step 13, at least one good node is in B which is not included in M_1, thus the execution goes to Step 2. And so on, meshes $M_1, M_2, \cdots, M_{12}$ expand within B.

Procedure Construct_DAG

1. Let DAG be a vertex V_0.
2. For $k = 1$ to q do /* q denotes the number of expanded meshes within B. */

2.1. Add vertex V_k to DAG.
2.2. For each M_t $(0 \le t < k)$ neighboring to M_k do
2.2.1. Add a directed edge (V_t, V_k) to DAG.
2.2.2. Let $G_{t,k}$ and $G_{k,t}$ be empty sets.
2.2.3. For each pair of neighboring nodes $w_t \in M_t$ and $w_k \in M_k$ do /* dim denotes the directed dimension where the link from node w_t to node w_k is located. */
2.2.3.1. $G_{t,k} = G_{t,k} \cup (w_t, dim, 1)$.
2.2.3.2. $G_{k,t} = G_{k,t} \cup (w_k, -dim, 2)$.

Example. See Fig. 3 for the detailed description of Construct_DAG. Initially, DAG contains only one vertex V_0 in Step 1. Then Construct_DAG adds vertex V_1 to DAG in Step 2.1 and adds a directed edge (V_0, V_1) to DAG in Step 2.2.1. In Step 2.2.3, $G_{0,1}$ and $G_{1,0}$ are used for storing the intermediate nodes from M_0 to M_1 and from M_1 to M_0, respectively.

$$G_{0,1} = \{((1,3),+0,1),\ ((1,2),+0,1),\ ((2,1),+1,1),\ ((3,1),+1,1),\ ((4,1),+1,1),\ ((5,1),+1,1)\},$$
$$G_{1,0} = \{((2,3),-0,2),((2,2),-0,2),((2,2),-1,2),((3,2),-1,2),((4,2),-1,2),((5,2),-1,2)\}.$$

$G_{0,1}$ and $G_{1,0}$ indicate that there are six intermediate nodes from M_0 to M_1 and from M_1 to M_0, respectively. Construct_DAG terminates after vertex V_{12} is added to DAG. It is noted that the intermediate nodes between neighboring expanded meshes M_i and M_j can be obtained from $G_{i,j}$.

In the following, Discover_IN is described, which is used to generate the intermediate nodes for the node in M_t.

Procedure Discover_IN

1. For $k = 0$ to q $(k \neq t)$ do
1.1. If vertex V_k is an ancestor of vertex V_t in DAG and $P_{up} : V_k \rightarrow \cdots \rightarrow V_a \rightarrow V_t$ is a directed shortest path from vertex V_k to vertex V_t, then $T_{t,k}$ is set to $G_{t,a}$.
1.2. Else if vertex V_t is an ancestor of vertex V_k in DAG and $P_{down} : V_t \rightarrow V_a \cdots \rightarrow V_k$ is a directed shortest path from vertex V_t to vertex V_k, then $T_{t,k}$ is set to $G_{t,a}$.
1.3. Else if $P_{up} : V_0 \rightarrow \cdots \rightarrow V_a \rightarrow V_t$ is a directed shortest path from vertex V_0 to vertex V_t, then $T_{t,k}$ is set to $G_{t,a}$. □

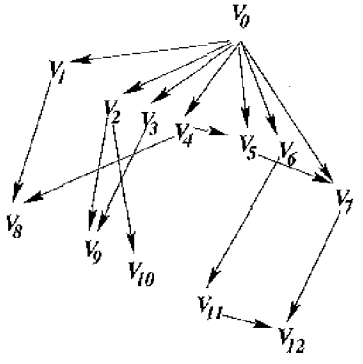


Fig. 3. The directed acyclic graph constructed for faulty block B in Fig. 2 after the completion of Construct_DAG.

Example. See Table 2 for the detailed description of Discover_IN. $T_{1,0}$ is set to $G_{1,0}$ due to Step 1.1. $T_{1,8}$ is set to $G_{1,8}$ due to Step 1.2. And, $T_{1,3}$ is set to $T_{1,0}$ due to Step 1.3. It is noted that Table 2 is not unique. For example, $T_{0,8}$ can be set to $G_{0,1}$ or $G_{0,4}$, since both path $V_0 \rightarrow V_1 \rightarrow V_8$ and path $V_0 \rightarrow V_4 \rightarrow V_8$ are the shortest paths from V_0 to V_8 in DAG. Besides, it is also noted that the intermediate nodes between any two expanded meshes M_i and M_j can be obtained from $T_{i,j}$.

Algorithm Establish_IN

1. For each node $x \in S(B) \cup SB(B)$ do
1.1. Call Divide_FB.
1.2. Call Construct_DAG.
1.3. Call Discover_IN.

3.2 The Time Complexity Analysis

In this section, we will derive the time complexity of Establish_IN. As Establish_IN is composed of three procedures: Divide_FB, Construct_DAG, and Discover_IN, we will first analyze these three procedures, respectively.

In Divide_FB, the data structure Order_Set is first introduced as shown in Fig. 4. In Order_Set, the elements are in order of value. There are four operations find, delete, find_min, int_max for Order_Set. Operation find(S, x) returns "True" if x is an element of set S, and returns "False" otherwise. Operation delete(S, x) removes the element x from set S. Operation find_min(S) returns $\emptyset$ if S is an empty set, and returns and deletes the element x with the smallest value in S otherwise. And, operation int_max(S, x) adds the element x to set S, where the value of x is larger than that of any element in S. It is clear that these four operations can be implemented in $O(1)$ time. In Divide_FB, S_g, S_E, SS_i are sets of Order_Set. S_g contains all unselected good nodes within faulty block B. It is initially constructed in $O(N)$ by repeated int_max operations, where N denotes the number of nodes within B. For example, in Fig. 2, S_g is constructed initially by the operations int_max$(S_g, (2,2))$, int_max$(S_g, (3,2))$, $\cdots$, int_max$(S_g, (10,2))$, int_max$(S_g, (2,3))$, int_max$(S_g, (3,3))$, $\cdots$, int_max$(S_g, (10,3))$, $\cdots$, int_max$(S_g, (2,10))$, int_max$(S_g, (3,10))$, $\cdots$, int_max$(S_g, (10,10))$ in order. S_g excludes all nodes in S_i after M_i is expanded in Divide_FB. S_E contains all unselected good nodes located on the edge of B. Also, S_E is initially constructed in $O(N)$ time. In Fig. 2, S_E is

TABLE 2
$T_{j,k}$ Obtained for Faulty Block B

j,k	0	1	2	3	4	5	6	7	8	9	10	11	12
0	-	1	2	3	4	5	6	7	1	2	2	6	7
1	0	-	0	0	0	0	0	0	8	0	0	0	0
2	0	0	-	0	0	0	0	0	0	9	10	0	0
3	0	0	0	-	0	0	0	0	0	9	0	0	0
4	0	0	0	0	-	5	0	5	8	0	0	0	5
5	0	0	0	0	4	-	0	7	0	0	0	0	7
6	0	0	0	0	0	0	-	0	0	0	0	11	11
7	0	0	0	0	5	5	0	-	0	0	0	0	12
8	1	1	1	1	4	4	4	4	-	1	1	1	4
9	2	2	2	3	3	3	3	2	2	-	3	3	3
10	2	2	2	2	2	2	2	2	2	2	-	2	2
11	6	6	6	6	6	6	6	6	6	6	6	-	12
12	7	7	7	7	7	7	11	7	7	7	7	11	-

This is obtained in Fig. 2 after the completion of Discover_IN, where "$T(j,k) = i$" denotes "$T(j,k) = G_{ji}$."

initially constructed by operations int_max$(S_E,(2,2))$, int_max$(S_E,(3,2))$, $\cdots$, int_max$(S_E,(10,2))$, int_max$(S_E,(2,3))$, int_max$(S_E,(10,3))$, $\cdots$, int_max$(S_E,(2,9))$, int_max$(S_E,(10,9))$, int_max$(S_E,(2,10))$, int_max$(S_E,(3,10))$, $\cdots$, int_max$(S_E,(10,10))$ in order. SS_i contains all unselected good nodes that are within B and neighbor to M_i. Still, it is initially constructed in $O(N)$ time by repeated int_max operations after M_i is expanded in Divide_FB. In Fig. 2, SS_1 is initially constructed by int_max$(SS_1,(3,4))$ and int_max$(SS_1,(4,4))$. It is noted that the construction of $SS_1, SS_2, \cdots, SS_q$ spends a total of $O(nN)$ time, since every node within B is included in at most $2n$ SS_i.

Now, we analyze the time complexity of Divide_FB. It is clear that Step 1 executes one time, Steps 2, 3, 12, 13 execute q times, and Steps 10, 11 execute nq times. It is clear that Step 1 runs in $O(1)$ time, Step 12 runs in $O(q)$ time, and Steps 3, 10, 11 run in $O(nq)$ time. In Step 2, Divide_FB needs a find_min(S_E) operation in Step 2.1, and needs a find_min(SS_{i_1}) operation in Step 2.2. (Assuming that M_{i_1}, M_{i_2}, $\cdots$ M_{i_t} $(i_1 < i_2 < \cdots < i_t)$ neighbor to at least one unselected good node within B.) Thus, Step 2 runs in $O(q)$ time. Similarly, Step 13 needs a find_min(S_g) operation, and thus runs in $O(q)$ time. In Step 5, Divide_FB counts the faulty nodes among the newly-added nodes of M_j, and checks whether all these newly-added nodes are contained in S_g. Because every node within B is checked at most $2n$ times in Step 5, Step 5 runs in $O(nN)$ time. The similar argument can be applied for Step 8. Besides, it is trivial that Steps 4 and 6 need less time than Step 5, and Steps 7 and 9 need less time than Step 8. Thus, the time complexity of Divide_FB is $O(nN)$.

In Construct_DAG, it is clear that Step 1 runs in $O(1)$ time, Step 2.1 runs in $O(q)$ time, and Steps 2.2.1 and 2.2.2 run in $|E|$ time, where $|E|$ denotes the number of edges in DAG. Since every node within faulty block executes Step 2.2.3 at most $2n$ times, Step 2.2.3 runs in $O(nN)$ time. It is trivial that Steps 2.2.1 and 2.2.2 require less time than Step 2.2.3. It implies that $|E| = O(nN)$. Thus, the time complexity of Construct_DAG is $O(nN)$.

In Discover_IN, the data structure Relation_Set is first introduced as shown in Fig. 5. Each element of Relation_Set contains two entries, including label and relation. Entry label stores the label of node z that is an ancestor or a

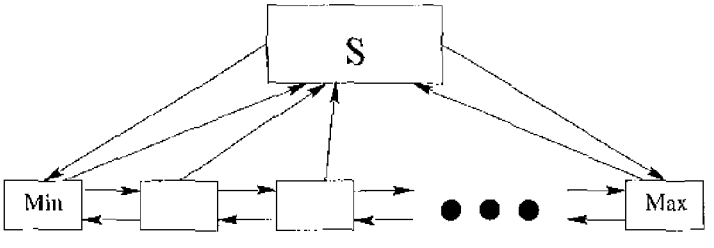


Fig. 4. The data structure Order_Set.

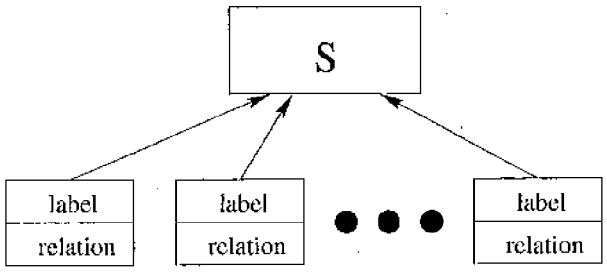


Fig. 5. The data structure Relation_Set.

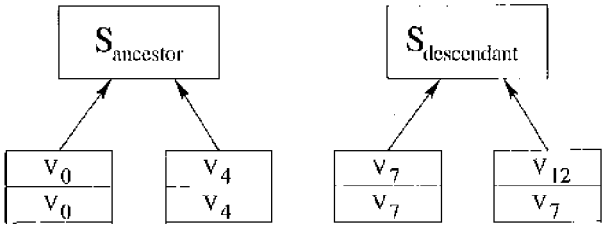


Fig. 6. Examples of $S_{ancestor}$ and $S_{descendant}$ in node V_5.

descendant of the current node x. And, entry relation stores the label of node y that is a parent or a child of node x and is in the shortest path between x and z. In Discover_IN, $S_{ancestor}$ and $S_{descendant}$ are sets of Relation_Set. $S_{ancestor}$ and $S_{descendant}$ consists of all ancestors and descendants of the current node x, respectively. Examples of $S_{ancestor}$ and $S_{descendant}$ constructed in node V_5 are given in Fig. 6. In Discover_IN, $S_{ancestor}$ and $S_{descendant}$ are constructed in advance in $O(nN)$ time (See Appendix B). Then, it needs $O(1)$ time to query whether vertex V_k is an ancestor or a descendant of V_l in Steps 1.1 and 1.2, and needs $O(1)$ time to find V_a in Steps 1.1 through 1.3. It implies each of the steps from Step 1.1 through 1.3, needs $O(q)$ time. Thus, the time complexity of Discover_IN is $O(nN)$.

In sum, the time complexity of Establish_IN is $O(nN)$.

4 The Routing Algorithm

In this section, we develop a two-staged adaptive and deadlock-free routing algorithm RIFP for wormhole networks. Fig. 7 shows the flow chart to route messages using our method. Each node first decides its node status, based on Definition 1. Afterwards, each node $x \in SB(B) \cup S(B)$ gets the failure information of each node $y \in S(B)$ using Algorithm Exchange_Inf (See Appendix A). Thereafter, node x (in M_i) evaluates $T_{i,j}$ for $0 \le j \le q$ (q denotes the number of expanded meshes within faulty block B) using Algorithm Establish_IN. Finally, when node x (in M_i) receives a message destined for a node y (in M_j), it can route the message to the destination y via any intermediate node z in $T_{i,j}$ using algorithm RIFP. For ease of reference, some necessary notations used in RIFP are summarized in Table 3.

4.1 The Routing Algorithm RIFP

In RIFP, the message header format is ($header.N_D$, $header.N_I$, $header.dim$, $header.dir$), where $header.N_D$ stores the address of the destination node N_D, $header.N_I$ stores the address of the intermediate node N_I, $header.dim$ stores the directed dimension along which the message is routed out from N_I, and $header.dir$ stores the virtual interconnection networks used for routing the message from N_C to N_R. RIFP classifies all nodes into three clusters by their locations. In cluster 1, each node x is within faulty block B. That is, $x \in S(B)$. In cluster 2, each node x is located on the border of faulty block B. That is, $x \in SB(B)$. Cluster 3 contains each other node x. That is, $x \notin S(B) \cup SB(B)$.

In RIFP, Su and Shin's method [28] is used to route messages outside faulty blocks. The reason is that their method needs only two virtual channels per physical link. It implies that all nodes in clusters 2 and 3 will route the received message using Su and Shin's method. The exception is that the node x in cluster 2 is sending the message into a faulty block. (See Step 2-3.2.) Besides, Glass and Ni's method [14] is used to route messages within faulty blocks. Thus, all nodes in cluster 1 will route the received message using Glass and Ni's method, excluding the node that is sending the received message into another expanded mesh. (See Step 1-3.)

As was described in Section 3, RIFP employs the pre-established intermediate nodes to instruct the message to reach their destinations. Thus, when a message enters a new expanded mesh or derives from a node within a faulty block, it is supposed to retrieve the information of the intermediate node. (See Step 1-2.) The same action is taken for the node N_F that is first traversed among all nodes in cluster 2 when the received message is going into the faulty block. (See Step 2-3.1.)

Suppose that a message with source node outside faulty blocks is destined for a node within a faulty block B. The message is routed toward its destination by Su and Shin's method until it encounters B. (See Step 3-2.) Since then, the message is routed toward the intermediate node. (See Step 2-3.3.) And thus, the routing direction can possibly be changed. (See Fig. 8.) It implies that messages that are not within faulty blocks can possibly reach a deadlock if they are routed by Su and Shin's method. To break the deadlock, we temporarily buffer the received message in the node N_F, e.g., node B in Fig. 8, that first receives the message among all nodes located on the border of the faulty block into which the message is going. (See Step 2-3.1.4.) Thereby, RIFP is a two-staged routing algorithm.

RIFP for cluster 1: $x \in S(B)$

1-1. If $N_C = N_D$, then exit. /* the message reaches its destination. */

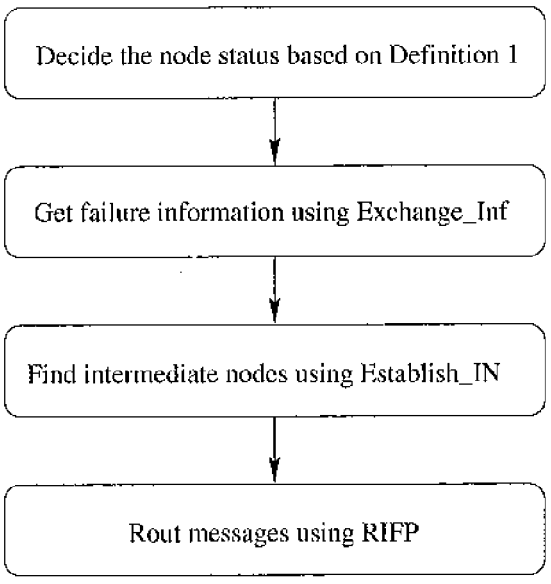


Fig. 7. The flow chart to route messages using RIFP.

1-2. If $N_C = N_S$ or $N_C = N_R$, then
  1-2.1. If N_D is a faulty node, then exit.
  1-2.2. If $N_C = N_S$ and (N_C and N_D are in an expanded mesh), then set $header.dir$ to 1 or 2.
  1-2.3. Else if (N_C and N_D are not in an expanded mesh), then update the message header by
    (1) selecting one element of $T_{q,r}$, say (w_i, dim, dir), assuming that N_C and N_D are good nodes in M_q and M_r, respectively.
    (2) setting $header.N_I$, $header.dim$, and $header.dir$ to w_i, dim, and dir, respectively.
1-3. If $N_C = N_I$, then route the message via $VC_{header.dim,header.dir}$.

1-4. Else if N_C and N_D are in an expanded mesh M_j, then route the message to N_D via $VIN_{header.dir}$ using Glass and Ni's algorithm.

1-5. Else route the message to N_I via $VIN_{header.dir}$ using Glass and Ni's algorithm.

RIFP for cluster 2: $x \in SB(B)$

2-1. If $N_C = N_D$, then exit.

2-2. If $N_D \notin S(B)$, then route the message to N_D via VIN_1 and VIN_2 using Su and Shin's algorithm.

2-3. Else
  2-3.1. If $N_C = N_F$, then
    2-3.1.1. If N_D is a faulty node, then exit.
    2-3.1.2. Select one element of $T_{0,r}$, say (w_i, dim, dir), assuming that N_D is a good node in M_r.

TABLE 3
Summary of Notation

N_S	the source node.
N_D	the destination node.
N_C	the current node.
N_I	the intermediate node.
N_R	the node receives the message from an intermediate node.
N_F	the node $x \in SB(B)$, that the message first traverses when the message is destined for a node within faulty block B.

    2-3.1.3. Set $header.N_I$, $header.dim$, and $header.dir$ to w_i, dim, and dir, respectively.
    2-3.1.4. If $N_C \neq N_R$ and $N_C \neq N_S$, then temporarily store the message in the buffers until the total message is received. /* The second stage begins. */
  2-3.2. If $N_C = N_I$, then route the message via $VC_{header.dim,header.dir}$.
  2-3.3. Else route the message to N_I via VIN_1 and VIN_2 using Su and Shin's algorithm.

RIFP for cluster 3: $x \notin S(B) \cup SB(B)$]

3-1. If $N_C = N_D$, then exit.

3-2. Else route the message to N_D via VIN_1 and VIN_2 using Su and Shin's algorithm.

Example. A message is routed from node $A(10,15)$ to node $G(6,7)$ using RIFP as shown in Fig. 8. At first, the message is routed toward its destination via VIN_1 and VIN_2 using Su and Shin's algorithm due to Step 3-2. Since the destination node G is within a faulty block B, the message must traverse a node in $SB(B)$, say node $B(6,11)$, before it reaches node G. At node B, the message header is updated to $((6,7),(8,11),-1,1)$, and the message is temporarily stored in the buffers until the total message is received due to Step 2-3.1. After that, the message is routed to node $C(8,11)$ via VIN_1 and VIN_2 using Su and Shin's algorithm due to Step 2-3.3. When the message reaches node C, it is routed to node $D(8,10)$ via $VC_{-1,1}$ due to Step 2-3.2. At node D, the header is updated to $((6,7),(8,8),-1,1)$ due to Step 1-2.3. Then it is routed to node $E(8,8)$ via VIN_1 using Glass and Ni's algorithm due to Step 1-5; and thereafter, it is routed to node $F(8,7)$ via $VC_{-1,1}$ due to Step 1-3. Finally, the message is routed to node G via VIN_1 using Glass and Ni's algorithm due to Step 1-4.

Example. A message is routed from node $A(7,6)$ to $J(6,5)$ using RIFP as shown in Fig. 9. At first, the message header is updated to $((6,5),(8,7),+1,2)$ due to Step 1-2.3. Then, the message is routed to node $B(8,7)$ via VIN_2 using Glass and Ni's algorithm due to Step 1-5. Later, it

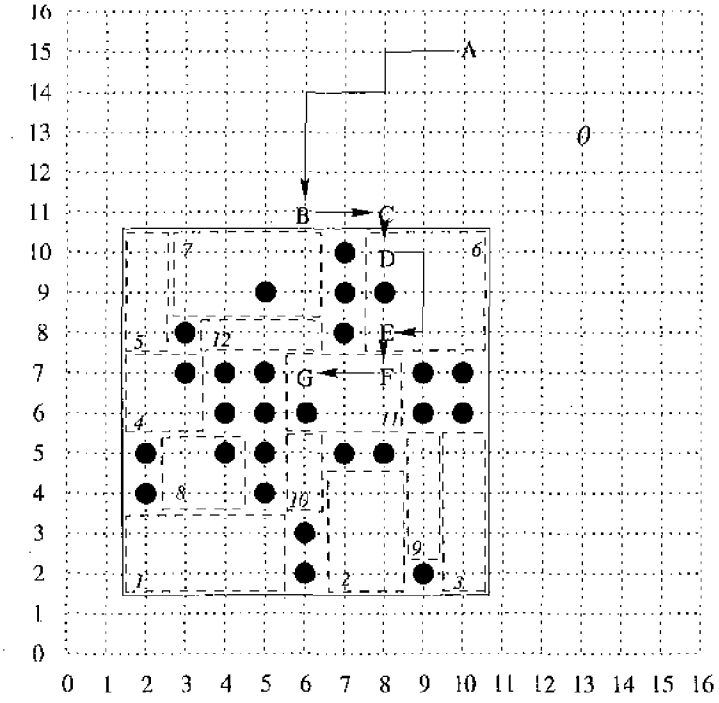


Fig. 8. A 17×17 injured mesh, where a message is routed from node $A(12,15)$ to node $G(6,7)$ by RIFP.

is routed to node C via $VC_{+1,2}$ due to Step 1-3. At node C, the header is updated to $((6,5),(10,8),+0,2)$ due to Step 1-2.3. Then it is routed to node $D(10,8)$ via VIN_2 using Glass and Ni's algorithm due to Step 1-5; and thereafter, it is routed to node $E(11,8)$ via $VC_{+0,2}$ due to Step 1-3. At node E, the header is updated to $((6,5),(8,1),+1,1)$ due to Step 2-3.1. Then it is routed to node $F(8,1)$ via VIN_1 and VIN_2 using Su and Shin's algorithm due to Step 2-3.3; and thereafter, it is routed to node $G(8,2)$ via $VC_{+1,1}$ due to Step 2-3.2. At node G, the header is updated to $((6,5),(7,4),-0,1)$ due to Step 1-2.3. Then it is routed to node $H(7,4)$ via VIN_1 using Glass and Ni's algorithm due to Step 1-5; and thereafter, it is routed to node $I(6,4)$ via $VC_{-0,1}$ due to Step 1-3. Finally, the message is sent to node J via VIN_1 using Glass and Ni's algorithm due to Step 1-4.

Some features of RIFP are uncovered:

1. RIFP requires only two virtual channels per physical link. It employs Su and Shin's algorithm to route messages outside faulty blocks, and employs Glass and Ni's algorithm to route messages within faulty blocks.
2. RIFP is reduced to Su and Shin's algorithm if both the source node and the destination node are not within a faulty block.
3. RIFP is a two-staged routing algorithm. Sometimes, it buffers the received message until the tail flit of this message is reached. We use this method, since it can prevent a deadlock without additional virtual channels. A similar method can be found in [25], [31], [32], where multistaged multicast algorithms for wormhole networks are proposed.

The following theorem shows the correctness of algorithm RIFP.

Theorem 1. *Algorithm RIFP is deadlock-free for wormhole networks using two virtual channels per physical link in meshes.*

Proof. The proof is divided into three parts. In the first part, we assign each virtual channel x one channel number $num(x)$. In the second part, we prove that a message can always find a virtual channel to use for each one hop in a nondecreasing order of channel numbers. In the third part, we claim that a waiting cycle is not a real deadlock.

- **Part 1.** Let virtual channel x be output from a good node in M_i ($i \geq 0$). Then, $num(x)$ is set to i if x is in VIN_1 and is set to $-i$ if x is in VIN_2.
- **Part 2.** Suppose that the current node N_C (a good node in M_q) receives the message from a good node in M_t via virtual channel y and, thereafter, sends out the message toward its destination N_D (a good node in M_r) via virtual channel x. We need to show $num(x) \geq num(y)$. If $N_C \neq N_R$, then $num(x) = num(y)$. Thus, we consider $N_C = N_R$; that is, N_C receives the message from an intermediate node N_I (a good node in M_t ($t \neq q$)). Three cases are discussed.
  - **C2.1.** Vertex V_t is an ancestor of vertex V_r in DAG. Then (V_t, V_q) is a directed edge in the directed shortest path P_{down} from vertex V_t to

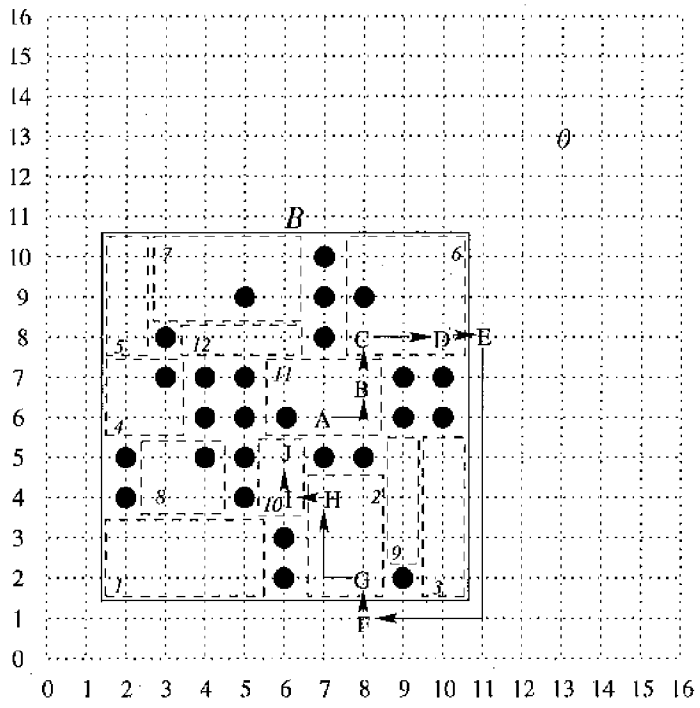


Fig. 9. A 17 × 17 injured mesh, where a message is routed from node $A(7,6)$ to node $J(6,5)$ by RIFP.

vertex V_r due to Step 1.2 of Discover_IN. It implies $t < q \leq r$ and virtual channels x and y are all in VIN_1. Thereby,

$$num(y) = t < q = num(x).$$

- **C2.2.** Vertex V_r is an ancestor of vertex V_t in DAG. Then (V_q, V_t) is a directed edge in the directed shortest path P_{up} from vertex V_r to vertex V_t due to Step 1.1 of Discover_IN. It implies $r \leq q < t$ and virtual channels x and y are all in VIN_2. Thereby,

$$num(y) = -t < -q = num(x).$$

- **C2.3.** Vertices V_r and V_t has a common ancestor V_0. Then (V_q, V_t) is a directed edge in the directed shortest path P_{up} from vertex V_0 to vertex V_t due to Step 1.3 of Discover_IN. If $V_q = V_0$, then virtual channel y is in VIN_2 and virtual channel x is in VIN_1. Thereby, $num(y) = -t < 0 = q = num(x)$. If $V_q \neq V_0$, then virtual channels x and y are all in VIN_2. Thereby,

$$num(y) = -t < -q = num(x).$$

- **Part 3.** Consider a waiting cycle as shown in Fig. 10, where message A (resp. B, C, D) holds virtual channel d (resp. a, b, c) and requests virtual channel a (resp. b, c, d). Owing to P2, we have

$$num(a) \geq num(d) \geq num(c) \geq num(b) \geq num(a).$$

It implies that

$$num(a) = num(d) = num(c) = num(b).$$

Thereby, virtual channels a, b, c, and d are in the same expanded mesh, M_i. If $i = 0$, then messages are routed using Su and Shin's algorithm. If $i \neq 0$, messages are routed using Glass and Ni's algorithm. Thereby, the waiting cycle is not a real deadlock. □

4.2 The Complexity Analysis

Let $O(S)$ be the time complexity of Su and Shin's algorithm, and $O(G)$ be the time complexity of Glass and Ni's algorithm. It is trivial that RIFP needs $O(S)$ time to make routing decision for each node in cluster 3. For the nodes in cluster 1, RIFP needs $O(1)$ time in Steps 1-1 and 1-3. In Step 1-2, RIFP needs to query whether N_D is a faulty node in Step 1-2.1 and whether N_C and N_D are in an expanded mesh in Steps 1-2.2 and 1-2.3. Step 1-2.1 needs $O(1)$ time, since every node has got the failure information of all the

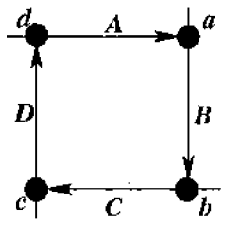


Fig. 10. A waiting cycle of four messages A, B, C, and D.

other nodes within faulty block after Exchange_Inf. Steps 1-2.2 and 1-2.3 need $O(1)$ time to check whether N_C and N_D are contained in the same S_i. In Steps 1-4 and 1-5, RIFP needs $O(1)$ time to query whether N_C and N_D are in an expanded mesh, and needs $O(G)$ time to decide routing direction. Thus, the time complexity of RIFP in each node in cluster 1 is $O(G)$.

For the nodes in cluster 2, RIFP needs $O(1)$ time in Step 2-1. In Step 2-2, RIFP needs $O(1)$ time to query whether N_D is within faulty block B, and needs $O(S)$ time to decide routing direction. In Step 2-3, RIFP needs $O(1)$, $O(1)$, and $O(S)$ time to make a routing decision in Steps 2-3.1, 2-3.2, and 2-3.3, respectively. Thus, the time complexity of RIFP in each node in cluster 2 is $O(S)$.

In sum, in each node, RIFP requires $O(S+G)$ to make a routing decision.

5 Conclusion

Many adaptive wormhole routing algorithms [2], [3], [5], [28] have been proposed to tolerate a large number of faults using a certain number of virtual channels. However, these methods introduce rectangular faulty blocks within which all faulty nodes are regarded as faulty ones. Hence, good nodes within a faulty block are prohibited from communicating with the other good nodes, resulting in considerably degraded node utilization.

For the purpose of exempting node utilization from degradation, we have proposed an adaptive and deadlock-free routing algorithm RIFP for wormhole networks. It is able to transmit messages from or into faulty blocks; and thus, it is able to tolerate irregular faulty patterns. The difficulty of routing messages within faulty blocks is how to prevent a deadlock. In our method, we first decompose a faulty block into several expanded meshes. After that, we design a routing method RIFP such that routing in each expanded mesh is deadlock-free, and cycles cannot form between expanded meshes by way of the usage of intermediate nodes.

In RIFP, the physical network is divided into two virtual networks: VIN_1 and VIN_2. When a message is not within a faulty block, it is routed via VIN_1 and VIN_2 using Su and Shin's algorithm. When a message is within a faulty block, it is routed using Glass and Ni's algorithm. The message within a faulty block is routed via VIN_1 to go out of the faulty block, and is routed via VIN_2 to go into the faulty block. In addition, RIFP is a two-staged routing algorithm. It buffers the received message until all flits are reached at few nodes. Furthermore, RIFP is reduced to Su and Shin's algorithm if both of the source node and the destination node of a message are not within faulty blocks.

Appendix A

Exchange of Failure Information

The following algorithm describes how each good node $x \in (S(B) \cup SB(B))$ gets the failure information of each node $y \in S(B)$. In Step 3, with respect to node x, node y is regarded as a "faulty node" if there is no path between nodes x and y.

Algorithm Exchange_Inf

1. Node x sends the failure information of itself and each neighbor $y \in S(B)$ to each neighbor node $z \in S(B) \cup SB(B)$.

2. If node $x \in S(B) \cup SB(B)$ receives the failure information of node $y \in S(B)$ at the first time, then it sends the information to each neighbor node $z \in S(B) \cup SB(B)$.

3. If node x has got the failure information of each node $y \in S(B)$, then exit.

4. Else goto Step 2.

The time complexity of Exchange_Inf is $O(D)$, where D is the length of the longest path between any two nodes in $S(B) \cup SB(B)$. It is trivial that $D = O(nN)$. It implies that Exchange_Inf runs in $O(nN)$ time.

Appendix B

The Construction of Sets $S_{ancestor}$ and $S_{descendant}$

We first describes how to construct sets $S_{descendant}$ and $S_{ancestor}$ in the following algorithms for the node in M_t. It then goes to analyze the time complexities of these algorithms:

Procedure Construct_Set

1. Let S be an empty set of Relation_Set.

2. Set x to V_t.

3. For each child c of x in DAG do
   - 3.1. If c is in S, then goto Step 3 and continues to scan the next child.
   - 3.2. Insert c into a queue Q.
   - 3.3. Let C be a new element of S, then
     - 3.3.1. $C.label$ is set to c.
     - 3.3.2. If $x = V_t$, then $C.relation$ is set to c.
     - 3.3.3. Else $C.relation$ is set to $X.relation$. /* X is an element of S, in which $X.label = x$. */

4. If Q is not empty, then let x be the element removed from Q, and goto Step 3.

Algorithm Construct_$S_{descendant}$

1. Call Construct_Set.

2. Let $S_{descendant}$ be the set S.

Algorithm Construct_$S_{ancestor}$

1. Change the edge direction for each directed edge in DAG.

2. Call Construct_Set.

3. Let $S_{ancestor}$ be the set S.

In Construct_Set, Steps 1 and 2 need $O(1)$ time. Since each node in DAG is inserted into Q at most once, Steps 3.2, 3.3, and 4 need $O(q)$ time, where q denotes the number of nodes in DAG. In Step 2.1, each directed edge in DAG is scanned at most once, thus it needs $O(|E|)$ time, where $|E|$ denotes the number of edges in DAG. Since $|E| = O(nN)$, as seen in the time complexity analysis of Construct_DAG, Step 2.1 runs in $O(nN)$ time. Summing the time for the different parts of Construct_Set gives a total of $O(nN)$. Besides, it is trivial that Construct_$S_{descendant}$ needs $O(nN)$ time. In Construct_$S_{ancestor}$, Step 1 needs $|E|$ time. Thus, the time complexity of Construct_$S_{ancestor}$ is $O(nN)$.

REFERENCES

[1] W.C. Athas and C.L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *Computer*, vol. 21, no. 8, pp. 9-24, 1988.

[2] R.V. Boppana and S. Chalasani, "Fault-Tolerant Wormhole Routing Algorithms for Mesh Networks," *IEEE Trans. Computers*, vol. 44, pp. 848-864, 1995.

[3] Y.M. Boura and C.R. Das, "Fault-Tolerant Routing in Mesh Networks," *Int'l Conf. Parallel Processing*, pp. 106-109, 1995.

[4] M.S. Chen and K.G. Shin, "Adaptive Fault-Tolerant Routing in Hypercube Multicomputers," *IEEE Trans. Computers*, vol. 39, pp. 1,406-1,416, 1990.

[5] A.A. Chien and J.H. Kim, "Planar-Adaptive Routing: Low-Cost Adaptive Networks for Multiprocessors," *19th Annual Int'l Symp. Computer Architecture*, pp. 268-277, 1992.

[6] C.M. Cunningham and D.R. Avresky, "Fault-Tolerant Adaptive Routing for Two-Dimensional Meshes," *Proc. First IEEE Symp. High Performance Computer Architecture*, pp. 122-131, 1995.

[7] W.J. Dally and H. Aoki, "Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, pp. 466-475, 1993.

[8] W.J. Dally and C.L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. Computers*, vol. 36, pp. 547-553, 1987.

[9] J. Duato, "A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, pp. 1,320-1,331, 1993.

[10] J. Duato, "A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks," *IEEE Trans. Parallel Distributed Systems*, vol. 6, pp. 1,055-1,067, 1995.

[11] P.T. Gaughan and S. Yalamanchili, "Adaptive Routing Protocols for Hypercube Interconnection Networks," *Computer*, pp. 12-23, May 1993.

[12] P.T. Gaughan and S. Yalamanchili, "A Family of Fault-Tolerant Routing Protocols for Direct Multiprocessor Networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 5, pp. 482-497, 1995.

[13] D. Gelernter, "A DAG-Based Algorithm for Prevention of Store-and-Forward Deadlock in Packet Networks," *IEEE Trans. Computers*, vol. 30, pp. 709-715, 1981.

[14] C.J. Glass and L.M. Ni, "Fault-Tolerant Wormhole Routing in Meshes," *23rd Int'l Symp. Fault-Tolerant Computing*, pp. 240-249, 1993.

[15] I.S. Gopal, "Prevention of Store-and-Forward Deadlock in Computer Networks," *IEEE Trans. Comm.*, vol. 33, pp. 1,258-1,264, 1985.

[16] J.M. Gordon and Q.F. Stout, "Hypercube Message Routing in the Presence of Faults," *Proc. Third Conf. Hypercube Concurrent Computing and Applications*, pp. 318-327, 1988.

[17] K.D. Gunther, "Prevention of Deadlocks in Packet-Switched Data Transport Systems," *IEEE Trans. Comm.*, vol. 29, pp. 512-524, 1981.

[18] R.I. Hadas and E. Brandt, "Original-Based Fault-Tolerant Routing in the Mesh," *Proc. First IEEE Symp. High Performance Computer Architecture*, pp. 102-111, 1995.

[19] P. Kermani and L. Kleinrock, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, vol. 3, pp. 267-286, 1979.

[20] C.K. Kim and D.A. Reed, "Adaptive Packet Routing in a Hypercube," *Proc. Third Conf. Hypercube Concurrent Computing and Applications*, pp. 625-630, 1988.

[21] Y. Lan, "An Adaptive Fault-Tolerant Routing Algorithm for Hypercube Multicomputers," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, pp. 1,147-1,152, 1995.

[22] T.C. Lee and J.P. Hayes, "A Fault-Tolerant Communication Scheme for Hypercube Computers," *IEEE Trans. Computers*, vol. 41, pp. 1,242-1,256, 1992.

[23] D.H. Linder and J.C. Harden, "An Adaptive and Fault-Tolerant Wormhole Routing Strategy for k-ary n-cubes," *IEEE Trans. Computers*, vol. 40, pp. 2-12, 1991.

[24] P.M. Merlin and P.J. Schweitzer, "Deadlock Avoidance in Store-and-Forward Networks—I: Store-and-Forward Deadlock," *IEEE Trans. Comm.*, vol. 28, pp. 345-354, 1980.

[25] P.K. McKinley, H. Xu, A.H. Esfahanian, and L.M. Ni, "Unicast-based Multicast Communication in Wormhole-Routed Networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 12, pp. 1,252-1,264, 1994.

[26] P. Ramanathan, K.G. Shin, "Reliable Broadcast in Hypercube Multicomputers," *IEEE Trans. Computers*, vol. 37, pp. 1,654-1,657, 1988.

[27] C. Seitz et al. *Wormhole Chip Project Report*. Winter, 1985.

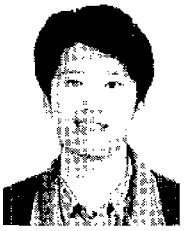
[28] C.C. Su and K.G. Shin, "Adaptive Fault-Tolerant Deadlock-Free Routing in Meshes and Hypercubes," *IEEE Trans. Computers*, vol. 45, pp. 666-683, 1996.

[29] A.S. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[30] M.J. Tsai and S.D. Wang, "A Fully-Adaptive Routing Algorithm for Dynamically-Injured Hypercubes, Meshes, and Tori," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 2, pp. 163-174, 1998.

[31] Y.J. Tsai and P.K. McKinley, "A Broadcast Algorithm for All-Port Wormhole-Routed Torus Networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 8, pp. 876-885, 1996.

[32] Y.C. Tseng, "A Dilated-Diagonal-Based Scheme for Broadcast in a Wormhole-Routed 2D Torus," *IEEE Trans. Computers*, vol. 46, no. 8, pp. 947-952, 1997.



Ming-Jer Tsai received the BS degree in computer and information science from National Chiao Tung University, Taiwan, in 1992, the MS degree in computer science and information engineering from National Chung Cheng University, Taiwan, in 1994, and the PhD degree in electrical engineering from National Taiwan University, Taiwan, in 1997. Since then, he has joined the Institute of Information Science, Academia Sinica, Taiwan, to be a postdoctoral research fellow. Also, in 1998, he joined the Computer and Communications Research Laboratories, Industrial Technology Research Institute, Taiwan. His research interests include neural networks, interconnection networks, and graph algorithms.

Sheng-De Wang received the BS degree from National Tsing Hua University, Hsinchu, Taiwan, in 1980, and the MS and the PhD degrees in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1982 and 1986, respectively. Since 1986, he has been on the faculty of the Department of Electrical Engineering at National Taiwan University, Taipei, Taiwan, where he is currently a professor. Since 1995, he has also served as the director of operating group in the Computer and Information Network Center, National Taiwan University. During the academic year of 1998-1999, he was with the Department of Electrical Engineering, University of Washington, Seattle, as a visiting scholar. His research interests include parallel and distributed systems, real-time operating systems, and intelligent systems. Dr. Wang is a member of the ACM and the IEEE Computer Society. He is also a member of Phi Tau Phi Honor Society.