AN ABSTRACT OF THE THESIS OF

Jai Eun Jang for the degree of Doctor of Philosophy in Computer Science presented on April 4, 1990

Title: Design and Analysis of Robust Algorithms for Fault Tolerant Computing

Redacted for Privacy

Abstract approved: ____________ ______________________________

Bella Bose

We propose a new strategy to recognize the maximum subcube of size k in an n-cube multiprocessor. This strategy will enhance the performance drastically so that our algorithm will outperform the buddy system by a factor ${}_nC_k$, the gray strategy by ${}_nC_k/2$ and Al-Dhelaan strategy by ${}_nC_k/(k(n-k)+1)$ in cube recognition. We present a very efficient processor allocation strategy which makes larger contiguous spaces for the new coming job than the buddy, gray and the Al-Dhelaan strategies do. Furthermore, this new strategy is suitable for static as well as dynamic processors allocation and it results in a less fragmentation and higher fault tolerance. We also describe an efficient procedure for task migration under the new strategy: 1) goal configuration under the new strategy; 2) node-mapping between source and destination nodes; and 3) the shortest deadlock-free routing algorithm.

We describe an optimal fault-tolerant broadcasting algorithm in the hypercube in the presence of n-1 faulty processors. This algorithm takes $\log_2(N) +1$ steps to broadcast the message to all other processors. Our broadcasting algorithm is a procedure by which a

processor can pass a message to all other processsors in the network non-redundantly: This procedure is important for diagnosis of the network, distribution agreement or clock synchronization.

A simple yet efficient algorithm to broadcast in a Cube-Connected Cycles Network containing faulty node/links is proposed. The algorithm is particularly useful in critical real-time systems that can't tolerate the time overhead of identifying the faulty processors on-line. The algorithm delivers multiple copies of the broadcast message through disjoint paths to all the nodes in the system. The salient feature of the proposed algorithm is that the delivery of the multiple copies is transparent to the processes receiving the message and does not require that the precesses know the identity of the faulty processors. The precesses on non-faulty nodes that receive the message identify the original message from the multiple copies using some scheme appropriate for the fault model used.

We describe the definition and theory of adjacent asymmetric error masking codes. When these codes are used for short-circuit faults, they are capable of masking a single adjacent asymmetric error in bus in LSIs. This can be used in minimizing the number of transistors in the decoder of the bus line circuits, i.e., the code have the minimum weight. The bus lines can also be minimized. We systematically derive more codewords than those of previously known codes and present a formula to find the total number of codewords for each weight 2, 3 and 4 in the constant weight codes. When the weight is 2 in constant weight code, we prove that the number of codewords obtained is maximum.

Design and Analysis of Robust Algorithms for Fault Tolerant Computing

by

Jai Eun Jang

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed April 4, 1990
Commencement June 1990

APPROVED:

Redacted for Privacy

Assoc. Professor of Computer Science in charge of major

Redacted for Privacy

Chairman of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented April 4, 1990

Typed by Jai Eun Jang for Jai Eun Jang

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my advisor Professor Bella Bose for the guidance, support, helpful discussion and encouragement throughout this research.

Also, I would like to extend my thanks and respect to other members in my Ph.D committee: Professor Ted Lewis, Professor Toshi Minoura, Professor Pat Lenders and Professor Christopher Mundt for their support, encouragement and unfailing cooperation during my graduate study at OSU and during the development of this research.

I would like to acknowledge Oregon State University for giving me the opportunity to pursue my Ph.D work and Army Headquarters of Republic of Korea for its financial support.

I am very thankful for visiting professor, Won-Kyung Cho, for his helpful comments and encouragement.

I am very thankful to Huan Keh, Seung-Jin Park, Sungwoon Choi, Hae-Sung Kim, Inkyu Kim and Major Kwon for having good discussions and fun here.

Also, I am very thankful for my family for putting up with the pressure of my work that comes with such research and for providing the warm and loving environment that is so important for such an endeavor.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Design and Analysis of Robust Algorithms for Fault Tolerant Computing

Chapter 1

Introduction

Fault tolerance is one of the principle mechanisms for achieving high reliability and high availability in digital systems. The field of fault-tolerance ranges from failure mechanisms in integrated circuits to the design of robust software[1,2].

High reliability in computer design was first achieved through so-called fault-avoidance techniques:these involved computer design which used high quality, thoroughly tested components. Sometimes simple redundancy techniques were employed to achieve limited fault-tolerance.

The issue of organization and architecture of computers are key ones to the design of fault-tolerant computers. In recent years the field of computer architecture has been increasingly concerned with multiprocessors and distributed processing. It is expected that the next generation of computers will consist of innovative interconnections of multiple computing elements. Fault-tolerance issues in interconnecting multiple computing elements therefore will inevitably receive increasing attention.

In this thesis we have contributed some results in these areas and in the rest part of the chapter we briefly describe them.

When designing a large multiprocessor, one of the most important factors is the topology of the communication structure among the processors.

One of the most popular topologies is the n-cube multiprocessors[3-13]. The hypercube is a network of loosely coupled processors connected in such a way that two processors, u and w, are linked if and only if the Hamming distance(u,w) = 1. Hypercube multiprocessors have been drawing considerable attention due to their structual regularity for easy construction and high potential for the parallel execution of various algorithms.

A task arriving at a hypercube multiprocessor must be assigned "optimally" to a subcube in the multiprocessor for execution. Upon completion of execution, the subcube used for the task must be released for later use. Efficient allocation and/or deallocation is a key to its performance and utilization. The processor allocation in a hypercube multiprocessor consists of two steps:

1) determination of the size of the incoming task in terms of the number of processors needed in order to accommodate it.

2) recognition and location of subcube of accommodating the incoming task within the hypercube multiprocessor. We want to maximize the utilization of available resources and also minimize the inherent system fragmentation.

Three allocation strategies for n-cube multiprocessor are addressed: the buddy strategy which is based on the buddy system, the GC strategy which uses single or multiple Gray codes and the Al-Dhelaan strategy[6]. The Buddy system strategy is implemented in [8], and the GC strategy is implemented on an NCUBE/six by the University of Michigan Advanced Architecture Lab[4]. Due to special structure of the n-cube multiprocessor, the availability of some subcubes can't be detected by any of the above systems, and processor utilization is thus degraded.

In chapter 2, we propose a new strategy to recognize the maximum subcube in a n-cube multiprocessor. This strategy will enhance the performance drastically so that our algorithm will outperform the buddy system by a factor ${}_nC_k$, the gray strategy by ${}_nC_k/2$ and Al-Dhelaan[6] by ${}_nC_k/(k(n-k)+1)$ in cube recognition.

We present a very efficient processor allocation strategy which makes larger contiguous spaces for the new coming job than buddy, gray strategy and Al-Dhelaan[6] do.

Furthermore, this new strategy is suitable for static as well as dynamic processors allocation and it results in a less fragmentation and higher fault tolerance.

Also we describe an efficient procedure for the *task migration* under this new strategy. 1) goal configuration, 2) node mapping between source and destination node, 3) shortest deadlock-free routing algorithm.

And we describe the *half-task migration* in the presence of faulty processors. The *half-task migration* is defined as follows: When 2^k processors are allocated for the job in an n-cube, we can have *half-task migration* from 2^{k-1} processors to another 2^{k-1} processors in order to continue the job in the case of processors failure. This approach has the following advantages: 1) We don't have to have the extra processors to reconfigure[11]. 2) It is easy and efficient to reconfigure the processors if the alternatives are chosen.

Broadcasting is an important means of communication among processors by which a processor can pass data or control to all other processors in the network. This operation is extremely important for diagnosis of the network, distributed agreement[13] or clock synchronization[14]. Distributed agreement and clock synchronization can be achieved only if there is no faulty node to deliver the message in the system[13-14]. This, however, is not easy to achieve in the presence of faulty node/link because the faulty nodes can either omit, corrupt, reroute, or alter information passing through them.

There are two possible approaches to overcome this problem. In the first approach, each node keeps limited information about the faulty nodes in the system. Fault-tolerant routing/broadcasting is achieved by going around the faulty nodes[7,16,21]. This approach can be used only if it is possible to identify the faulty processors "on-line". Since the overhead of identifying the faulty processors and passing the fault information to the other nodes could be quite severe, this approach is not suitable for many real-time applications. In the second approach, fault tolerance is achieved by sending multiple copies of the message through disjoint paths[13,17,20]. The nodes that receive the message identify the original message from the multiple copies by using some scheme that is appropriate for the fault model, e.g., majority voting. The second approach has the advantage of not having to identify the faulty processors.

Sullivan and Bashkov[12] developed an algorithm for broadcasting in the hypercube. This algorithm was developed on the assumption of having no faulty processors. Al-Dhelaan[7] developed an algorithm for broadcasting in the hypercube in the presence of some faulty processors. The algorithm works if only one child processor is faulty under any node in the broadcasting tree. Their algorithm does not make explicit use of the properties of the hypercube topology. Ramanathan and Shin[13] developed another algorithm for the hypercube in the presence of faults which uses the second approach mentioned above.

In chapter 3, we describe an optimal fault tolerant broadcasting algorithm when n-1 processors are faulty. The proposed algorithm takes $\log_2(N)+1$ steps to broadcast a message from one processor to all other processors. Our broadcasting algorithm is a procedure by which a processor can pass a message to all other processsors in the network non-redundantly: this message can either be information or control.

Cube-connected-cycles is a parallel network architecture proposed by Preparata and Vuillemin[18]. The CCC can efficiently solve a large class of problems that include Fourier transform, sorting, permutations, etc,. The operation of the cube-connected-cycles network is based on the combination of piplining and parallelism, which leads to the following results[19]:

1. The number of connections per processor is reduced to three.

2. Processing time is not significantly increased with respect to that achievable on the cube-connected network.

3. The overall structure complies with the basic requirements of the VLSI technology: modularity, ease of layout, simplicity of communication among processors, simplicity in timing and control of the entire system.

In chapter 4, we present two approaches mentioned earlier in the presence of faulty node/link in the CCC. The first broadcasting algorithm[17] delivers multiple copies of the message to all nodes in the CCC through disjoint paths. The basic idea of our algorithm is as follows. The node that wants to broadcast a message sends the message to all its neighbors in the same ring. The neighbors in the same ring and the node initiating the message in turn broadcast the message using a simple yet efficient algorithm. The algorithm executed by the neighbors is coordinated such that the copies of the message received by a node have traveled through disjoint paths. The good feature of the proposed algorithm is that the delivery of the multiple copies is transparent to the processes receiving the message and does not require the processes to know the identity of the faulty processors. Depending on the fault modes used, the algorithm can tolerate either s-1 or $\lfloor s/2 \rfloor$ or $\lfloor s/3 \rfloor$ node/link faults. The algorithm completes in $\lfloor s/2 \rfloor + (2s-1) + \lfloor s/2 \rfloor$ steps and 4s steps if each node can use all and at most one of its outgoing links at a time respectively.

The second broadcasting algorithm[21] delivers a copy of message to all nodes nonredundantly. The basic idea of our algorithm is as follows. The node that wants to broadcast a message checks if its neighbor node is faulty or not. If the neighbor node is faulty, the initiating node gives this information to its non-faulty neighbor node. This non-faulty node broadcasts the message to the nodes to be broadcasted by faulty-node. We prove that this algorithm is optimal. This algorithm tolerates 2 processors if two adjacent nodes are faulty and s-1 rings or s-1 processors faults, otherwise. This optimal fault-tolerant broadcasting algorithm takes (5s+4)/2 steps.

In Chapter 5, we design efficient adjacent asymmetric error masking codes (AAEMC) which are useful for masking adjacent bus lines in ROMs. We systematically derive the number of codewords in AAEMC of constant weight, with weight 2, 3 and 4.

When these codes are used in VLSIs, they are capable of masking a single adjacent asymmetric error in bus lines. Furthermore, using these codes we can minimize the number of transistors in the decoder of the bus line circuits and the number of bus lines. However a separate encoder circuit is needed which is not very complex.

REFERENCES

1. R. Negrini and M. G. Sami, Fault Tolerance Through Reconfiguration in VLSI and WSI Arrays, MIT Press, 1989

2. Fault-Tolerant Computing: Theory and Techniques, Prentice-Hall, 1986

3. M. Chen and K.G. Shin, "Processor Allocation in an N-cube Multiprocessor Using Gray Codes", IEEE Trans. Computer, Dec. 1987 pp. 1396-1407.

4. M. Chen and K.G. Shin, "Task Migration in Hypercube Multiprocessor", Proc. 16th Annual Int'l Symp. on Computer Architecture. Jun 1989, pp. 105-111

5. B. Becker and H.U. Simon, "How robust is the n-cube?", in Proc. 27th Ann. Symp. Foundations of Comp. Sci. Oct. 1986 pp. 283-291.

6. A. Al-Dhelaan and B. Bose, "A New strategy for Processor Allocation in an N-cube Multiprocessor", Phoenix Conference on Computer and Communication, Mar 1989. pp. 114-118.

7. A. Al-Dhelaan and B. Bose, "Efficient Fault Tolerant Broadcasting Algorithm for the Hypercube", Proc. The fourth Conf. on Hypercube Concurrent Comp. and Applications, Monterey, Mar 1989. pp. 123-128.

8. NCUBE Corp, NCUBE/10: An Overview, Beverton, OR, Nov 1985

9. J. E. Jang, S. W. Choi and W. K. Cho, "A New Approach to Processor Allocation and Task Migration in an N-cube Multiprocessor, Proceedings, International Conference on Supercomputing, Nov, 1989. pp. 314-325.

10. J. E. Jang and W. K. Cho, "Maximality of Subcube Recognition and Fault Tolerance in an N-cube Multiprocessor", Proceedings, 4th SIAM conference on Parallel Processing for Scientific Applications, Dec, 1989.

11. M. Sultan and Rami Melhem, "Fault Tolerance and Reliable Routing in Augmented Hyercube Architecture", 8th IEEE Int'l Phoenix Conference on Computer and Communication, Mar, 1989. pp. 19-23.

12. H. Sullivan and T. R. Baskow, "A large scale homogeneous, fully distributed parallel machine,I," Proc. Fourth Symp. Comp. Architecture, Mar. 1977, pp. 105-117.

13. P. Ramanathan and K.G. Shin, "Reliable Broadcasting in Hypercube Multicomputers", IEEE Trans. on Comp. Dec 1988, pp 1654-1657.

14. L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," ACM Trans. Programming language System, pp. 382-401, Jul. 1982.

15. T. K. Srikanth and S. Toueg, "Optimal clock synchronization," J. ACM pp.626-645, Jul.1987.

16. J. E. Jang, "Optimal Fault Tolerant Broadcasting Algorithm for Hypercube Multiprocessor", Proceedings, 1990 ACM Computer Science Conference, Feb, 1990. pp. 96-102.

17. J. E. Jang, "Reliable Broadcasting Algorithm in an Cube-Connected Cycles Network", Proceedings. 9th International Phoenix Conference on Computers and Communications, Mar, 1990

18. F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles, A Versatile Network for Parallel Computation," Communication of ACM, pp. 30-39, May 1981.

19. A. Al-Dhelaan and B. Bose, "Efficient Fault Tolerant Broadcasting Algorithm for the Cube-Connected Cycles Network", Proc. IEEE Pacific Rim Conference, May 1989, pp. 161-164.

20. T. K. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerating algorithms," Tech. Rep. 84-623, Dep. Comp. Cornell Univ., Jul. 1984.

21. J. E. Jang, "Optimal Fault Tolerant Broadcasting Algorithm in an Cube-Connected Cycles Network", Proceedings. in the PARBASE-1990, Mar, 1990. pp. 206-215.

22. Kazumitsu. Matsuzawa and Eiji. Fujiwara, "Masking Asymmetric Line Faults using Semi-distance Codes", 18th FTCS, pp. 354-359

Chapter 2

A New Approach to Processor Allocation and Fault-Tolerance in an N-cube Multiprocessor

2.1 Introduction

Hypercube multiprocessors have been drawing considerable attention due to their structual regularity for easy construction and high potential for the parallel execution of various algorithms. Numerous research efforts related to hypercube architectures, operating systems, etc., have been undertaken[1-15]. The problem of processor recognition, allocation, task migration and fault-tolerance in an n-cube is the subject of this chapter. A task arriving at a hypercube multiprocessor must be assigned "optimally" to a subcube in the multiprocessor for execution. Upon completion of execution, the subcube used for the task must be released for later use. Efficient allocation and/or deallocation is a key to its performance and utilization. The processor allocation in a hypercube multiprocessor consists of two steps: 1) determination of the size of the incoming task in terms of the number of processors needed in order to accommodate it, and 2) recognition and location of subcube of accommodating the incoming task within the hypercube multiprocessor. We want to maximize the utilization of available resources and also minimize the inherent system fragmentation.

Three allocation strategies for n-cube multiprocessor are addressed: the buddy strategy which is based on the buddy system, the GC strategy which uses a single or multiple Gray codes and Al-Dhelaan strategy[8]. Due to special structure of the n-cube multiprocessor, the availability of some subcubes can't be detected by any of the above systems, and processor utilization is thus degraded.

In this chapter we propose a new strategy to recognize the maximum subcube in a n-cube multiprocessor. This subcube recognition algorithm can be done in both serial and parallel and this method is analyzed. This strategy will enhance the performance drastically so that our algorithm will outperform the buddy system by a factor ${}_nC_k$, the gray strategy by ${}_nC_k/2$ and Al-Dhelaan[8] by ${}_nC_k/(k(n-k)+1)$ in cube recognition.

Also we present a very efficient processor allocation strategy which makes larger contiguous spaces for the new coming job than buddy, gray strategy and Al-Dhelaan[8] do. Furthermore, this new strategy is suitable for static as well as dynamic processors allocation and it results in a less fragmentation and higher fault tolerance.

Even though enough number of hypercube nodes are available for the incoming job, allocation and deallocation of subcube usually result in a fragmented hypercube. The fragmentation problem in a hypercube can be solved by task migration, i.e., relocating tasks within the hypercube to remove the fragmentation. We describe a shortest deadlock-free routing algorithm for task migration under the new strategy. 1) goal configuration, 2) node mapping between source and destination nodes, 3) shortest deadlock-free routing algorithm.

We describe the *half-task migration* in the presence of faulty processors. The *half-task migration* is defined as follows: When 2^k processors are allocated for the job in an n-cube, we can have *half-task migration* from 2^{k-1} processors to another 2^{k-1} processors in order to continue the job in the case of processors failure. This approach has the following advantages: 1) We don't have to have the extra processors to reconfigure[15]. 2) It is easy and efficient to reconfigure the processors if the alternatives are chosen.

This chapter is organized as follows. Section 2.2 introduces the necessary notation and background. Section 2.3 will describe the new approach to recognize the maximum subcubes in Q_n. Section 2.4 will explain the new allocation strategy which is suitable for static as well as dynamic processor allocation and results in a less system fragmentation, more subcube recognition and higher fault tolerance. In section 2.5 we will describe the parallel algorithm for processor allocation problem and this algorithm has a time complexity of $O({}_nC_k)$. The time complexity of serial algorithm is $O(2^k {}_nC_k 2^{n-k})$. Section 2.6 will describe an efficient procedure for the task migration under this new strategy. 1) goal configuration, 2) node mapping between source and destination node, 3) shortest deadlock-free routing algorithm. Section 2.7 will describe an efficient procedure for the half-task migration under this new strategy.

2.2 Preliminaries and Background

A n-cube can be defined as follows:

Definition: An n-cube Q_n is defined recursively as

a) Q_0 is a trivial graph with one node, and

b) $Q_n = K_2 * Q_{n-1}$, where K_2 is the complete graph with two nodes. Fig. 2.1 and Fig 2.2 shows a Q_3, and a Q_4, hypercubes respectively.

A coding scheme with n bits is defined as a one-to-one mapping from an integer number between 0 and 2^n-1 to a binary representation with n bits. For example, the three bit binary representation of 5 is $B_3(5) = 101$.

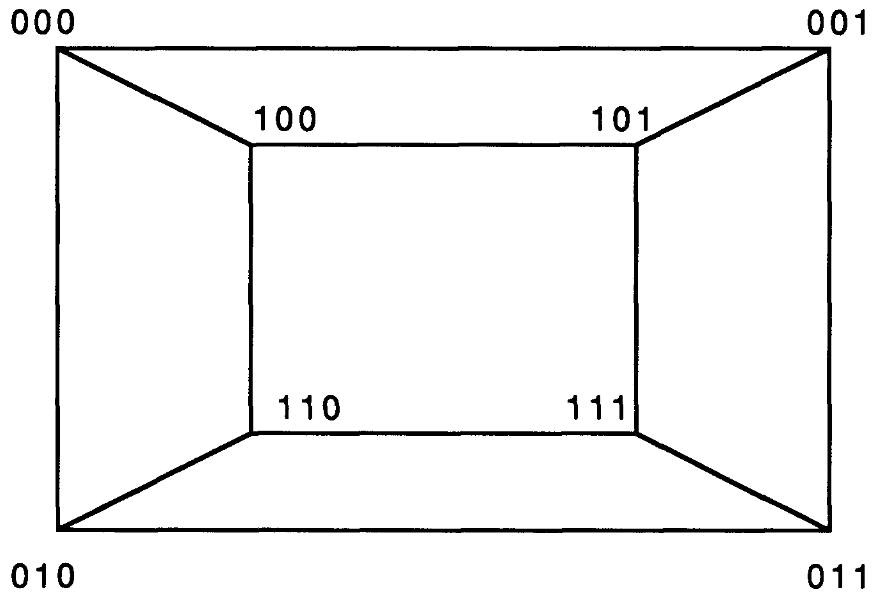


Fig 2.1 A 3-dimensional hypercube, Q_3

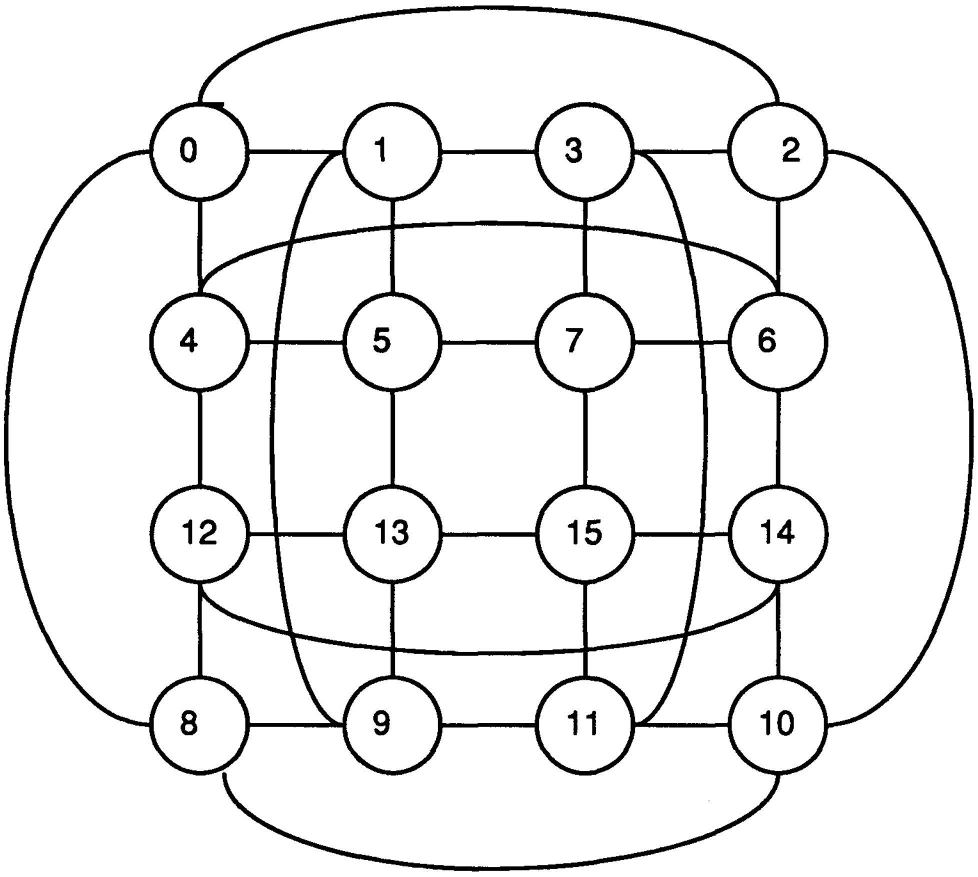


Fig 2.2 A 4-dimensional Hypercube, Q_4

Gray codes, G_n, can be generated as follows. One starts with the sequence of the two 1-bit numbers 0 and 1. This is a 1-bit Gray code. To build a 2-bit Gray code, take the same sequence and insert a zero in front of each number, then take the sequence in reverse order and insert a one in front of each number. In other words, we get the sequence

$G_2 = \{00,01,11,10\}$,

$G_3 = \{000,001,011,010,110,111,101,100\}$

The gray code can be obtained from the corresponding binary number as follows[16].

$g_i = b_i \oplus b_{i+1}$ where $0 \leq i \leq n-1$

$g_n = b_n$

Definition: The Hamming distance between two hypercube nodes with addresses $u = u_n u_{n-1} \ldots u_1$ and $w = w_n w_{n-1} \ldots w_1$ in a Q_n is defined as

$$H(u,w)= \sum_{i=1}^{n} h(u_i,w_i), \text{ where } h(u_i,w_i) = 1, \text{ if } u_i \neq w_i$$
$$= 0, \text{ if } u_i = w_i$$

For example, if u = (10010) and v = (01011), then H(u,w) = 3.

The hypercube is a network of a loosely coupled processors connected in such a way that two processors, u and w, are linked if and only if H(u,w) = 1, i.e., the indices of neighboring processors differ by a power of 2. We can represent an n-cube using link list structure. For example, a processor with address 0000(0) in Fig. 2.2 is linked with 0001(1), 0010(2), 0100(4), 1000(8) in Q_4. In the same way we can represent all the connections of Q_4 in the Table 2.1, where each connection has no duplicated link.

#	address	link(H_1)		#	address	link(H_1)
0	0000	1,2,4,8		8	1000	9,10,12
1	0001	3,5,9		9	1001	11,13
2	0010	3,6,10		10	1010	11,14
3	0011	7,11		11	1011	15
4	0100	5,6,12		12	1100	13,14
5	0101	7,13		13	1101	15
6	0110	7,14		14	1110	15
7	0111	15		15	1111	

Table 2.1 Link connections to each processor without duplication in Q_4

Therefore we can represent all the connection of Q_n with each address using link list structure like Table 2.1. From the Table 2.1 we can find some interesting properties.

First of all, the number of link at each address is the same as the the number of 0's with same address. The value of link in each address is sorted in ascending order and greater than that of each address.

Let Σ be the ternary symbol set {0,1,x}, where x is Don't Care symbol. Then every subcube of an n cube can be uniquely represented by a string of symbols in S. By assigning all combinations of "0" and "1" to x, we can find all the *partners* for the job. For example, when incoming job requires 2^3 processors with node "0", arbitrary 3 link are chosen. If we choose processors {1,2,4} connected to address "0" node, then we have {0000, 0001, 0010, 0100}, that is {0xxx} type. Thus, we can find all the *partners*, {0110,0101,0011,0111}, thus assigning {0000, 0001, 0010, 0100, 0110, 0101, 0011, 0111} for the incoming job to require 2^3 processors.

Let us define some notation to find all the *partners* directly from above table. Let the first column which ranges $0 \sim 2^n-1$ in the link connection table be table_index.
Definition : There are r links per table_index in ascending order. Let p be the position in r. Then partner(table_index, p) will give the corresponding value.
For example, partner(4, 2) = 6.

We have the following Lemmas from the characteristics of the link list table.

Lemma 2.1*:* For every combination pa and pb in the table_index where pb > pa, position(pa) will be the p and pb will be new table_index. Then partner(pb,position(pa)) will be the partner, where position(pa) is the position in r.

proof: All the codewords in the link column are constant weight codes which are Hamming distance 1 from table_index. Therefore, they are Hamming distance 2 from each other and H(table_index,pa) and H(table_index,pb) =1.Thus, H(partner(pb,position(pa)),pb) = 1 And H(partner(pb,position(pa)),table_index) = 2. So, H(partner(pb,position(pa)),pa) = 1. Partner(pb,position(pa)) gives the *partner*.

For example, when incoming job needs Q_2 subcubes in Q_4 with address 0, there are ${}_4C_2$ combinations, that is, (1,2) (1,4) (1,8) (2,4) (2,8) (4,8). Therefore we can find the *partners* 3, 5, 9, 6, 10, 12, respectively.

Lemma 2.2 : While finding the partners, if there are two more links in the new table_index, keep doing Lemma 2.1. Then we will find all the partners.

Proof: If there are one more links in the new table_index, there are one more partners. This means we must find another partner to satisfy Hamming distance 1 for those links. The other part is the same as Lemma 2.1.

For example, when incoming job needs Q_3 subcubes in Q_4 with address 0, there are ${}_4C_3$ combinations, that is, (1,2,4) (1,2,8) (1,4,8) (2,4,8). Let's get the partner when we choose processors (1,2,4). We have the following partner processors: processor 3 for processors (1,2), 5 for (1,4), 6 for (2,4), 7 for (5,6). Thus, we can recognize processors (0,1,2,3,4,5,6,7) for Q_3.

Lemma 2.3: When we have r links at each node with table_index we can recognize ${}_rC_j$ subcubes for the incoming 2^j jobs where $r \geq j$.

proof: Let r and j be the number of link at each table-index and size j for the incoming job 2^j, respectively. When we have j combination in r, we can find the partners using Lemma 2.1. And there are ${}_rC_j$ ways to combine. Therefore we can recognize ${}_rC_j$ subcubes for the incoming 2^j jobs.

For example in Table 2.1, node 0 has 4 links. When incoming job needs 2^2 sub-cubes, there are ${}_4C_2$ ways to recognize the subcubes. We can find the partners for each connection. Therefore we have the following subcubes, which are {(0,1,2,3), (0,1,4,5), (0,1,8,9), (0,2,4,6), (0,2,8,10), (0,4,8,12)}.

This three Lemmas leads to the following important result.

<u>**Theorem 2.1:**</u> We can find all the partners to recognize the incoming subcubes Q_k from the link list table of Q_n.

2.3 Maximality of Subcube Recognition

In this section we propose a new strategy that outperforms the buddy system by a factor of ${}_nC_k$, the gray strategy by a factor of ${}_nC_k/2$ and Al-Dhelaan [8] by ${}_nC_k/(k(n-k)+1)$ in recognizing subcubes of size k in Q_n. Now we are considering how many ways to recognize the subcubes for the incoming jobs.

Lemma 2.4: Total number of subcubes of Q_n to recognize subcubes of size k for incoming 2^k jobs are $\sum_{j=0}^{n-k} {}_nC_{n-j}\ {}_{n-j}C_k$.

proof: According to Lemma 2.2, there are ${}_nC_k$ ways to recognize subcubes of size k. The number of link, r, at each address is the same as the number of 0's in the table index. The number, j, of 0's distribution in Q_n is ${}_nC_j$ where $0 \le j \le n$. In order to recognize the size k, j must be equal or greater than k. So, the total number of subcubes of Q_n to recognize subcubes of size k is $\sum_{j=0}^{n-k} {}_nC_{n-j}\ {}_{n-j}C_k$.

For example, we have the following 0's distribution in Q_4.

$${}_4C_0 = 1 \qquad {}_4C_1 = 4 \qquad {}_4C_2 = 6 \qquad {}_4C_3 = 4 \qquad {}_4C_4 = 1$$

So there are 1 four 0, 4 three 0, 6 two 0, 4 one 0, 1 zero 0. When we examine the Table 2.1, we have 1 *4-links*, 4 *3-links*, 6 *2-links*, 4 *1-link* 1 *0-link*. According to Lemma 2.4, the total number of subcubes to recognize the incoming 2^2 processors in Q_4 is ${}_4C_4\ {}_4C_2$ + ${}_4C_3\ {}_3C_2 + {}_4C_2\ {}_2C_2 = 24$

Lemma 2.5: All the subcubes generated by the new approach are disjoint among themselves.

proof: We choose every combination in each table index and then find the partner according to the combination. partner(table_index,p) will search different value every time, since every combination gives different table_index and p. Thus, all the subcubes are disjoint.

These Lemmas 2.4 and 2.5 lead to the following important result.

Theorem 2.2: Total number of subcubes generated by Lemma 2.4 are maximum.

Proof: The number of distinct subcubes are ${}_nC_k 2^{n-k}$[3].

Therefore we can prove ${}_nC_k 2^{n-k} = \sum_{j=0}^{n-k} {}_nC_{n-j}\,{}_{n-j}C_k$.

$$\sum_{j=1}^{n-k} {}_nC_{n-j}\,{}_{n-j}C_k = \sum_{j=0}^{n-k} \frac{n!}{i!(n-i)!}\,\frac{(n-i)!}{k!(n-i-k)!}$$

$$= \sum_{j=0}^{n-k} \frac{n!}{k!}\,\frac{1}{j!(n-j-k)!}$$

$$= \frac{n!}{k!(n-k)!} \sum_{j=0}^{n-k} \frac{(n-k)!}{j!(n-j-k)!}$$

$$= {}_nC_k \sum_{j=0}^{n-k} \frac{(n-k)!}{j!(n-j-k)!}$$

We have the the binomial theorem as follows.

$$(a+b)^n = \sum_{j=0}^{n} {}_nC_j\, a^{n-j} b^j$$

$$2^{n-k} = \sum_{j=0}^{n-k} {}_{n-k}C_j = \sum_{j=0}^{n-k} \frac{(n-k)!}{j!(n-j-k)!}$$

Therefore, ${}_nC_k 2^{n-k} = \sum_{j=0}^{n-k} {}_nC_{n-j}\,{}_{n-j}C_k$

Here is the example of Q_2 in Q_4, which shows 24 subcubes.

(0,1,2,3) (0,1,4,5,) (0,1,8,9) (0,2,4,6)

(0,2,8,10) (0,4,8,12) (1,3,5,7) (1,3,5,9)

(1,5,9,13) (2,3,6,7) (2,6,10,14) (2,3,10,11)

(3,7,11,15) (4,5,6,7) (4,5,12,13) (4,6,12,14)

(5,7,13,15) (6,7,14,15) (8,9,10,11) (8,9,12,13)

(8,10,12,14) (9,11,13,15) (10,11,14,15) (12,13,14,15)

The number of subcubes recognizable by each of the four strategies is presented in Table 2.2, especially for 2^2 incoming jobs in Q_4.

	Q_0	Q_k	Q_n
Number of distinct subcube	2^n	${}_nC_k 2^{n-k} = 24$	1
The Buddy strategy	2^n	$2^{n-k} = 4$	1
The Gray Code strategy	2^n	$2^{n-k+1} = 8$	1
Ref[8]	2^n	$(k(n-k)+1)2^{n-k} = 20$	1
The New strategy	2^n	${}_nC_k 2^{n-k} = 24$	1

Table 2.2 The number of subcube recognizable by the Buddy, Gray, Al-Dhelaan[8] and New strategy.

2.4. A New Method To Allocate Processors

Node processors in an n-cube multiprocessor must be allocated to incoming tasks in order to maximize processor utilization and minimize system fragmentation. First we will briefly describe the known methods, the buddy strategy and the Gray strategy[3,8] and the one described in ref[8]. Then we can describe the new processor allocation strategy which outperforms the above three strategies. An example of the buddy, Gray, Al-Dhelaan[8] and the new strategies is given in Table 2.3.

A. The Buddy Strategy

Since there are 2^n processor node in a Q_n, 2^n allocation bits are used to keep track of the availability of all the nodes. An allocation bit with value 0 (1) is available (not available). The buddy strategy consists of two parts, processor allocation and processor relinquishment. The algorithm is given below.

Processor allocation :

Step 1 : Set k to the dimension of a subcube required to accommodate the request.

Step 2 : Determine the least integer α, $0 \leq \alpha \leq 2^{n-k+1}-1$ such that all the βth allocation bits are 0's where $\alpha 2^k \leq \beta \leq (\alpha+1)2^k-1$.

Set all these bits to 1's.

Step 3 : Allocate processors with address $B_n(\beta)$ to the request, where $\alpha 2^k \leq \beta \leq (\alpha+1)2^k-1$.

Processor Relinquishment :

Reset every pth allocation bits to 0, where $B_n(p)$ is used in the subcube released.

This strategy can be explained by the completely binary tree. The level where the root node resides is numbered 0, and the nodes in level i are associated with subcubes of dimension n-i. When a Q_k is needed, the buddy strategy searches for a region of allocation bits with 0's whose addresses start with an integral multiple of 2^k.

B. The Gray Strategy

Similar to the buddy strategy, the GC strategy can also be described by the following two parts[3,8].

Processor allocation :

Step 1 : Set k to the dimension of a subcube required to accommodate the request.

Step 2 : Determine the least integer a, $0 \leq a \leq 2^{n-k+1}-1$ such that all the ($b \bmod 2^n$)th allocation bits are 0's, where $a2^{k-1} \leq b \leq (a+2)2^{k-1}-1$.

Set all these bits to 1's.

Step 3 : Allocate processors with address $G_n(b \bmod 2^n)$ to the request, where $a2^{k-1} \leq b \leq (a+2)2^{k-1}-1$.

Processor Relinquishment :

Reset every pth allocation bits to 0, where $G_n(p)$ is used in the subcube released.

This strategy also can be explained by the complete binary tree. This strategy recognizes $2^{n-k+1}Q_k$ within the n-cube multiprocessor and this is an improvement by a factor of two over the buddy strategy.

C. Al-Dhelaan[8]

The path from the root of the tree to any node is that node's address. This address corresponds to the subcube which consists of all the descendents processors (leaf node). Note that in Q_4 subcube 01, 01X or 01XX denotes the same subcube. Before describing the algorithms some definitions are stated first.

Definition : The αth partner of $a_{k-1},a_{k-2},...a_{\alpha+1},a_\alpha,a_{\alpha-1},...a_0$ for any $0 \le \alpha \le k-1$ is defined as

$$\begin{array}{ll} a_{k-1},a_{k-2},...a_{\alpha+1},a_\alpha,a_{\alpha-1},...a_0, & \text{if } a_\alpha = 0 \\ \text{undefined} & \text{if } a_\alpha = 1. \end{array}$$

The pth partner of $B_k(i)$ is defined as $B^p{}_k(i)$.

Definition : For any integer a, $0 \le a \le 2^{n-k+1}-1$, the node $B_{n-k+1}(a)$ is free if and only if all of its descents are free. For example, for n=4 and k=2, the node 000 is free if and only if the processors 0000, 0001 are free.

Processor allocation :

Step 1 : Set k to the dimension of a subcube required to accommodate the request.

Step 2 : Determine the least integer α, $0 \le \alpha \le 2^{n-k+1}-1$ such that $B_{n-k+1}(\alpha)$ is free and it has a pth partner $B^p{}_{n-k+1}(\alpha)$ which is also free where $0 \le p \le n-k$. Take p as small as possible.

Step 3 : Allocate these processors to the request and set their allocation bits to 1.

Processor Relinquishment :

Reset the allocation bits of all the processors that correspond to the descendents of the nodes $B_{n-k+1}(a)$ and $B^p{}_{n-k+1}(a)$ to 0.

This strategy can recognize $(n-k+1)2^{n-k}\,Q_k$ cubes.

D. A New Strategy :

In this section we present a very efficient processor allocation strategy which makes larger contiguous spaces for the new coming job than buddy, Gray strategy and Al-Dhelaan[8] do. This is a significant improvement because in practical system it is normal to have many small incoming jobs and large number of processors. Furthermore, this new strategy is suitable for static as well as dynamic processors allocation and it results in a less fragmentation and higher fault tolerance.

The new strategy can be described by the following two parts.

Processor Allocation:

```
Step 1 : Set k := |Ij|, where |Ij| is the dimension of a subcube required to accommodate the
         request Ij.
Step 2 : Get one possible link combination(in order) in the link table and Find the partner
         processors
         Check if those processors are available or not
         If operation succeed then go to step 3
                                   else go to step 2
Step 3.  Allocate nodes.
```

Processor Relinquishment :

Reset every allocation nodes.

This allocation strategy is different from 3 strategies mentioned earlier. Though all three strategies can be explained by the binary tree, tree structure may not express all the link connections in the n-cube. So, instead of using tree structure, we allocate the processors for the incoming job using index scheme from the link table.

Because of its enhanced subcube recognition ability, the new strategy can allocate subcubes more densely at one end, thus making larger subcubes available at the other end for future use. An allocation strategy is said to be statically optimal if a Q_n using the strategy can accommodate any input request sequence $\{I_i\}$ iff $\sum_{j=1}^{k} 2^{|I_i|} \le 2^n$, where $|I_i|$ is the subcube dimension required by request I_i. The buddy and Gray strategies are statically optimal[4]. Also the new strategy is statically optimal.

Theorem 2.3: The new strategy is statically optimal.

An example of the buddy, Gray, Al-Dhelaan[8] and the new strategies is given in Table 2.3 where the input sequence is as follows.

$I_1=Q_0$ $I_3=Q_0$ $I_5=Q_1$ $I_7=Q_0$

$I_2=Q_2$ $I_4=Q_0$ $I_6=Q_2$ $I_8=Q_1$

#	Buddy system	Gray system	Ref[8]	New system
0.	0000 ----I_1	0000 ----I_1	0000---- I_1	0000 ----I_1
1.	0001 ----I_3	0001 ----I_3	0001---- I_3	0001 ----I_2
2.	0010 ----I_4	0011 ----I_2	0010---- I_2	0010 ----I_3
3.	0011 ----I_7	0010 ----I_2	0011---- I_2	0011 ----I_2
4.	0100 ----I_2	0110 ----I_2	0100---- I_4	0100 ----I_4
5.	0101 ----I_2	0111 ----I_2	0101---- I_5	0101 ----I_2
6.	0110 ----I_2	0101 ----I_4	0110---- I_2	0110 ----I_5
7.	0111 ----I_2	0100 ----I_5	0111---- I_2	0111---- I_2
8.	1000 ----I_5	1100 ----I_5	1000---- I_6	1000 ----I_6
9.	1001 ----I_5	1101 ----I_7	1001---- I_6	1001 ----I_6
10	1010 ----I_8	1111 ----I_6	1010---- I_6	1010 ----I_6
11	1011 ----I_8	1110 ----I_6	1011---- I_6	1011 ----I_6
12	1100 ----I_6	1010 ----I_6	1100---- I_7	1100 ----I_7
13	1101 ----I_6	1011 ----I_6	1101---- I_5	1101 ----I_8
14	1110 ----I_6	1001 ----I_8	1110---- I_8	1110 ----I_5
15	1111 ----I_6	1000 ----I_8	1111---- I_8	1111 ----I_8

Table 2.3 Comparison among 4 different allocation strategies

It can be observed that the new strategy outperforms the buddy strategy, the GC strategy and Al-Dhelaan[8] in the first-fit search and will pack incoming request more densely, thus making larger contiguous regions available than the buddy strategy, the GC strategy and Al-Dhelaan[8] can.

The subcube recognition problems becomes more important when considering some faulty processors. In these situations the new strategy does better than the above strategies as illustrated in the following example.

Example: (*Fault tolerance*)

In a 4-cube multiprocessor if two nodes, one from (0000, 0001) and the other from (1000, 1001) are faulty. Then neither the buddy system allocation strategy nor Gray code strategy will be able to satisfy the requests $\{I_1=Q_3, I_2=Q_2\}$ but new strategy will satisfy this. When (0000, 1000) are faulty, (1,3,5,7,9,11,13,15) for Q_3 and (4,6,12,14) for Q_2 are assigned.

When processor relinquishment is taken into account, the buddy strategy and the GC strategy is shown to be poor in recognizing the availability of subcubes in the n-cube multiprocessor, and the processor utilization is thus degrade. But the new strategy does better than those strategies as illustrated in the following example.

Example: (*Dynamic allocation*)

Consider the request $\{I_1=Q_1, I_2=Q_2, I_3=Q_1, I_4=Q_3\}$. Let processors{0,1} and Processors {4,5} be allocated for I_1 and I_3, respectively. If I_1 and I_3 released their processors and others do not then using the buddy system strategy or the Gray code strategy a request like $\{I_5=Q_2\}$ will not be satisfied. But new strategy will combine the two released Q_1s into a Q_2 and allocate it to I_5. When I_1 and I_3 released their processors, we can allocate {0,1,4,5} for $\{I_5=Q_2\}$.

2.5 Analysis of Algorithm

In this section we describe the algorithm explained in previous sections. In sequential version of our algorithm, we get $O(2^k * {}_nC_k\, 2^{n-k})$ time complexity. A formal description of our algorithm as follows.

```
Algorithm allocation;
 (tindex=0; tindex<2^subcubes; tindex++)
        get_combination(Qn, Qk, tindex);
get_combination(Qn, Qk, tindex)
    find all kinds of combination in tindex-th row in the link   table;
     (i=0; i< rlink; i++) /* rlink is the number of links in the table index */
        index=0;
        find_one_cube(0, tindex, n);

find_one_cube(pos,tindex, size)         /* find the partners  */
        if (pos <size) {
            if (pos == 0) {
                path[index] = table[tindex][temp[pos]].no;
                if (table[path[index]][0] == ON)
                        return(FAILURE);
                    index++;
                if (index== exp(subcubes))
                    print path;             /*  print result when one cube is found */
                return(find_one_cubes(pos+1,tindex,size);
            }
          else {
                path[index] = table[tindex][temp[pos]].no;
                if (table[path[index]][0].duty == ON)
                        return(FAILURE);
                    index++;
                if(find_one_cube(0,table[tindex][temp[pos]].no, pos) == SUCCESS)
                    return(find_one_cube(pos+1,tindex,size));
                else
                    return(FAILURE);
          }
        }
```

It can be parallelized resulting in $O({}_nC_k)$ time complexity as shown below. A further advantage of our parallel allocation algorithm is that they are dynamic and require little storage. The algorithm is shown in c style with added constructs, "par" and "seq" like those of the parallel language Occam. Here is the parallel version of our algorithm.

```
par (tindex=0;tindex<$2^n$; tindex++)
        get_combination($Q_n$, $Q_k$, tindex);

get_combination(Qn, Qk, tindex)
        seq
           find all kinds of combination in tindex-th row in  the link table;
        par  (i=0; i< rlink; i++) /* rlink is the number of links in the table index */
             index=0;
             find_one_cube(0, tindex, combination);

find_one_cube(pos,tindex, size)        /* find the partners  */
        par
            seq
               path[index] = table[tindex][temp[pos]].no;
               index++;
               if (index== $Q_k$2)
                        print path;        /*  print result  when one cube is found */
            par(pos=1;pos<size;pos++1)
               path[index] = table[tindex][temp[pos]].no;
               index++;
               find_one_cube(table[tindex][combination[pos].no],pos);
```

2.6 A New Approach to Task Migration

Even though enough number of hypercube nodes are available for the incoming job, allocation and deallocation of subcube usually result in a fragmented hypercube. That is, they don't form the recognizable subcube to accommodate an incoming job. The fragmentation problem in a hypercube can be solved by task migration, i.e., relocating tasks within the hypercube to remove the fragmentation.

Fig. 2.3 shows an example of a fragmented hypercube where four available nodes{010,011,110,100} can't form a Q_2 to be used: thus, when a task requiring a Q_2 arrives, it has to be either queued or rejected.

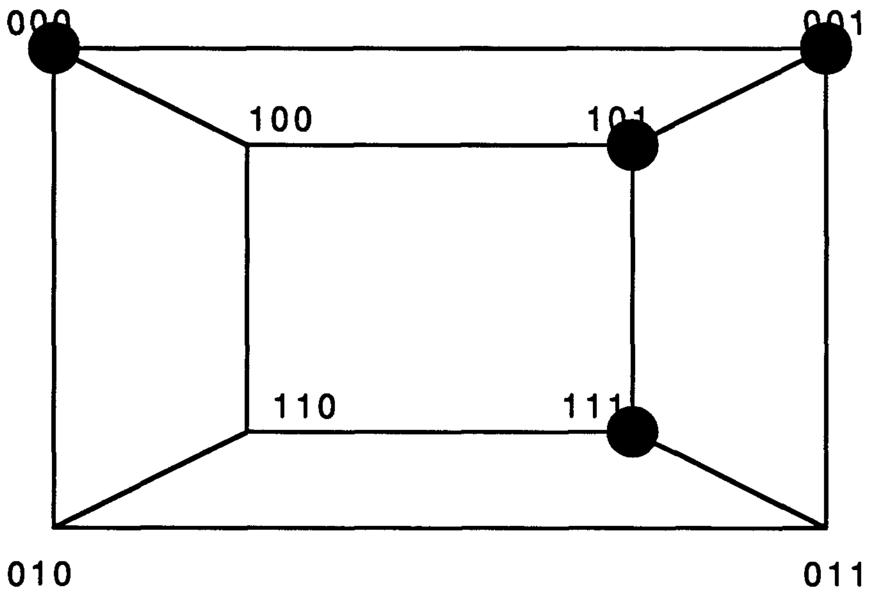


Fig. 2. 3 An example of hypercube fragmentation

Such fragmentation leads to poor utilization of hypercube nodes, thus limiting the improvement achieved by the new strategy. Fragmentation problem in conventional memory allocation can be handled by memory compaction. Also the fragmentation problem in a hypercube can be solved by *task migration*[4], i.e., relocating active tasks and compacting those within the hypercube at one end in order to make enough subcubes available for the incoming request. There is a close relationship between allocation strategy used and task migration, because active tasks must be relocated to where allocation strategy can recognize.

A collection of occupied subcubes is called a configuration. We first find the goal configuration so that a given fragmented hypercube must change its position by relocating active tasks. When a task is allocated to a subcube, the portion of the task located at each hypercube node of this subcube is called a *task module*[4].

A moving step is called H(source node, neighboring node) = 1. The cost of each task migration is then measured in terms of Hamming distance required while task migrations between different pairs of source nodes and destination nodes are performed in parallel. In order to move tasks in parallel, it is very important to avoid deadlock during task migration.

We formulate the node-mapping between each pair of source and destination node in such a way that the Hamming distance(source,destination) is minimized and develop a routing algorithm for shortest deadlock-free paths for task migration.

We assume that the hardware of the hypercube system under consideration is designed in such a way that each hypercube node has separate input and output ports. So, each node can receive a task module while sending another task module to its next hop. One time unit is defined as each moving step which will take the same amount of time.

In the following section we shall determine the goal configuration, the node-mapping between the source and destination subcubes, and shortest deadlock-free paths for task migration.

A. Determination of Goal Configuration

Since task migrations between different pairs of source nodes and destination nodes are performed in parallel, it is very important to avoid any deadlock during the migration. A deadlock might occur if there is a circular wait among nodes. To prevent this, a linear ordering of hypercube nodes is established in such a way that a node with address $G(B_n(p))$ sends its task module to another node with address $G(B_n(q))$ if and only if $p>q$. Thus, we can avoid the any circular wait. The goal configuration without fragmentation can be determined by the allocation algorithm developed in section IV.

Given a configuration of occupied subcubes, we do the following steps.

step 1: Label each task in the availability list with a distinct number

step 2: Relocate all tasks according to an increasing order of their labels.

We can compare the goal configuration without fragmentation between Ref[4] and the new strategy developed in Section 2.4 in Table 2.4.

Gray strategy			New strategy		
#	Before	After	#	Before	After
0. 0000		-- task 1	0. 0000		-- task 1
1. 0001		-- task 4	1. 0001		-- task 2
2. 0011	-- task 1	-- task 2	3. 0010	-- task 1	-- task 3
3. 0010		-- task 2	2. 0011		-- task 2
4. 0110		-- task 2	7. 0100		-- task 4
5. 0111		-- task 2	6. 0101		-- task 2
6. 0101	-- task 2	-- task 3	4. 0110		-- task 3
7. 0100	-- task 2	-- task 3	5. 0111		-- task 2
8. 1100	-- task 2		15. 1000	-- task 2	
9. 1101	-- task 2		14. 1001	-- task 2	
10. 1111			12. 1010	-- task 2	
11. 1110	-- task3		13. 1011	-- task 2	
12. 1010	-- task3		8. 1100	-- task 3	
13. 1011			9. 1101	-- task 3	
14. 1001	-- task4		11. 1110		
15. 1000			10. 1111	-- task 4	

Table 2.4 Task migration under the GC strategy and New strategy

B. Node mapping Between Source and Destination Node

After the goal configuration is determined, each active task will be moved from its source subcube to the destination subcube. The minimal number of moving steps required to move a task from one node to another node can be determined by Hamming distance between the two node locations.

We can define that the shortest distance between the source, p and destination, q subcube is the Hamming distance(p,q). The order of source in the new strategy is not necessarily the same as their corresponding destination nodes after the node-mapping. For example, if we have the same order between the source and destination node for task 2, we have the following node-mapping.

task 2 : 1000 -> 0001, H(p,q) = 2;   1001 -> 0011, H(p,q) = 2;

1010 -> 0101, H(p,q) = 4;   1011 -> 0111, H(p,q) = 2;

Therefore it will take 4 steps even though all moving can be done in parallel.

If we adjust the order in the goal configuration, we will reduce the steps. For example, we have a different following node-mapping.

task 2 : 1000 -> 0001, H(p,q) = 2;   1001 -> 0101, H(p,q) = 2;

1010 -> 0011, H(p,q) = 2;   1011 -> 0111, H(p,q) = 2;

In this case all nodes have Hamming distance 2, resulting in 2 steps. Here, we want to use the Theorem developed in [4]. The node-mapping between two subcubes recognizable by the new strategy can be determined as follows: Suppose $\alpha=a_n a_{n-1}\ldots a_1$ is the source subcube and $\beta=b_n b_{n-1}\ldots b_1$ is the destination subcube. Let p and q be the dimension in which $a_p \in \{0,1\}$ and $b_p=*$ and $a_q=*$ and $b_q \in \{0,1\}$.

Theorem 2.4[4] : Each source node $u=u_n u_{n-1}\ldots u_1 \in \alpha$ can be one-to-one mapped to a destination node $w=w_n w_{n-1}\ldots w_1 \in \beta$ in such a way that

$$\text{when } i \neq p,\ w_i = \begin{cases} b_i & \text{if } b_i \in \{0,1\}, \\ u_i & \text{if } a_i = b_i = *, \end{cases}$$

$$\text{when } i = p,\ w_p = \begin{cases} u_p(\text{negate}) & \text{if } w_q=u_q, \\ u_p, & \text{if } w_q \neq u_q. \end{cases}$$

When modules of a task are migrated in parallel, the moving distance between two nodes is equal to the number of Hamming distance between the source and destination node of a task.

B. Determination of Shortest Deadlock-Free(SDF) Routing

Now we want to develop a routing method to move each task module from its source node to its destination node. In order to avoid deadlock, a linear ordering among hypercube nodes is needed such that each node can only move its task module to a node with a lower address. So, we give all the nodes the value of Gray code to corresponding to the original binary node.

If $G_n(B_n(i))$ and $G_n(B_n(j))$ are two nodes in a SDF path, then $G_n(B_n(i))$ is ahead of $G_n(B_n(j))$ in the path iff $i>j$.

For example, {1111(10)->1101(9) ->1100(8)->0100(7)} is a SDF path in G_4, whereas {1111(10) -> 0111(5) -> 0101(6) -> 0100(7)} is not.

Once the node-mapping between each pair of source and destination subcubes is determined, each source node appends to its task module the address of its destination node. Each node can then determine the next hop on which to route a task module by the algorithm below[4].

```
Step 1 : Each node compares the destination address $d=d_nd_{n-1}...d_1$ with its own
         address $s=s_ns_{n-1}...s_1$ from left to right. Let the j-th and k-th dimensions
         be respectively the first and second dimensions in which they differ.
step 2: If $\sum_{i=k}^{j-1} s_i$ is even then send the task module to a neighboring node along the
             k-th dimension
        else send the task module to a neighboring node along the j-th dimension.
```

For example, suppose the source node is $B_4(15)$= 1111 and the destination node is $B_4(4)$ = 0100, then j=4 and k=2. The next determined by above algorithm is $B_4(13)$=1101 since $\sum_{i=2}^{3} s_i$ is odd. Then the next hop by the intermediate node $B_4(13)$=1101 is $B_4(1100)$

then final destination (0100). is reached. It can be verified that [1111(10) -> 1101(9) -> 1100(8) -> 0100(7)] is a SDF path.

Theorem 2.5[4] : The path determined by above algorithm is SDF.

To illustrate the entire process of task migration, consider the fragmented configuration in Table 4. From above theorem, we obtain the goal configuration. By the node-mapping scheme developed above, we have 0001->0000 for task 1,1000->0001, 1001-> 0101, 1010->0011, 1011->0111 for task 2, 1100->0010,1101->0110 for task 3, 1111-> 0100 for task 4. The SDF routing can then determined by above algorithm as follows.

task 1: 0001-> 0000

task 2: 1000 ->1001->0001, 1001->1101-> 0101,

1010 ->0010->0011, 1011->1111-> 0111

task 3: 1100 ->0100-> 0110->0010, 1101-> 0101-> 0111->0110

task 4: 1111 ->1101->1100->0100

2.7 Half-Task Migration Under Processor Failure

One approach to achieve fault tolerance is to decompose the hypercube structure hierachially and add redundancy at several levels. This approach requires a global reconfiguration algorithm in which a global controller reconfigures a set of cross-bar switches. It also does not take full advantage of the available hardware because a given module at a specific level may be replaced by a spare module even when most of its components are functioning properly. Another approach to achieve fault tolerance where degraded performance is not allowed is to initially designate only some of the processors as active and designate the rest as spares that may cover for faulty processors. Such approach is only useful for applications that require a number of processors less than the number of processors in the available hypercube.

In this section we describe the *half-task migration* in the presence of faulty processors. The *half-task migration* is defined as follows: When 2^k processors are allocated for the job in an n-cube, we can relocate active tasks from 2^{k-1} processors to another 2^{k-1} processors in order to continue the job in case of processors failure. Note that there is a strong dependence of *half-task migration* on the subcube allocation strategy used, since active tasks must be relocated in such a way that the availability of subcubes can be detected by the new allocation strategy.

The procedure can be done as follows: 1) determination of a goal configuration, 2) the node-mapping between the source and destination subcubes, and 3) determination of the shortest routing for moving half-task modules. This approach has the following advantages: 1) We don't have to have the extra processors to reconfigure[15]. 2) It is easy and efficient to reconfigure the processors if the alternatives are chosen.

We assume that the hardware of the hypercube system under consideration is designed in such a way that each hypercube node has separate input and output ports. So each node can receive a task module while sending another task module to its next hop. Each moving step will take the same amount of time and will be used to define one time unit.

A. Goal configuration

When 2^k processors are allocated for the job in an n-cube, we can relocate active tasks from 2^{k-1} processors to another 2^{k-1} processors so as to continue the job in case of processors failure. Given the configuration of 2^{k-1} faulty subcubes, the goal configuration can be determined by the algorithm below. There are n-k alternatives for Q_k in Q_n.

<u>Algorithm A2 : Determination of the goal configuration</u>

```
A2 :  (tindex=0; tindex<$2^n$; tindex++)
              choose smallest tindex in $Q_k$
              get_combination($Q_n$, $Q_k$, tindex);
get_combination($Q_n$, $Q_k$, tindex)
    find every combination in tindex-th row in the link table;
    (i=0; i< rlink; i++)  /* rlink is the number of links in the table index */
        index=0;
        find_one_cube(tindex, $Q_k$, combination);

find_one_cube(tindex,size, combination)
    path[index] = table[tindex][combination[0]].no;
    index++;
    if ((index ==$Q_k^2$) and (path is not faulty))
        print path;  /* print result when one cube is found and non-faulty */
    (pos=1; pos<size; pos++)
      path[index] = table[tindex][combination[pos]].no;
      index++;
      find_one_cube(table[tindex][combination[pos]].no,pos);
```

For example, processors {0,1,4,5} are allocated for task which requires Q_2 in Q_4. Let processors {4,5} be faulty during execution. We can replace processors {4,5} with processors {2,3} or {8,9} . Thus we can have processors {0,1,2,3} or {0,1,8,9} in order to continue the job. We check processors {2,3} and {8,9} in sequence if they are available.

B. Node-Mapping

After the goal configuration is determined, 2^{k-1} processors will be moved from their source subcube to the destination subcube. The minimal number of moving steps required to move 2^{k-1} processors location to another location is determined by the Hamming distance between the two subcube locations.

We have the following theorem for the minimal number of moving steps required to move a task from one subcube location to another.

Theorem 2.6 : The order of source subcubes must be the same as their corresponding destination subcubes after node mapping.

Proof : Let $a_1\ a_2\ a_3\ \ldots a_n$ be the 2^{k-1} faulty source subcubes and $b_1\ b_2\ b_3 \ldots b_n$ be the alternative 2^{k-1} destination subcubes and $c_1\ c_2\ c_3\ \ldots c_n$ be the 2^{k-1} remaining source subcubes. There are $2^{k-1}!$ ways to map from source subcube to the destination subcube. Here we can find $H(a_1, c_1) = 2$, $H(a_2, c_2) = 2$, $H(a_n, c_n) = 2$. Thus, in order to minimize the moving steps, we must have $H(b_1, c_1) = 2$, $H(b_2, c_2) = 2$, $H(b_n, c_n) = 2$. Therefore, we have to move the half-task in the same order as their corresponding destination subcubes. That is, $a_1 \rightarrow b_1$, $a_2 \rightarrow b_2$, $\ldots$ $a_n \rightarrow b_n$.

<u>Corollary 2.1:</u> The maximum Hamming distance between source processor and destination processor in *half-task migration* is 2.

According to above theorem, we can choose only one way which minimizes the Hamming distance. For example, processors {2,3,6,7,10,11,14,15} are allocated for Q_3. Assume that processors {6,7} are faulty during the execution. Then according to the goal configuration, we can find processors {8,9,12,13} available. Here we have $2^{3-1}!$ ways to map from processors {2,3,6,7} to {8,9,12,13}. If we choose 2->13, 3->9, 6->12, 7->8 respectively, then we have H(0010,1101)=4, H(0011,1001)=2, H(0110,1100)=2, H(0111,1000)=4. Therefore it will take 4 steps even though all moving can be done in parallel. But according to the above theorem, we have H(0010,1000)=2, H(0011,1001)=2, H(0110,1100)=2, H(0111,1101)=2. Thus, it will take only 2 steps.

C. Shortest Routing Procedure

Now we want to develop a routing method to move each half-task module from the source node to its destination node. Once the node-mapping between each pair of source and destination subcubes is determined, each source node appends to its task module the address of its destination node. Each node can then determine the next hop on which to route a half-task module by the algorithm below.

Step 1 : Each node compares the destination address $d=d_n d_{n-1} \ldots d_1$ with its own address $s=s_n s_{n-1} \ldots s_1$ from left to right. Let the j-th and k-th dimension be respectively the first and second dimensions in which they differ.

Step 2: if the j-th dimension in the source processor is 0 then send the task module to a neighboring node along the j-th dimension else send it to a neighboring node along along the k-th dimension.

<u>**Theorem 2.7:**</u> The path determined by the above algorithm is the shortest safe-path.
proof : Let $a_1\ a_2\ a_j\ \ldots a_k \ldots a_n$ be the source subcubes and $b_1\ b_2\ b_j \ldots b_k \ldots b_n$ be the destination subcubes. In corollary 2.1, we described that the maximum Hamming distance between source processor and destination processor is 2. Thus, when a_j is 0, we have to send the task along j-dimension.

For example, when processors {0,1,4,5} are assigned for Q_2, processors {4,5} are faulty during the execution. If we have *half-task migration* from processors {4} to {2} and {5} to {3} in the goal configuration, then we have 0100 -> 0110 -> 0010 and 0101 -> 0111 -> 0011. If we don't follow the above procedure, we may have 0100 -> 0000 -> 0010 and 0101 -> 0001 -> 0011. Then we give unnecessary interrupt to processors {0,1}.

REFERENCES

1. K. Hwang and F.A. Briggs, Computer Architecture and Parallel Processing, New York: McGraw_Hill, 1984

2. R. M. Chamberlain, "Gray codes, Fast Fourier Transformations and Hypercubes", Parallel Computing, 6, 1988, pp. 225-233.

3. M. Chen and K.G. Shin, "Processor Allocation in an N-cube Multiprocessor Using Gray Codes", IEEE Trans. Computer, Dec. 1987 pp. 1396-1407.

4. M. Chen and K.G. Shin, "Task Migration in Hypercube Multiprocessor", Proc. 16th Annual Int'l Symp. on Computer Architecture. Jun 1989, pp. 105-111

5. M. Chen and K.G. Shin, "Embedment of interesting task modules into a hypercube multiprocessor", in Proc. Second Hypercube Conf., Oct 1986, pp. 121-129

6. B. Becker and H.U. Simon, "How robust is the n-cube?", in Proc. 27th Ann. Symp. Foundations of Comp. Sci. Oct. 1986 pp. 283-291.

7. H. P. Kattesff, "Incomplete hypercubes", IEEE Trans. Computer, May 1988, pp. 604-608.

8. A. Al-Dhelaan and B. Bose, "A New strategy for Processor Allocation in an N-cube Multiprocessor", Phoenix Conference on Computer and Communication, Mar 1989. pp. 114-118.

9. A. Al-Dhelaan and B. Bose, "Efficient Fault Tolerant Broadcasting Algorithm for the Hypercube", Proc. The fourth Conf. on Hypercube Concurrent Comp. and Applications, Monterey, Mar 1989, pp. 123-128.

10. P. Ramanathan and K.G. Shin, "Reliable Broadcasting in Hypercube Multicomputers", IEEE Trans. on Comp. Dec 1988, pp 1654-1657.

11. Y. Saad and M.H. Schultz, "Topological Properties of Hypercubes", IEEE Trans. on Computer, Jul 1988, pp 867-872

12. J. E. Jang, S. W. Choi and W. K. Cho, "A New Approach to Processor Allocation and Task Migration in an N-cube Multiprocessor", Proceedings, International Conference on Supercomputing, Nov, 1989. pp. 314-325

13. J. E. Jang and W. K. Cho, "Maximality of Subcube Recognition and Fault Tolerance in an N-cube Multiprocessor", Proceedings, 4th SIAM conference on Parallel Processing for Scienctific Applications, Dec, 1989.

14. M. Sultan and Rami Melhem, "Fault Tolerance and Reliable Routing in Augmented Hyercube Architecture", Proc, 8th IEEE Phoenix Int'l Conference on Computer and Communication, Mar, 1989. pp. 19-23.

15. H. P. Kattesff, "Incomplete hypercubes", IEEE Trans. Computer, May 1988, pp 604-608.

16. Z. Kohavi, Switching and Finite Automata Theory, New York: McGraw-Hill, 1978

Chapter 3

An Optimal Fault-Tolerant Broadcasting Algorithm for a Hypercube Multiprocessor

3.1 Introduction

Rapid advancing technology has made it possible for a large number of processing elements(PEs) to be interconnected together on a single chip as a viable means of implementing high performance integrated systems. A number of parallel architectures have been proposed, such as hypercubes, meshes, trees and cube-connected-cycles(CCC)[1-4]. Among them, hypercube multiprocessors have been drawing considerable attention due to their structual regularity for easy construction and high potential for the parallel execution of various algorithms. And its architecture allows high level of concurrency and efficiency. Numerous research efforts related to hypercube architectures, operating systems, etc., have been undertaken[5-15, 20-21].

Most of the research effort on hypercube architecture has focused on the fault-free situation. However, the increasing use of hypercube multicomputers for critical applications has made their fault tolerance an important issue. Efficient routing of message is a key to the performance of a multicomputer system. Especially, the increasing use of multicomputer systems for reliability-critical applications has made it essential to design fault-tolerant routing strategies for such systems. By fault-tolerant routing, we mean the successful routing of messages between any pair of non-faulty nodes in the presence of faulty components.

Broadcasting is an important means of communication among processors by which a processor can pass data or control to all other processors in the network. This operation is extremely important for diagnosis of the network, distributed agreement[16] or clock synchronization[17]. Distributed agreement and clock synchronization can be achieved only if there is no faulty node to deliver the message in the system[16-17]. This, however, is not easy to achieve in the presence of faulty node/link because the faulty nodes can either omit, corrupt, reroute, or alter information passing through them.

There are two possible approaches to overcome this problem. In the first approach, each node keeps limited information about the faulty nodes in the system. Fault-tolerant routing/broadcasting is achieved by going around the faulty nodes[9, 13]. This approach can be used only if it is possible to identify the faulty processors "on-line". Since the overhead of identifying the faulty processors and passing the fault information to the other nodes could be quite severe, this approach is not suitable for many real-time applications. In the second approach, fault tolerance is achieved by sending multiple copies of the message through disjoint paths[14-15]. The nodes that receive the message identify the original message from the multiple copies by using some scheme that is appropriate for the fault model, e.g., majority voting. The second approach has the advantage of not having to identify the faulty processors.

Sullivan and Bashkov[5] developed an algorithm for broadcasting in the hypercube. This algorithm was developed on the assumption of having no faulty processors. Al-Dhelaan[13] developed a broadcasting algorithm for the hypercube in the presence of some faulty processors. However, their algorithm works if only one child processor is faulty under any node in the broadcasting tree. Their algorithm does not make explicit use of the properties of the hypercube topology. Ramanathan and Shin[14] developed another algorithm for the hypercube in the presence of faults which uses the second approach mentioned above.

In this chapter, we develop an optimal fault tolerant broadcasting algorithm for the hypercube multicomputers. In other words, our algorithm can tolerate n-1 processors failure in Q_n and uses the first approach where each processor keeps a small amount of information about other nodes.

This chapter is organized as follows. Section 3.2 describes the preliminaries, problem statement and notation used in this paper. Section 3.3 outlines the previous broadcasting algorithm developed by Sullivan[5] and Al-Dhelaan[13]. Section 3.4 describes the proposed an optimal fault-tolerant broadcasting algorithm. The algorithm developed in Section 3.4 is formally proved to be optimal and is evaluated in terms of steps required for complete broadcasting in Section 3.5.

3.2 Preliminaries and Problem Statement

A n-cube can be defined as follows:

Definition[18]: An n-cube Q_n is defined recursively as

a) Q_0 is a trivial graph with one node, and

b) $Q_n = K_2 * Q_{n-1}$, where K_2 is the complete graph with two nodes.

The problem addressed in this chapter can be easily stated as follows. Given 1) an n-dimensional hypercube subject to node faults, 2) maximum n-1 node faults, develop a broadcasting algorithm that satisfies the following condition.

Condition : If the node initiating the broadcasting is non-faulty, then all non-faulty nodes in the hypercube must receive the message broadcasted by the initiating node.

We refer to the processors of a multiprocessor as nodes, and the communication links connecting these processors as links. The processors communicate by sending messages over the links and this could be direct or indirect, i.e., through some intermediate processors.

In our algorithm, we use two types of information to control the flow of the message in the hypercube.

- $\oplus_i(s)$: initiating node s sends the message to *i*th neighbor node by complementing ith position bit.

- $\oplus_{(i,j)}(s)$: initiating node s sends the message, "$\oplus_i(s)$ is faulty" to $\oplus_j(s)$

We will show the example to use the above notation.

Example: Consider a hypercube of 3-dimension with eight nodes. The initiating node 000(0) can send a message to its 2nd neighbor, i.e., processor 100(4) by executing $\oplus_2(000)$. Also, the processor 000(0) can send a message, "node 010(2) is faulty" to its neighbor node 100(4) by executing $\oplus_{(1,2)}(000)$. Thus, node $\oplus_2(000)$ gets the information that the node $\oplus_1(000)$ is faulty.

3.3. Previous Fault-Tolerant Broadcasting

We define N(T) to be the total number of nodes in the broadcasting tree.

Sullivan and Bashkov have devised an algorithm for broadcasting in the hypercube[5]. This algorithm sends the message to all other nodes non-redundantly, which means that broadcasted message is sent to each processor exactly once. The algorithm takes $\log_2(N)$ steps to broadcast the message. It works by sending a weight along with each message; this weight is used to decide how the algorithm should continue broadcasting the message from the receiving node.

The route that the broadcasted message will take can be shown using a tree where the nodes and arcs of the tree correspond to the nodes and links of the hypercube respectively. Furthermore, the root of the tree represents the source, i.e., originator of the broadcasted message.

We will briefly describe the algorithm developed by Al-Dhelaan[13] which tolerates the existence of some faulty processors.

Definition[13] : When any node receives a paired weight(i,j) it interprets it as:

1. Take i as the weight and send the message according to direction *i*.
2. Send the message with a singular weight i to your jth neighbor. i.e., execute Send(message,(i,j)).

Example: Consider a hypercube of 3-dimensions with eight nodes. The processor 000 can send a message to its 2nd neighbor, i.e., processor 100, via the 2nd link with weight 1 by executing Send(message,1,2).

The algorithm starts at the source node using the following steps[13].

```
Generate the message
FOR j = 0 to ($\log_2(N)$ -1) DO
    IF for some i>j the ith neighbor is faulty
        THEN Send (message,j,(j,i))
        ELSE Send (message,j,j)
```

and the other processor needs to follow the steps using the following steps[13].

```
Extract the message and process locally
IF the node weight is paired
    THEN BEGIN
            Let the paired weight be (a,b)
            Send (message, b,a)
            Set weight to a
        END
FOR j = 0 to (weight -1) DO
    IF the ith neighbor is faulty
        THEN Send(message, j,(j,i))
        ELSE Send(message, j,j)
```

Their algorithm does not make explicit use of the properties of the hypercube topology. It works only if there is one single faulty processor under any node in the broadcasting tree. For example, Fig. 3.1 shows how a message would be broadcasted from node 011 in a Q_3 where node 111 is faulty. We can see that the nodes {101, 110, 100} under the faulty node (111) can be re-broadcasted by the brother nodes (001, 010) of the faulty node. Here we can show one problem that their algorithm can't satisfy. If node 111 and 001 are faulty, how does node 011 send the message to non-faulty nodes in the broadcasting tree? We will describe the solution in section 3.4. Their algorithm gives all the brother-nodes the burden of sending the message. In next section, we describe the algorithm that gives only one brother the burden of sending the message and that can tolerate n-1 nodes failure.

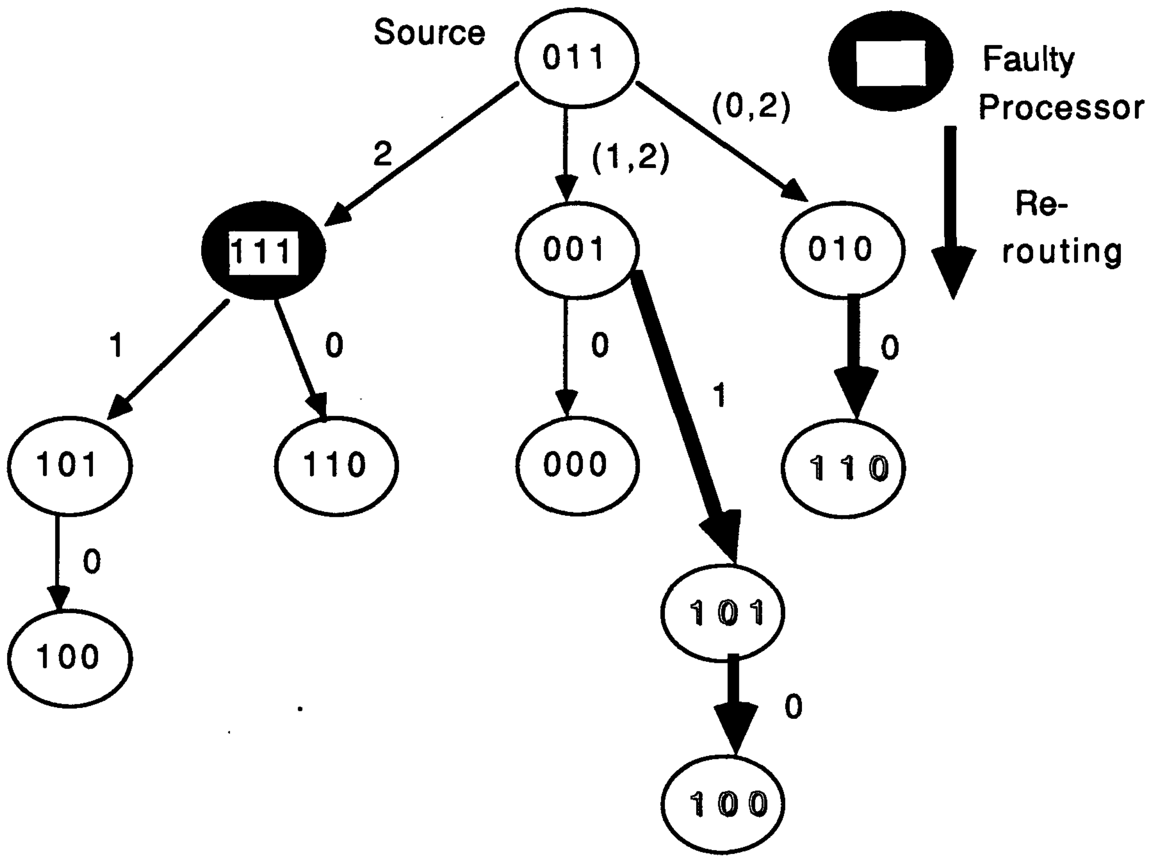


Fig. 3.1 Broadcasting in the presence of a single faulty processor[13]

3.4 An Optimal Fault-Tolerant Broadcasting

Our broadcasting algorithm is the same as the algorithm in Sullivan[5] except the sequence of the message, which is in the reverse order. Our algorithm also takes $\log_2(N)$ steps to broadcast the message. The delivery mechanism proceeds in two phases. In the first phase, the node initiating the broadcast sends the message to all its neighbors. In the second phase, the neighbors use a "*Coordinated*" procedure to broadcast the message to all the nodes. The sequence of directions used by these neighbors in their "*Coordinated*" phases is coordinated in order to ensure that each node gets the broadcasting message in sequence nonredundantly. A formal description of the algorithm is given below.

```
Algorithm Broadcast(s);     /* s: initiating node */
begin
    Generate the message
    for $0 \le i \le n-1$ do begin
        send the message from s to $\oplus_i(s)$.
        Coordinated($\oplus_i(s)$,i)
    end;
end;

procedure Coordinated (m,k)  /* m: initiating node;   $d_k$: starting direction */
begin
    R := {m};  /* R: set of nodes that have received the message */
    for each node $j \in R$
      do
        for $k+1 \le i \le n-1$  do begin
            send the message from j to $\oplus_i(j)$;
            if i < n-1 then
                R := R $\cup$ {$\oplus_i(j)$};  /* all receiving nodes are added */
        end;
        k := k+1;
        R := R -{j};  /* initiating node is deleted  */
      until R = empty;
end;
```

Let us explain the algorithm briefly. In the first phase, the source node initiating the broadcasting sends the message to all its neighbors according to the *i* direction. In the second phase, each neighbor node becomes initiating node and broadcasts the message to its children nodes according to the new direction. Whenever a new child node is broadcasted, it is added to its father node for another broadcasting. Finally, the initiating node is deleted from the all node sets, thus remaining nodes send the message to its neighboring nodes until there is no node to send the message further.

For example, Fig 3.2 shows the broadcasting of a message from source node 011 to all other processors in a Q_3.

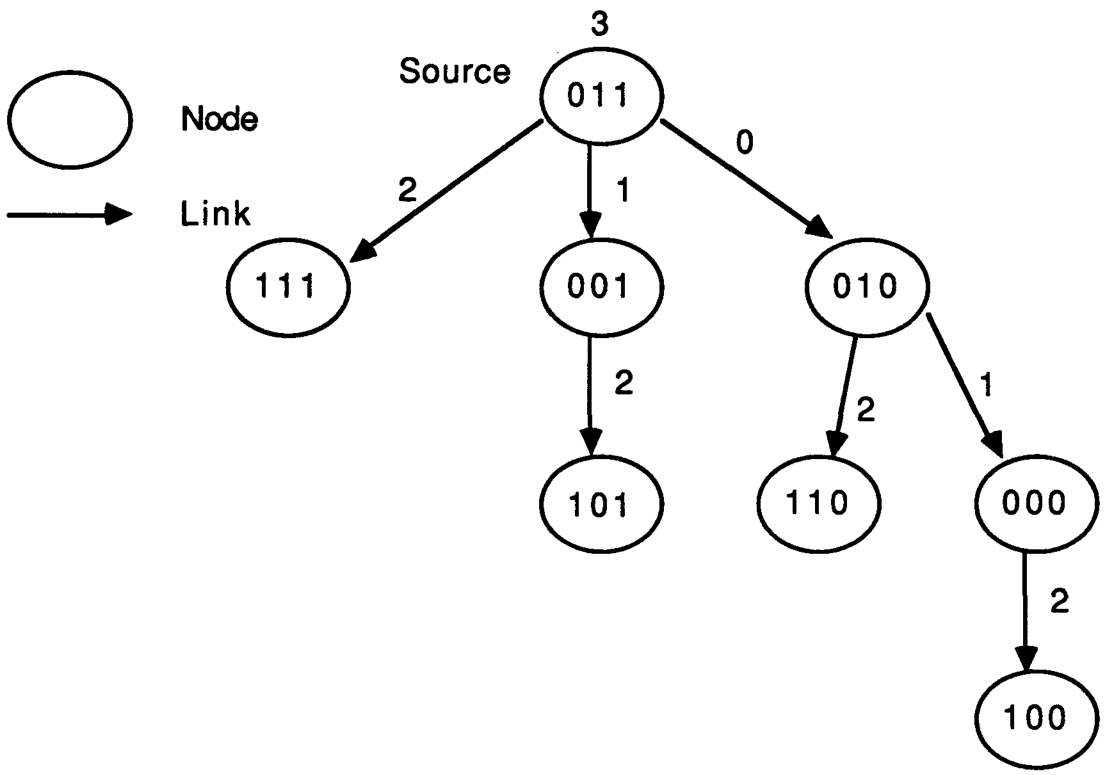


Fig 3.2. Broadcasting in a Q_3 from the node 011

Given below is an example to illustrate the basic idea of the algorithm.

Example : Consider a Q_3. Let node 011 initiate the broadcasting as shown in Fig. 3.2. In the first phase, node 011 sends the its message to nodes 010, 001, 111. In the second phase, node 010 uses procedure *Coordinated* to broadcast the message it received from the node 011 to node 000 along d_1, then to node 110 along d_2, and finally node 000 sends the message to node 100 along d_2. Similarily, node 001 sends the message to node 101 along d_2.

We describe some Lemmas that are needed to prove that the algorithm sends the message to each node non-redundantly. First, we will describe the binomial tree.

Definition[19] : Binomial trees are defined as follows: For each $k \geq 0$, we define class B_k of ordered trees as follows:

1. Any tree consisting of a single nodes is a B_0 tree.

2. Suppose that Y and Z are disjoint B_{k-1} trees for $k \geq 1$. Then the tree obtained by adding an edge to make the root of Y become the the leftmost offspring of the root of Z is a B_k tree. All binomial trees having a given index are isomorphic in the sense that they have the same shape. We have some properties of binomial trees.

Lemma 3.1[19] : Let Z be a B_k tree. Then

1. Z has 2^k nodes.

2. Z has ${}_kC_l$ nodes on level l.

Lemma 3.2 : The broadcasting tree developed by the algorithm *Broadcast* is a binomial tree.

Proof : The broadcasting tree developed by $\oplus_i(s)$ has the 2^{n-1-i} nodes in the procedure *Coordinated.* The total number of nodes of the broadcasting tree developed by $\oplus_j(s)$ where $i+1 \leq j \leq n-1$ is $\sum_{j=i+1}^{n-1} 2^{n-1-j}$. Thus, we have the the following equation, $2^{n-1-i} = \sum_{j=i+1}^{n-1} 2^{n-1-j}$ where $i+1 \leq j \leq n-1$. Also the initiating node s is connected to $\oplus_i(s)$ and $\oplus_j(s)$ where $i+1 \leq j \leq n-1$. The new initiating node $\oplus_i(s)$ is connected to $\oplus_j(\oplus_i(s))$ where $i+1 \leq j \leq n-1$. Therefore, the broadcasting tree developed by the algorithm *Broadcast* is a binomial tree.

Lemma 3.3: The nodes in the broadcasting tree are distinct among themselves.

Proof : First of all, all neighboring nodes of the originally initiating node are distinct in algorithm *Broadcast.* In "*Coordinated*" procedure, m and d_k is different whenever they are invoked by the algorithm *Broadcast.* As the value i in $k+1 \leq i \leq n-1$ changes, $\oplus_i(j)$ is also changed. Thus, all nodes are distinct in the *for-loop.* Therefore, all nodes in the broadcasting tree are distinct.

Now we want to prove the following important theorem.

Theorem 3.1: The algorithm *Broadcast* sends the message to all nodes nonredundantly.

Proof: The fact that the algorithm sends the message to all nodes in the hypercube nonredundantly can easily deduced from the following facts: a) In the broadcasting tree, each node receives the message from exactly one node, its father. b) The maximum number of nodes in a Q_n hypercube are 2^n nodes. c) From the Lemma 3.2, the broadcasting tree for Q_n has 2^n nodes, d) From the Lemma 3.3, all nodes in the broadcasting tree are distinct.

In the rest of this section we introduce a fault-tolerant broadcasting algorithm in the presence of n-1 processors failure in Q_n. The basic idea of our algorithm is as follows. The node *s* that wants to broadcast a message checks $\oplus_i(s)$ where $0 \le i \le n-1$ and sends the message if it is non-faulty. Otherwise it sends $\oplus_{(i,j)}(s)$. Then $\oplus_j(s)$ will be the new initiating node and send the message to the broadcasting tree developed by $\oplus_i(s)$ in the procedure *coordinated.* The neighbors also follow the same procedure as the initiating node. A formal description of the algorithm is given below.

```
Algorithm Broadcast(s);               /* s: initiating node */
begin
    for $0 \le i \le n-1$ do begin
      if $\oplus_i(s)$ is faulty then begin       /* checking neighboring node */
        p := i+1;
        true := 1;
        while ((p $\le$n-1) and (true)) do
          begin
            if $\oplus_p(s)$ is non-faulty then  /* another neighboring node */
              begin
                while (p$\le$n-1) do begin
                  if $\oplus_p\oplus_i(s)$ is non-faulty then   /* one of son-nodes
                                                   under faulty node */
                    begin
                      send the M from s to $\oplus_{(i,p)}(s)$;
                      send the M from $\oplus_p(s)$ to $\oplus_p(\oplus_i(s))$;
                      Fault-Tolerant-Source($\oplus_p(\oplus_i(s))$, i,p);
                      true := 0;
                    end;
                  else  p:= p+1;
                end;

                if (true = 1) then  /* father node and all its son nodes are
                                       faulty */
                    begin
                      send the M from $\oplus_i(s)$ to s;
                       for i+1$\le$j$\le$n-2 do begin
```

```
                                        send the M from s to $\oplus_j(s)$
                                        All-Sons-Dead($\oplus_j(s)$,i);
                                    end;
                                end;

                        end;
                      else p := p+1;
                  end;
                if (true = 1) then /* all $\oplus_i(s)$ nodes are faulty */
                  begin
                      send the M from $\oplus_i(s)$ to s;
                      for $i+1 \le j \le n-2$ do begin
                          send the M from s to $\oplus_j(s)$
                          All-Sons-Dead($\oplus_j(s)$,i);
                      end;
                  end;
        else begin
                send the M from s to $\oplus_i(s)$.
                alive := $\oplus_i(s)$;
                Coordinated($\oplus_i(s)$,i)

                if younger-brother-nodes are faulty then
                      Younger-Brother-Dead(alive,i);

                if all-sons-dead are faulty in the younger brother node then
                      Give the Information from the older-brother node to them.
              end;
    end;
end;
```

Let us explain this procedure briefly. First, the initiating node s checks its $\oplus_i(s)$ where $0 \le i \le n-1$. If that node is faulty, then check $\oplus_{i+1}(s)$, $\oplus_{i+2}(s)$, . . . $\oplus_{n-1}(s)$. One of them is non-faulty, s gives the information "$\oplus_i(s)$ node is faulty" to non-faulty node, that is, $\oplus_{(i,j)}(s)$. Then non-faulty node, $\oplus_j(s)$, sends the message to one of son-nodes of $\oplus_i(s)$, which is non-faulty node and becomes the new initiating node to send the message to the remaining nodes in the subtree under the fault-node. Here we have 2 more procedures to handle : 1) In case that all son-nodes are faulty under the initiating node, which can be shown in Fig. 3.6. 2) In case that $\oplus_i(s)$ is non-faulty but $\oplus_j(s)$ is faulty where $j > i$, which can be shown in Fig. 3.4.

Now we can explain the *Fault-Tolerance-Source* procedure under the new initiating node in case of node failure. At that time, the broadcasting tree under the faulty node is re-organized. The direction of the new initiating node is different from fault-free broadcasting. The number of son-nodes of the new initiating node is one less than that of the faulty node because the original initiating node is faulty. Here we assume that there is no faulty node under the new initiating node. In procedure *Fault-Tolerance-Source*, if the link of the new initiating node is j, the direction is j+1, j+2, ..,i, .. . j-1, where i is greater than faulty direction

```
procedure Fault-Tolerance-Source(s',d,k);  /* k : non-faulty direction
                                              d : faulty direction  */
begin
        for  k+1 mod n≤i≤k-1 mod n do
             begin
                    if i >  d then
                             Send the message from s' to $\oplus_i(s')$;
             end;
        Fault-Coordinated($\oplus_{k-i}(s')$,k-i,k,d)
end;
```

In this procedure its neighboring nodes are different from the normal broadcasting. You will find the difference in Fig. 3.3 and Fig. 3.4. In Fig. 3.3, the new initiating node 0011 sends the message to $\oplus_1(0011)$ and $\oplus_2(0011)$ while the new initiating node 1001 sends the message to $\oplus_2(1001)$ and $\oplus_3(1001)$ in Fig. 3.4.

We will describe the procedure *Fault-Coordinated*, which is the procedure under the new receiving node in case of initiating node failure. We assume that there is no faulty processor, since fault checking routine is the same. The direction is as follows;
If the direction from the new initiation node is j, it broadcasts j+1, j+2, .. i, .. , bound, where i is greater than faulty direction.

```
procedure Fault-Coordinated (m,k,bound,d) /* bound: non-faulty direction,
                                             d: faulty direction */
begin
    R := {m};  /* R: set of nodes that have received the message */
    for each node $j \in R$
      do
          for $(k+1) \bmod n \le i \le bound$  do begin        /* k+1, k+2, . . . */
              if $i <> d$ then
                  send the message from j to $\oplus_i(j)$;
              if $i < bound$ then
                  $R := R \cup \{\oplus_i(j)\}$;  /* all receiving nodes are added */
          end;
          k := k+1;
          R := R -{j};  /* initiating node is deleted  */
      until R = empty;
end;
```

First of all, we will show the different directions in the Fig. 3.3 and Fig. 3.4. In Fig. 3.3, the direction is d_2 -> d_3 while the direction is d_3 -> d_1 in Fig. 3.4. Here, final direction is the same as the non-faulty node direction. For example, d_3 came from $\oplus_3(1010)$ in Fig. 3.3, while d_1 came from $\oplus_1(1010)$ in Fig. 3.4. It depends on that which node among non-faulty nodes becomes the new initiating node to send the message. One of the son nodes of the faulty node becomes the initiating node and sends the message to all the non-faulty nodes in the broadcasting tree.

When the initiating node is faulty, the subtree is reorganized under the new initiating node. In the binomial tree, Hamming distance between source node and son node is 1, while Hamming distance is 2 among son-nodes. Since one of son-nodes becomes the new initiating node, all its brother nodes can't receive the message directly from the new initiating node, leaving them the leaf nodes. We will show this example in Fig. 3.3.

Now we can explain the procedure that can handle the case where all son-nodes are faulty. Then all younger brother-nodes become the initiating nodes. After they finished normal broadcasting procedure, all nodes except the initiating node send the message to $\oplus_i$(all-nodes). We show the example in Fig. 3.6.

```
procedure All-Sons-Dead(m,i);
begin
        finish normal operation using procedure "coordinated" ;
        send the message from all-nodes to $\oplus_i$(all-nodes);
end;
```

We have another procedure *Younger-Brother-Dead* that can handle the case where younger brother nodes are faulty. That means, the node *s* sends the message to $\oplus_i(s)$, because it is fault-free node, but in case that $\oplus_j(s)$ is faulty where $j > i$, $\oplus_i(s)$ sends the message to the subtree under $\oplus_j(s)$. We can show the example in Fig. 3.4.

```
procedure Younger-Brother-Dead(m,i);
begin
        finish normal broadcasting using procedure "Coordinated" ;
        follow direction i until leaf node is reached;
        send the message from leaf node to $\oplus_i$(leaf node);
end;
```

We will show several examples that can tolerate n-1 processors failure and broadcast the message using our algorithm *Broadcast.* Those examples will start from the initiating node. When n-1 nodes are faulty under the initiating node, at least one of them is fault free. This will be the new initiating node and send the message to all the remaining nodes under the faulty node. Fig. 3.3 shows the optimal fault-tolerant broadcasting of Q_4 when 3 processors are faulty under the initiating node. In this example, only the youngest-son node is non-faulty. Since this broadcasting tree is the binomial tree, the youngest-son node is connected to one of nodes under the older-son nodes. Thus, it can send the information to one of non-faulty node under the faulty node.

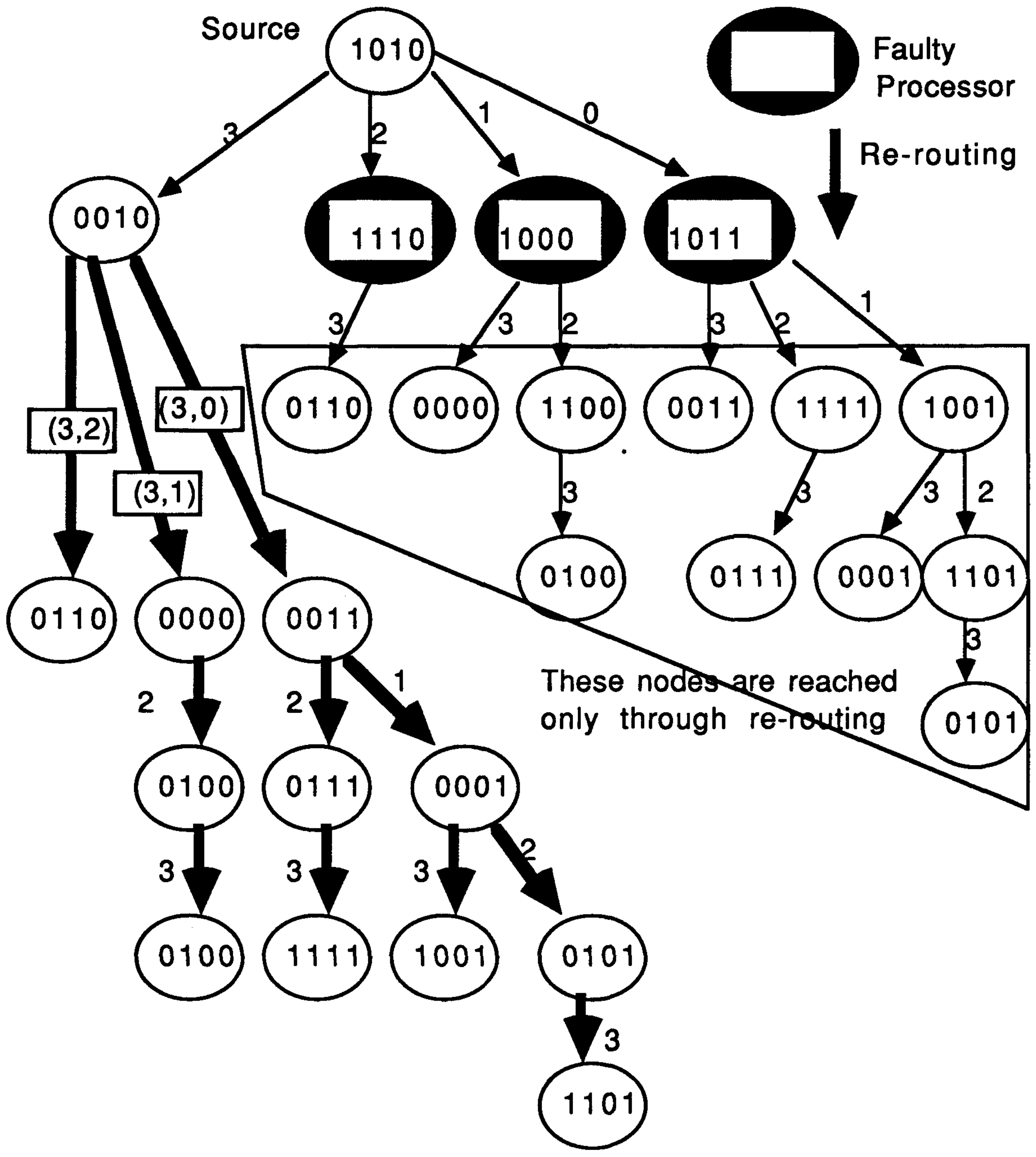


Fig 3.3. Broadcasting where three faulty processors under the source node in Q_4

Let us explain the above example briefly. The source node 1010 checks the $\oplus_0(1010)$ and finds the node faulty. Then it checks $\oplus_1(1010)$ and $\oplus_2(1010)$ to find if they are faulty or not. Finally $\oplus_3(1010)$ is non-faulty, so source node 1010 gives "$\oplus_0(1010)$ is faulty" to $\oplus_3(1010)$. Then $\oplus_3(1010)$ gives the message to node $(\oplus_0\oplus_3(1010)) = 0011$ and calls the

Fault-Tolerance-Source procedure. Thus the node 0011 becomes the new initiating node and sends the message to all the nodes in the subtree developed by original node $\oplus_0(1010)$. Here, node 0011 sends the message to $\oplus_1(0011)$. Next node 0011 and $\oplus_1(0011)$ sends the message to node 0111 and 0101 Finally node 0111, 0001 and 0101 will send the message to node 1111, 1001 and 1101 along d_3. Therefore, it takes 4 steps from node 0010 to node 1101. Similarily, when $\oplus_1(1010)$ and $\oplus_2(1010)$ are faulty, node $(\oplus_1\oplus_3(1010)) = 0000$ and $(\oplus_2\oplus_3(1010)) = 0110$ will be the new initiating node, respectively and follow the same routine as above. Here we can see all son-nodes {0011, 1111 and 1001} of 1000 will be the leaf nodes after re-broadcasting.

We show the another optimal broadcasting example where only the middle node is non-faulty in Fig. 3.4. When $\oplus_0(1010)$ is faulty, the same routine as in Fig. 3.3 is executed. So, the non-faulty node 1001 becomes the new initiating node and sends the message to the subtree under $\oplus_0(1010)$. Node 1001 will send the message to 0001 and 1101 using procedure *Fault-Tolerance-Source.* Node 1101 and 0001 sends the message to all the remaining nodes using procedure *Fault-Coordinated.* On the other hand, the non-faulty node 1000 is saved. When $\oplus_2(1010)$ and $\oplus_3(1010)$ are faulty, node (1010) gives the information to $\oplus_{(2,1)}(1010)$ and $\oplus_{(3,1)}(1010)$. When $\oplus_1(1010)$ receives the message of $\oplus_{(2,1)}(1010)$, it becomes the initiating node and finishes the normal broadcasting procedure, then calls the procedure *Younger-Brother-Dead,* where the direction *2* is followed and the leaf node is complemented by *direction 2*. Node 0110 receive the message through 1010 -> $\oplus_2(1010)$ -> $\oplus_3(\oplus_2(1010))$ in case of non-faulty broadcasting. However, in case of node $\oplus_2(1010) = 1110$ is faulty, node 0110 receives the message through 1010 -> $\oplus_1(1010)$ -> $\oplus_2(\oplus_1(1010))$ -> $\oplus_3(\oplus_2(\oplus_1(1010)))$ -> $\oplus_1(\oplus_3(\oplus_2(\oplus_1(1010))))$. Here, we know that $\oplus_3(\oplus_2(1010))$ is equal to $\oplus_1(\oplus_3(\oplus_2(\oplus_1(1010))))$.

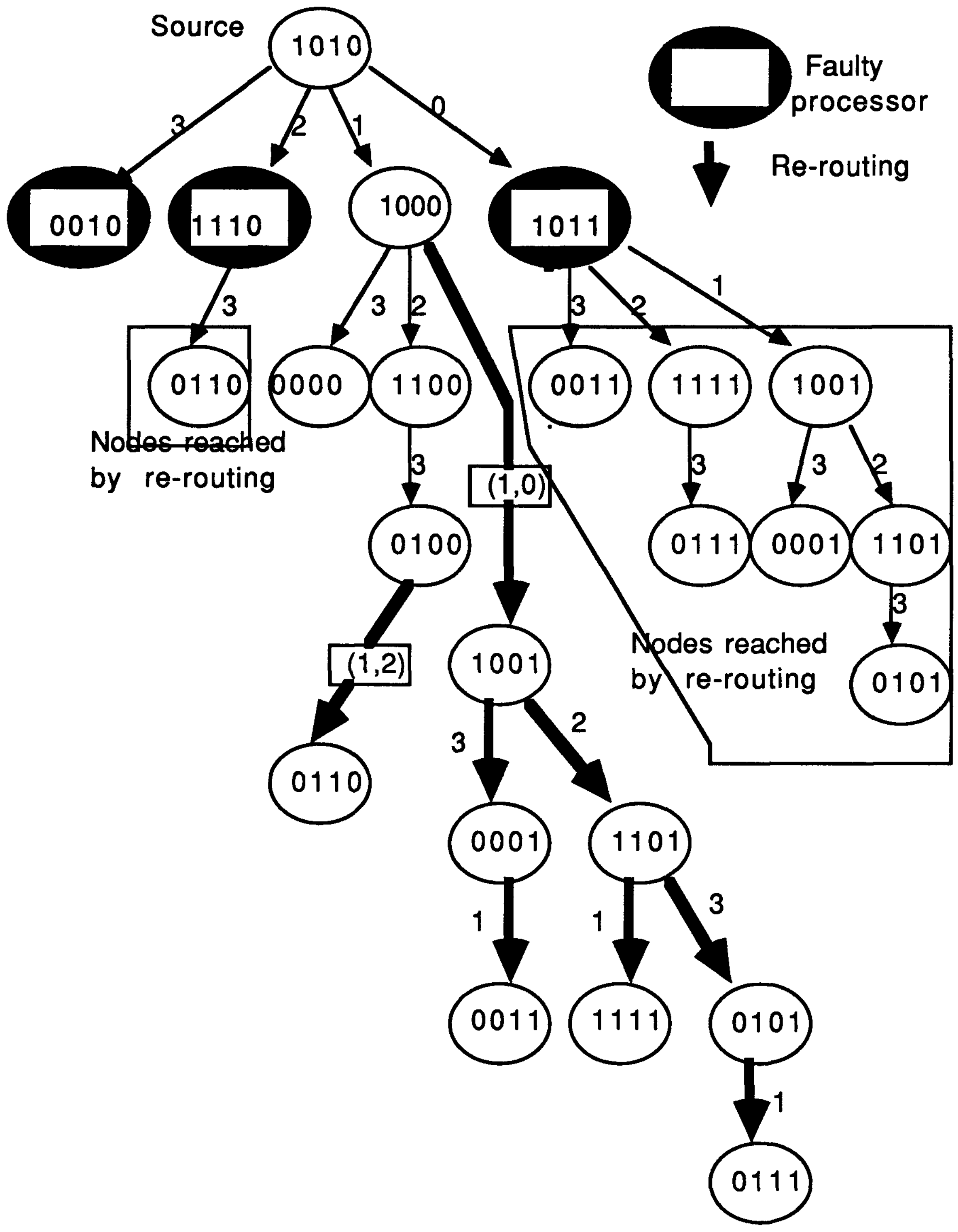


Fig. 3.4 Broadcasting when middle node is non-faulty in Q4

We show another example where 3 processors are faulty in different levels in Fig. 3.5. When $\oplus_1(1010)$ and $\oplus_2(1010)$ are faulty, the same procedure as in Fig. 3.4 is

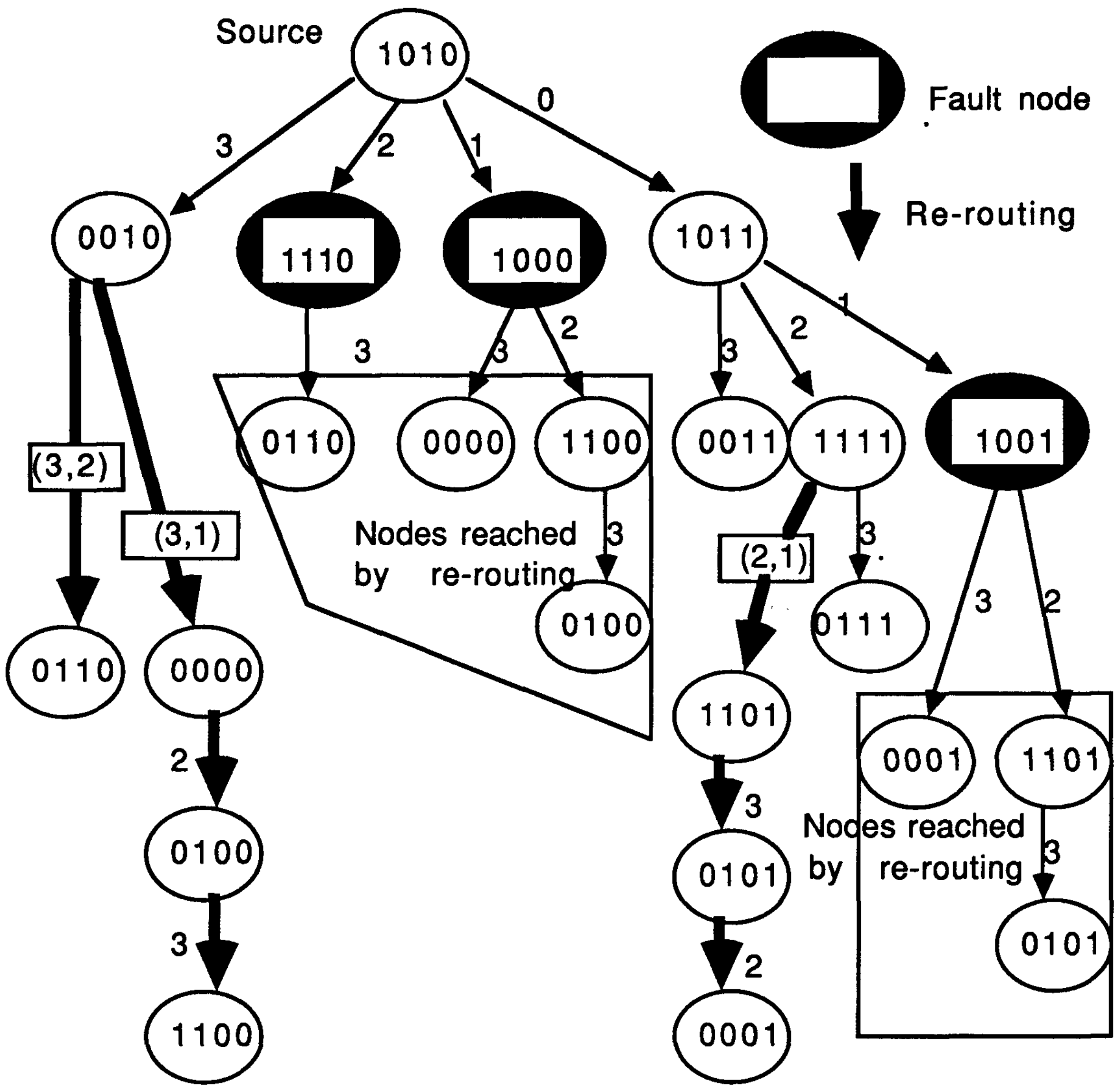


Fig. 3.5 Broadcasting in the presence of 3 faulty processors in different level in Q4

executed. However, when $\oplus_1(\oplus_0(1010))$ is faulty, the same procedure *coordinated* is called and $\oplus_2(\oplus_0(1010))$ sends the message to $\oplus_{(1,2)}(\oplus_1(1011))$ and calls the *fault-tolerance-source* procedure, so node 1101 becomes the initiating node and sends the message to all nodes 0101 and 0001. Node 1101 receive the message through 1011 -> $\oplus_1(1011)$ -> $\oplus_2(\oplus_1(1011))$ in case of non-faulty broadcasting and the message through 1011 -> $\oplus_2(1011)$ -> $\oplus_1(\oplus_2(1011))$ in case of faulty node.

We show an another example where 3 processors are faulty under the initiating processor in the subtree in Fig. 3.6. Here all son nodes are faulty and this example corresponds to procedure *all-sons-nodes*. In that case, all its younger brother nodes become the initiating nodes to send the message and finish the normal algorithm *broadcasting* and complements the leaf nodes by the direction *i* derived from the initiating node. Node 1101 receives the information through 1010 -> 1011 -> 1001 -> 1101 in normal broadcasting. When the node 1001 is faulty, the node 1101 receives the message through 1010 -> 1000 -> 1100 -> 1101. The node 0001 receives the message through 1010 -> 1000 -> 0000 -> 0001. That is, all the nodes broadcasted by $\oplus_i(s)$ can receive the message from all the nodes broadcasted by $\oplus_j(s)$ where $i+1 \leq j \leq n-2$. We shall prove this in next section.

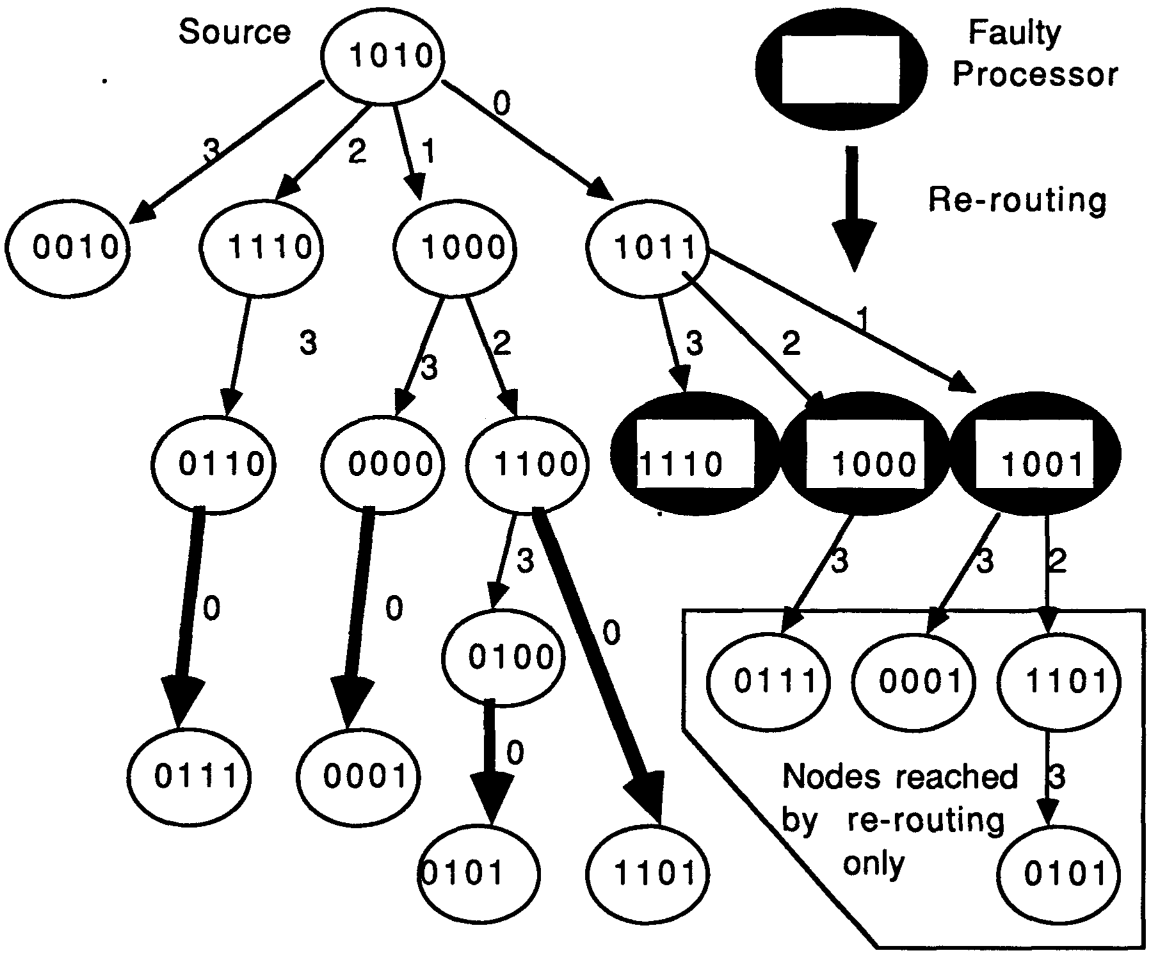


Fig. 3.6 Broadcasting when all-sons processors are faulty in Q4

3.5. Analysis of Broadcasting Algorithm

In this section we want to prove that our algorithm is optimal in case of n-1 nodes failure. We need the following Lemmas to prove. First we start with the initiating node to broadcast the message.

Lemma 3.4 : In broadcasting tree, if n-1 nodes are faulty under the initiating node s, then we can send the message to one of son-nodes under the faulty node.

proof: The initiating node s checks the $\oplus_i(s)$ where $0 \leq i \leq n-1$. Let us assume that only $\oplus_j(s)$ where $0 \leq j \neq i \leq n-1$ is non-faulty. Let one son node be $\oplus_j(\oplus_i(s))$ under $\oplus_i(s)$. When $\oplus_i(s)$ is faulty, s gives $\oplus_{(i,j)}(s)$ to $\oplus_j(s)$. Node $\oplus_j(s)$ gives the message to $\oplus_i(\oplus_j(s))$ in the algorithm Broadcast and $\oplus_i(\oplus_j(s))$ becomes the initiating node. If $\oplus_i(s)$ is non-faulty, the message is sent through s -> $\oplus_i(s)$ -> $\oplus_j(\oplus_i(s))$. However, when $\oplus_i(s)$ is faulty, the message is sent through s -> $\oplus_j(s)$ -> $\oplus_i(\oplus_j(s))$. Here, we find $\oplus_j(\oplus_i(s))$ is equal to $\oplus_i(\oplus_j(s))$. Therefore we can reach one of sons in the faulty node.

For example, if nodes 1010, 1000 and 1110 are faulty, the non-faulty node 0010 sends the message to nodes 0011, 0000 and 0110, respectively in Fig 3.3.

When the initiating node is faulty, one of the son-nodes will receive the message from one of the initiating node's brother node and will be the new initiating node to send the message. Then we have the following Lemma.

Lemma 3.5 : In broadcasting subtree, if one of son-nodes is non-faulty, then it can be the initiating node to send the message to all other processors.

proof : Let the initiating node s be faulty and $\oplus_i(s)$ be non-faulty. The $\oplus_i(s)$ receive the information, "s is faulty". Then $\oplus_i(s)$ calls the procedure *Fault-Tolerance-Source* and becomes the new initiating node s' and sends the message to its neighbors in the next level and calls the procedure *coordinated.* At that time, since $H(s', \oplus_i(s')) = 2$, s' can't broadcast directly to $\oplus_i(s')$. Thus, when the broadcasting tree is reorganized, all the brother-nodes of s' receives the message from their son-nodes. That is, $\oplus_i(s')$ must be leaf nodes when procedure *Fault-Tolerance-Source* is called.

For example, node 0011 becomes the initiating node and send the message to nodes 0111 and 0001 in Fig. 3.5. Those nodes send the message to all the subtrees through procedure *coordinated.* Here we can see that the nodes 1111 and 1001 which are brother nodes of node 0111 are leaf nodes

We can see the another example in Fig. 3.6, which shows all nodes 1110, 1000, 1011 are faulty under node 1011. In that case, node 1011 gives the information, "all his son-nodes are faulty", to all his brother-nodes. Then all his younger brother-nodes become the initiating nodes and broadcast the message to the non-faulty nodes under node 1011 by complementing *direction 0* of leaf nodes. We formally prove the result indicated by the above example.

Lemma 3.6 : In broadcasting subtree, if all son-nodes are faulty or initiating node and its all son nodes are faulty, then the initiating node's all younger brother-nodes can be initiating nodes to send the message to all the processors under the faulty node.

proof : Let s', m and m' be initiating node, $\oplus_p(s')$ and $\oplus_i(s')$ respectively. Let $\oplus_i(m)$ where $0 \le i \le n-1$ be faulty. If $\oplus_i(m)$ is non-faulty, the the son $\oplus_j(\oplus_i(m))$ of $\oplus_i(m)$ will receive the message through s' -> $\oplus_p(s')$ -> $\oplus_i(\oplus_p(s')$ -> $\oplus_j(\oplus_i(\oplus_p(s')))$. Since $\oplus_i(m)$ is faulty, m give the information, "$\oplus_i(m)$ is faulty" to s' and s' to m'. Thus, m' sends the message to $\oplus_j(m')$ and finally to $\oplus_p(\oplus_j(m'))$. So, if $\oplus_i(m)$ is faulty, the the son $\oplus_j(\oplus_i(m))$ of $\oplus_i(m)$ will receive the message through s' -> $\oplus_i(s')$ -> $\oplus_j(\oplus_i(s')$ -> $\oplus_p(\oplus_j(\oplus_i(s')))$. Here, $\oplus_j(\oplus_i(\oplus_p(s')))$ is equal to $\oplus_p(\oplus_j(\oplus_i(s')))$. Therefore, any son node under all faulty nodes can receive the message from the grand-father's younger brothers.

Lemma 3.7 : In a broadcasting tree, left(right) subtree of the initiating node are mapped directly to right(left) subtree by one step.

proof : According to Lemma 3.2, the broadcasting tree developed by the algorithm *Broadcast* is a binomial tree. All binomial trees having a given index are isomorphic in the sense that they have the same shape. Therefore, we can map left(right) subtree to right(left) subtree by one step.

All the above Lemmas 3.4, 3.5, 3.6 and 3.7 lead to the following theorem.

Theorem 3.2: Our broadcasting algorithm tolerates the failure of n-1 processors.

Proof: From Lemma 3.4, the initiating node to send the message can tolerate n-1 processors failure. This Lemma corresponds to algorithm *Broadcast* and *All-Brothers-Dead*. From Lemma 3.5, if one of the son-nodes is non-faulty, it becomes the new initiating node and sends the message to all the non-faulty nodes in the sub-broadcasting tree. This Lemma corresponds to procedure *Coordinated*. From Lemma 3.6, if all sons are faulty, all of initiating node's younger brothers can be the initiating nodes and each sends the message to all the non-faulty nodes in the sub-broadcasting tree. This corresponds to procedure *All-Sons-Dead*. Finally, these Lemmas can be applied to any level of subtrees. Therefore, our algorithm can tolerate the loss of n-1 processors.

In the rest of this section, we will evaluate the performance of algorithm *Broadcast* in terms of the number of steps required to complete the message delivery. It is shown in Theorem 3.3 below that the fault-tolerant delivery can be completed in n+1 steps. We are not considering the steps to send the information, the $\oplus_{(i,j)}(s)$.

Theorem 3.3 : Our optimal fault-tolerant broadcasting algorithm needs n+1 steps.

Proof : When the node initiating the broadcast (say s) sends the message, it checks $\oplus_i(s)$ and gives the $\oplus_j(s)$ to $\oplus_{(i,j)}(s)$. Then, $\oplus_j(s)$ gives the message to one of the sons of $\oplus_i(s)$, which will be the new initiating node and sends the message to all non-faulty nodes in the same step as the normal broadcasting tree. That is, we need one more step from the new initiating node to its brother node when the subtree under the faulty node is reorganized. Therefore, we need n+1 steps in this optimal fault-tolerant broadcasting algorithm.

REFERENCES

1. F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles, A Versatile Network for Parallel Computation," Communication of ACM, pp. 30-39, May 1981.

2. J. D. Ullman, Computational Aspects of VLSI, Computer Science Press, 1984

3. J. E. Jang,"Optimal Fault Tolerant Broadcasting Algorithm in an Cube-Connected Cycles Network", Proc. Int'l Conference on Databases, Parallel Architectures and its applications (PARBASE-1990), Mar, 1990. pp. 206-215.
(To appear as a chapter in a book published by IEEE)

4. H. S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Computers, pp. 153-161, Feb. 1971.

5. H. Sullivan and T. R. Baskow, "A large scale homogeneous, fully distributed parallel machine" Proc. Fourth Symp. Comp. Architecture, Mar. 1977, pp. 105-117.

6. J. E. Jang, S.W. Choi and W.K. Cho, "A New Approach to Processor Allocation and Task Migration in an N-cube Multiprocessor", Proc, Supercomputing 89', Nov. 1989. pp.314-325.

7. M. Chen and K.G. Shin, "Processor Allocation in an N-cube Multiprocessor Using Gray Codes", IEEE Trans. Computer, Dec. 1987 pp. 1396-1407.

8. M. Chen and K.G. Shin, "Task Migration in Hypercube Multiprocessor", Proc. 16th Annual Int'l Symp. on Computer Architecture. Jun 1989,

9. M. Chen and K.G. Shin, " Adaptive Fault-Tolerant Routing in Hypercube Multicomputers" To appear in IEEE Trans. on Computers, 1989

10. B. Becker and H.U. Simon, "How robust is the n-cube?", in Proc. 27th Ann. Symp. Foundations of Comp. Sci. Oct. 1986 pp 283-291.

11. H. P. Kattesff, "Incomplete hypercubes", IEEE Trans. Computer, May 1988, pp 604-608.

12. A. Al-Dhelaan and B. Bose, "A New strategy for Processor Allocation in an N-cube Multiprocessor", Phoenix Conference on Computer and Communication, Mar 1989. pp. 114-118.

13. A. Al-Dhelaan and B. Bose, "Efficient Fault Tolerant Broadcasting Algorithm for the Hypercube", Proc.The fourth Conf. on Hypercube Concurrent Comp. and Applications, Monterey, Mar 1989, pp. 123-128.

14. P. Ramanathan and K.G. Shin, "Reliable Broadcasting in Hypercube Multicomputers", IEEE Trans. on Comp. Dec 1988, pp 1654-1657.

15. Y. Saad and M.H. Schultz, "Topological Properties of Hypercubes", IEEE Trans. on Computers, Jul 1988, pp 867-872

16. L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," ACM Trans. Programming language System, pp. 382-401, Jul. 1982.

17. T. K. Srikanth and S. Toueg, "Optimal clock synchronization," J. ACM pp.626-645, Jul.1987.

18. N. Deo, Graph Theory with applications to Engineering and Computer Science, Prentice-Hall, 1974.

19. M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms", SIAM J. Comput, Vol. 7, Aug. 1978, pp 298-319.

20. J. E. Jang and W.K. Cho, "Maximality of Subcube Recognition and Fault-Tolerance in an N-cube multiprocessors", Proc. 4th SIAM Conference on Parallel Processing, Dec, 1989.

21. J. E. Jang, "An Optimal Fault Tolerant Broadcasting Algorithm foe a Hypercube Multiprocessor", Proc.1990 ACM Computer Science Conference, Feb, 1990. pp. 96-102.

Chapter 4

Reliable Broadcasting Algorithm for a Cube-Connected Cycles Network

4.1 Introduction

Rapid advancing technology has made it possible for a large number of processing elements(PEs) to be interconnected together on a single chip as a viable means of implementing high performance integrated systems. A number of parallel architectures have been proposed, such as hypercubes, meshes, trees and cube-connected-cycles(CCC)[1-4]. Several of these interconnections are well suited to VLSI implementation due to their structural regularity.

Cube-connected-cycles is a parallel network architecture proposed by Vuillemin[1]. The CCC can efficiently solve a large class of problems that include Fourier transform[5], sorting[6], permutations, etc,. Unfortunately, the cube[5-6] is not readily usable for VLSI design since each processor in a k-dimensional cube is connected to k other processors[7-8,15]. The operation of the cube-connected-cycles network is based on the combination of piplining and parallelism, which leads to the following results[12]:

1. The number of connections per processor is reduced to three.

2. Processing time is not significantly increased with respect to that achievable on the cube-connected network.

3. The overall structure complies with the basic requirements of the VLSI technology: modularity, ease of layout, simplicity of communication among processors, simplicity in timing and control of the entire system.

Broadcasting is an important means of communication among processors by which a processor can pass data or control to all other processors in the network. This operation is extremely important for diagnosis of the network, distributed agreement [9] or clock synchronization[10].

Distributed agreement and clock synchronization can be achieved only if there is no faulty node to deliver the message in the system. This, however, is not easy to achieve in the presence of faulty node/link because the faulty nodes can either omit, corrupt, reroute, or alter information passing through them.

There are two possible approaches to overcome this problem. In the first approach, each node keeps limited information about the faulty nodes in the system. Fault-tolerant routing/broadcasting is achieved by going around the faulty nodes[11-12]. This approach can be used only if it is possible to identify the faulty processors "on-line". Since the overhead of identifying the faulty processors and passing the fault information to the other nodes could be quite severe, this approach is not suitable for many real-time applications. In the second approach, fault tolerance is achieved by sending multiple copies of the message through disjoint paths[13-14]. The nodes that receive the message identify the original message from the multiple copies by using some scheme that is appropriate for the fault model, e.g., majority voting. The second approach has the advantage of not having to identify the faulty processors during the normal operation of the system. This advantage is especially important in many critical real-time applications.

In this chapter, we present both approaches. The first broadcasting algorithm delivers delivers multiple copies of the message to all nodes in the CCC through 3 disjoint paths. The basic idea of our algorithm is as follows. The node that wants to broadcast a message sends the message to all its neighbors in the same ring. The neighbors in the same ring and the node initiating the message in turn broadcast the message using a simple yet efficient algorithm. The algorithm executed by the neighbors is coordinated such that the copies of the message received by a node have traveled through disjoint paths. The good feature of the proposed algorithm is that the delivery of the multiple copies is transparent to the processes receiving the message and does not require the processes to know the identity of the faulty processors. Depending on the fault modes used, the algorithm can tolerate either s-1 or $\lfloor s/2 \rfloor$ or $\lfloor s/3 \rfloor$ node/link faults. The algorithm completes in $\lfloor s/2 \rfloor$ + (2s-1) + $\lfloor s/2 \rfloor$ steps and 4s steps if each node can use all and at most one of its outgoing links at a time respectively.

The second broadcasting algorithm delivers a copy of message to all nodes nonredundantly. The basic idea of our algorithm is as follows. The node that wants to broadcast a message checks if its neighbor node is faulty or not. If the neighbor node is faulty, the initiating node gives this information to one of non-faulty son-nodes of the faulty node. This non-faulty node broadcasts the message to the non-faulty nodes in the subtree under the faulty-node. This algorithm tolerates 2 processors failure if the neighbor nodes are faulty and s-1 rings or s-1 processors faults, otherwise.

This chapter is organized as follows. Section 4.2 describes the preliminaries, problem statement and the notation used in this paper. Section 4.3 describes the proposed algorithm which is composed of delivery mechanism and reception mechanism. We evaluate the performance of the proposed algorithm in terms of steps required for completion for different communication capabilities at each node in Section 4.4. An optimal fault tolerant broadcasting algorithm is described in Section 4.5. Section 4.6 evaluates the performance of algorithm in Section 4.5 in terms of steps to broadcast in case of s-1 processors failure and proves that this algorithm is optimal.

4.2 Preliminaries and Problem Statement

The cube-connected-cycles as proposed by Preparata and Vuillemin [1] is a network of identical processing elements (PEs) each having three interconnection ports. Each link connecting two PEs can be used for the bidirectional transmission of data. The entire system can be synchronized either locally or globally. A general version of the CCC had been proposed in which some of the PEs have two ports while the others have three. In order to describe the interconnections for the generalized CCC network, we assume that the number of PEs is $n=h*2^s$ for $h \geq s$. The PEs are grouped into 2^s cycles, each cycle consisting of h PEs. We define *s* be the dimension of the CCC. Each PE has an address that can be expressed as a pair (l,p) of integers where l refers to the address of the cycle to which a PE belongs, and p refers to the address of the PE within the cycle. Here $l = 0,1,2 \ldots 2^{s-1}$ and $p = 0, \ldots, h-1$. The PEs with $p = 0, \ldots, h-1$ have three interconnection ports: F,B,L (for forward, backward and lateral) whereas the PEs with $p = s, \ldots, h-1$ have only F and B ports. The generalized CCC connection is given formally as follows:

$F(l,p)$ is connected to $B(1,(p+1) \bmod h)$

$B(l,p)$ is connected to $F(1,(p-1) \bmod h)$

$L(l,p)$ is connected to $L(1+ e2^p,p)$

where $e = 1-2bit_p(1)$: bit_p means the pth bit of 1.

To provide an intuitive feeling for the topology, a CCC with h=3, s=3 and h=4, s=3 is illustrated in Figure 4.1 and 4.2 respectively. In this paper we treat that s is the same as the h.

Each ring in a CCC can be uniquely represented by an n-bit address in such a way that the address of the adjacent ring nodes differ in exactly one bit. For convenience, we will number the bits in an address of a ring in the CCC from right to left as 0 to s-1. If two adjacent ring differ in their *i*th bit, then they will be said to be in direction *i* with respect to each other. For example, the ring with address U=111 will be said to be in direction 1 with respect to node w=101 and vice versa, that is, $\oplus_1(U) = W$, where $\oplus$ means exclusive-OR operation. It is clear from this definition there are *s* distinct ring directions in a CCC, denoted by $d_0, d_1, \ldots d_{s-1}$. The node in direction i with respect to the node u will be denoted by $\oplus_i(u)$.

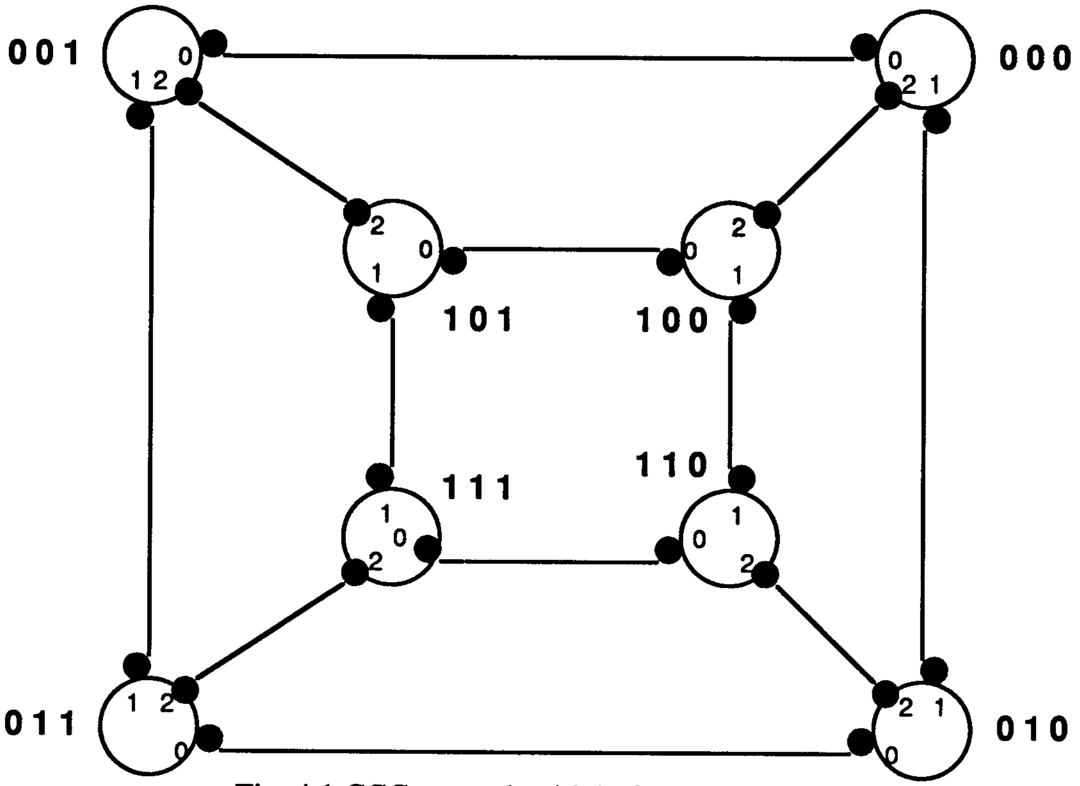


Fig. 4.1 CCC network with h=3 and s=3 (n=24)

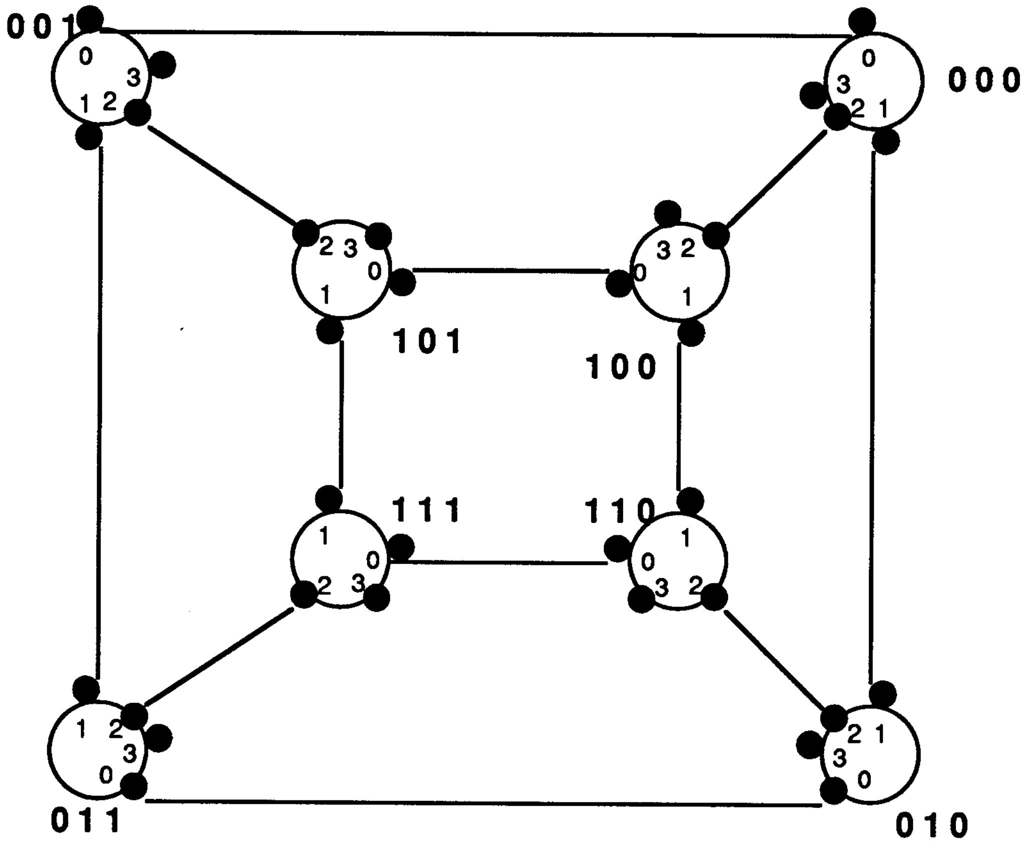


Fig. 4.2 CCC network with h=4 and s=3 (n=32)

We can easily state the problem to solve in this paper as follows.

Given 1) an s-dimensional CCC subject to node/link failures, 2)there are a maximum of t node/link faults in the CCC, and 3) the identity of the faulty node/link is not known, the problem is to develop a broadcasting algorithm that satisfies the following condition.

Condition: If the node initiating the broadcasting is non-faulty, then all the non-faulty nodes in the CCC must agree on the message broadcasting by the initiating node.

As shown below in Example 1, the *condition* is not always easy to satisfy in the presence of faulty node/link. We will first present a broadcasting algorithm that delivers multiple copies of the message through disjoint paths and then determine the maximum number of faults t that the algorithm can tolerate for different fault models. The proposed algorithm is suitable for applications that cannot tolerate the time overhead of identifying the faulty processors, which could in general be quite long.

Example 1: Consider the CCC shown in Fig 4.1. Suppose node (0,0) initiates a broadcasting using the following algorithm.

Step 1: Node (0,0) sends the message to node (1,0).

Step 2: Nodes (1,0) and (0,0) simultaneously send the message to nodes {(1,1), (1,2)} and (0,1) respectively.

Step 3: Node (1,1) and (0,1) simultaneously send the message to nodes (3,1) and (2,1) respectively.

Step 4: Nodes (3,1), (2,1) and (0,0) simultaneously send the message to nodes {(3,0), (3,2)}, {(2,0), (2,2)} and (0,2) respectively.

Step 5: Nodes (0,2), (1,2), (2,2) and (3,2) simultaneously send the message to node (4,2), (5,2), (6,2) and (7,2) respectively.

Step 6: Nodes (4,2) and (5,2) and (6,2) and (7,2) simultaneously send the message to nodes {(4,0), (4,1)}, {(5,0), (5,1)}, {(6,0), (6,1)} and {(7,0), (7,1)} respectively.

Now suppose node (1,0) is faulty. Then, at the end of the algorithm, nodes (1,1), (1,2), (3,1), (3,0), (3,1), (5,0), (5,1), (5,2), (7,0), (7,1), (7,2) may have received either an incorrect message or no message at all, therefore the *condition* is violated.

4.3 Proposed Broadcasting Algorithm

There are two kinds of mechanisms in any broadcasting algorithm: the *delivery* and the *reception* of messages. The "message delivery" is compromised of algorithms used by the nodes to deliver multiple copies of the broadcasting message to all nodes. In the proposed algorithm, s copies of the message are correctly delivered to all nodes if there are no faults in the system, where s is the dimension of the CCC. However, in the presence of faults, some of the s copies may either get lost or corrupted.

The "message reception" is comprised of algorithms used by the nodes to interpret and identify the correct information from the multiple copies. The identification of the correct information from the multiple copies is strongly dependent on the fault model used. Section 3-A and 3-B describe the delivery and reception mechanism.

A. The Delivery Mechanism

The delivery mechanism proceeds in two phases. In the first phase, the node initiating the broadcasting sends the message to all its neighbors in the same ring and to its neighbor in the adjacent ring applying $\oplus_i$ to the source node. In the second phase, the neighbors in the same ring and the node initiating the message use "coordinated" procedure to broadcast the message to all the nodes. The sequence of directions used by these neighbors in their "coordinated" phase is coordinated in order to ensure that each node gets the broadcasting message through *s* disjoint paths. A formal description of the algorithm is given below.

```
Algorithm broadcasting(A(l,p))      /* A : initiating node */
begin
      if (each node can use all its outgoing links at a time) then
        for $1 \le i \le \lceil h/2 \rceil$ do
          send message from A(l,p) to A(l,p + i mod h)  /* forward in the ring */
        for $1 \le i \le \lfloor h/2 \rfloor$ do
          send message from A(l,p) to A(l,p - i mod h)  /* backward in the ring */
      else (each node can use at most one of its outgoing links at a time)
```

```
        for $1 \le i \le h-1$ do begin
          send message from $A(l,p)$ to $A(l,p + i \bmod h)$ /* forward in the ring */
      for $0 \le i \le s-1$ do begin
              send message from $A(l,i)$ to $A(\oplus_i l, i)$;
              Coordinated $(A(\oplus_i l, i), i+1)$
      end;
end;
procedure Coordinated $(m(p,q), k)$ /* m : initiating node, $d_k$ : starting direction */
begin
      for $0 \le i \le s-1$ do begin
        if (each node can use all its outgoing links at a time)
          for $1 \le i \le \lceil h/2 \rceil$ do begin
            send message from $A(l,p)$ to $A(l,p + i \bmod h)$ /* forward */
          for $1 \le i \le \lfloor h/2 \rfloor$ do begin
            send message from $A(l,p)$ to $A(l,p - i \bmod h)$ /* backward */
        else (each node can use at most one of its outgoing links at a time)
          for $1 \le i \le h-1$ do begin
            send message from $A(l,p)$ to $A(l,p + i \bmod h)$ /* forward */
        R = {m};     /* R: set of rings that have received the message */
        for each ring $\in$ R
                begin
                        Adjacent_Ring = $\{\oplus_{k+i \bmod s}(j)\}$
                        Calculate the position for the processor to connect two
                                rings using the definition of interconnection of CCC
                        send message from j.position to Adjacent_Ring.position;
                        R = R $\cup$ Adjacent_Ring
                end;
      end;
end;
```

In the absence of faults, each node gets s identical copies of the message in the above algorithm. This is because the nodes get one message in each of the "coordinated" sequences initiated by the *s* neighbors of A. It is shown later in Theorem 4.1 that the paths through which a node receives the s copies of the message are disjoint. Given below is an example to illustrate the basic idea of the algorithm.

Fig. 4.3 shows the broadcasting example of the multiple copies by executing the above algorithm for the CCC. Node (000,1) is the initiating node.

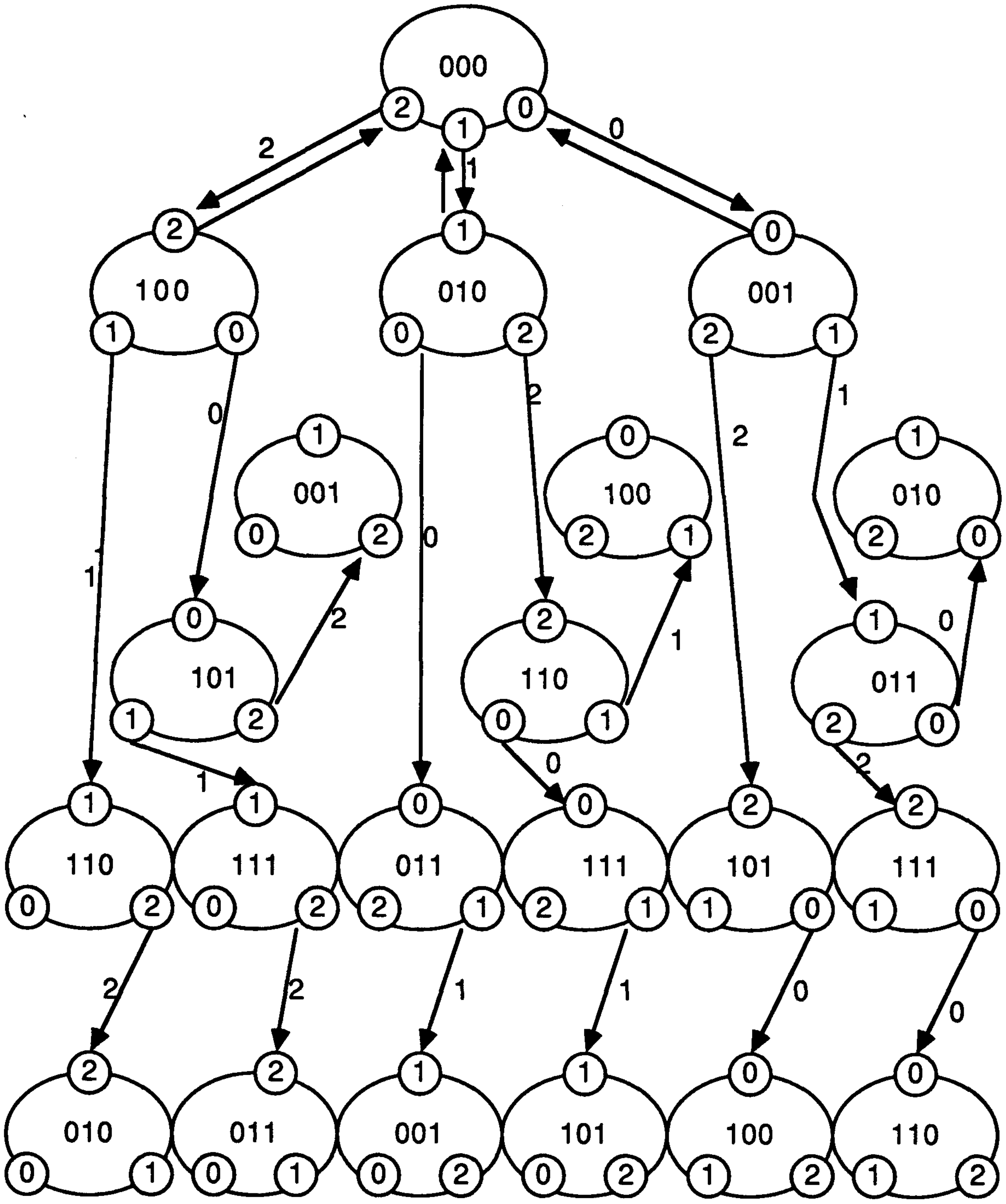


Fig. 4.3 Broadcasting multiple copies in a CCC

Example 2: Consider the CCC in Fig 4.1. Let node (0,0) initiate the broadcasting. In the first phase, node (0,0) sends its message to nodes (0,1), (0,2) in the same ring and to nodes (1,0), (2,1), (4,2) in the adjacent rings. In the second phase, nodes (1,0), (2,1), (4,2) use coordinated recursive doubling to broadcasting the message they received from node (0,0) to all other nodes in the CCC. In its "Coordinated" procedure, node (1,0) first sends message to nodes (1,1), (1,2) in the same ring, then find the adjacent ring along d_1 be 3 and the nodes (1,2) and (3,1) be connected, so node (1,1) sends the message to node (3,1), which sends message to nodes (3,0), (3,2). Then nodes (1,2) and (3,2) send the message along d_2 to the adjacent ring nodes (5,2) and (7,2) respectively, which send the message nodes (5,0), (5,1) and nodes (7,0), (7,1) respectively, and finally nodes (1,0), (3,0), (5,1), (7,1) send the message along d_0 to the adjacent ring nodes (0,0), (2,0), (6,0), (4,0), resulting in {(0,1), (0,2)}, {(2,1), (2,2)}, {(6,1), (6,2)}, {(4,1), (4,2)} respectively. Similarly, node (2,1) and node (4,2) uses $d_2 \rightarrow d_0 \rightarrow d_1$ and $d_0 \rightarrow d_1 \rightarrow d_2$ as the sequence of the directions so as to find the adjacent ring in its "Coordinated" procedure.

We next prove some Lemmas that are needed to prove that all the nodes in the system receive the same multiple copies of message.

Lemma 4.1 : All the nodes in the algorithm *Broadcasting* are $s*h*2^s$.

Proof : Let $A(l,p)$ be the node initiating the broadcasting and R be the set of rings. A(l,p) sends the message to its neighbors in the following order: $A(\oplus_0 l, p)$, $A(\oplus_1 l, p)$, . . . $A(\oplus_{s-1} l, p)$. The neighbors use "Coordinated" procedure, where they send the message to the all nodes in the same ring and "$0 \le i \le s-1$" and "$R = R \cup$ Adjacent_Ring" make the total number of nodes be $1 + \sum_{i=0}^{s-1} 2^i = 2^s$. Therefore, the total number of nodes are $s*h*2^s$.

Table 4.1 shows the disjoint paths through which nodes (1,0) - (7,2) receive their messages where the initiating node is (1,0).

	Path via		
Node	Node (1,0)	Node (2,1)	Node (4,2)
(1,0)	(0,0)	(2,1)(2,0)(3,0)(3,1)(1,1)	(0,2)(4,2)(4,0)(5,0)(5,2)(1,2)
(1,1)	(0,0)(1,0)	(2,1)(2,0)(3,0)(3,1)	(0,2)(4,2)(4,0)(5,0)(5,2)(1,2)
(1,2)	(0,0)	(2,1)(2,0)(3,0)(1,1)	(0,2)(4,2)(4,0)(5,0)(5,2)
(2,0)	(0,0)(1,0)(1,1)(3,1)(3,0)	(2,1)	(0,2)(4,2)(4,1)(6,1)(6,2)(2,2)
(2,1)	(0,0)(1,0)(1,1)(3,1)(3,0)(2,0)	(0,1)	(0,2)(4,2)(4,1)(6,1)(6,2)(2,0)
(2,2)	(0,0)(1,0)(1,1)(3,1)(3,0)(2,0)	(2,1)	(0,2)(4,2)(4,1)(6,1)(6,2)
(3,0)	(0,0)(1,0)(1,1)(3,1)	(2,1)(2,0)	(0,2)(4,2)(4,0)(5,0)(5,1) (7,1)(7,2)(3,2)
(3,1)	(0,0)(1,0)(1,1)	(2,1)(2,0)(3,0)	(0,2)(4,2)(4,0)(5,0)(5,1) (7,1)(7,2)(3,2)
(3,2)	(0,0)(1,0)(1,1)(3,1)	(2,1)(2,0)(3,0)	(0,2)(4,2)(4,0)(5,0)(5,1) (7,1)(7,2)
(4,0)	(0,0)(1,0)(1,2)(5,2)(5,0)	(2,1)(2,2)(6,2)(6,1)(4,1)	(0,2)(4,2)
(4,1)	(0,0)(1,0)(1,2)(5,2)(5,0)(4,0)	(2,1)(2,2)(6,2)(6,1)	(0,2)(4,2)
(4,2)	(0,0)(1,0)(1,2)(5,2)(5,0)(4,0)	(2,1)(2,2)(6,2)(6,1)(4,1)	(0,2)
(5,0)	(0,0)(1,0)(1,2)(5,2)	(2,1)(2,2)(6,2)(6,1)(7,1)(7,0)	(0,2)(4,2)(4,0)
(5,1)	(0,0)(1,0)(1,2)(5,2)	(2,1)(2,2)(6,2)(6,0)(7,0)(7,1)	(0,2)(4,2)(4,0)(5,0)
(5,2)	(0,0)(1,0)(1,2)	(2,1)(2,2)(6,2)(6,0)(6,0)(7,0)(7,1)(5,1)	(0,2)(4,2)(4,0)(5,0)
(6,0)	(0,0)(1,0)(1,1)(3,1)(3,2)(7,2)(7,0)(6,0)	(2,1)(2,2)(6,2)	(0,2)(4,2)(4,1)
(6,1)	(0,0)(1,0)(1,1)(3,1)(3,2)(7,2)(7,0)(6,0)	(2,1)(2,2)(6,2)	(0,2)(4,2)(4,1)
(6,2)	(0,0)(1,0)(1,1)(3,1)(3,2)(7,2)(7,0)(6,0)	(2,1)(2,2)	(0,2)(4,2)(4,1)(6,1)
(7,0)	(0,0)(1,0)(1,1)(3,1)(3,2)(7,2)	(2,1)(2,2)(6,2)(6,0)	(0,2)(4,2)(4,0)(5,0)(5,1)(7,1)
(7,1)	(0,0)(1,0)(1,1)(3,1)(3,2)(7,2)	(2,1)(2,2)(6,2)(6,0)(7,0)	(0,2)(4,2)(4,0)(5,0)(5,1)
(7,2)	(0,0)(1,0)(1,1)(3,1)(3,2)	(2,1)(2,2)(6,2)(6,0)(7,0)	(0,2)(4,2)(4,0)(5,0)(5,1)(7,1)

Table 4.1 Paths through which the nodes receive the broadcasting

In the table, columns 2, 3 and 4 indicate the path through which the node in column 1 receives its message in the procedure "Coordinated" initiated by the nodes (1,0), (2,1), (4,2).

Lemma 4.2: The nodes in the algorithm Broadcasting are distinct.

Proof : In "Coordinated" procedure, m and d_k is different whenever they are invoked by the algorithm *Broadcasting*. As the value i in $0 \le i \le s-1$ changes, Adjacent_Ring = $\{\oplus_{k+i \bmod s}(j)\}$ is also changed. Therefore, all nodes are distinct in the **for-loop**.

These Lemmas lead to the following important results.

Theorem 4.1 : Algorithm Broadcasting sends the same multiple copies of message to all modes in the system.

Proof: The total number of nodes in CCC are $h*2^s$. According to the Lemma 4.1 and Lemma 4.2, each node receives the *s* copies of the message.

It is important to note that if the neighbors of the initiating node do not coordinate the sequence of directions in "Coordinated" procedure, then some nodes will not receive their copies of the message through disjoint paths. If two or more copies are reviewed through non-disjoint paths, then a single faulty node could corrupt more than one copy of the message. As we will see later in Section 3-B, this could cause severe problems in identifying the original from the multiple copies.

The following example illustrates the effect on the paths of the multiple copies if the neighbors of the initiating node in the same ring do not adhere to the sequence of directions in order to find the adjacent ring specified in algorithm *Broadcasting*.

Example 3: Suppose that node (2,1) in Fig 4.1 does not adhere to the sequence of directions specified in algorithm *Broadcasting*. Let the sequence of directions used by node (2,1) be $d_0 \rightarrow d_2 \rightarrow d_1$. Then it is easy to verify that node (7,2) will receive one message from node (0,0) through nodes (0,1) -> (2,1) -> (2,0) -> (3,0) -> (3,2) ->(7,2) and the other message from node (0,0) through nodes (1,0) -> (1,1) -> (3,1) -> (3,2) -> (7,2), i.e., node (7,2) receives messages from node (0,0) through paths that are not disjoint.

In ths rest part of this section, we formally prove the results indicated by Example 2 and 3. Let R refer to the "coordinated" phase of $\oplus_i(A)$ in the above algorithm. Let $P_i(q)$

denote the path through which node q receives a copy of the message in R_i. Define $P_i(q) \cap P_j(q)$ to mean the set of nodes common to both $P_i(q) \cap P_j(q)$.

Theorem 4.2: In algorithm *Broadcasting*, all paths are disjoint, that is, $P_i(q) \cap P_j(q) = 0$, for all nodes q and $0 \leq i,j \leq s - 1$, $i \neq j$.

Proof : Suppose not. Then, there are exist q, j and k such that $P_i(q) \cap P_j(q) \neq 0$. Without loss of generality, one can assume that j=0. Let $r \in P_i(q) \cap P_0(q)$. It follows from 1) all directions in R_0 are distinct, and 2) d_0 is the final direction in R_0. So, H(A,q) = H(A,r) + H(r,q) where $r \in P_0(q)$. Using similar reasoning, H(A,q) = H(A,r) + H(r,q) where $r \in P_i(q)$. Even though the Hamming distance is the same, the direction *i* to make the Hamming distance same is different. So, r can not be set of nodes common to both $P_i(q) \cap P_j(q)$. Contradiction.

For example, Hamming distance of the ring node between node (0,1) and node (7,1) is 3. But, node (7,1) receive the message through 000 -> 001 -> 011 -> 111 in R_0, 000 -> 010 -> 110 -> 111 in R_1, 000 -> 100 -> 101 -> 111 in R_2.

B. The Reception Mechanism

As mentioned earlier, the reception mechanism strongly depends on the fault model used. In this section, we will consider the reception mechanism for different kinds of fault models. Since the identity of the faulty component is not known, it is impossible to distinguish between node and link failures. Thus, we will treat them to be equivalent.

Let us consider the simple omission faults, i.e., a faulty node either sends the message correctly or does not send any message at all. A faulty processor or a faulty link at that node could have resulted in this case. Since copies that arrive at the receiving node are not corrupted, the original message can be identified from any node of the received copies. So, the broadcasting is guaranteed to satisfy the *Condition* in Section 3.2, if there are fewer than s faults. Hence, the maximum number of faults that algorithm *Broadcasting* can tolerate is s-1 in the case of simple omission fault model.

If the faulty nodes can corrupt the messages passing through them, then the reception mechanism is more complicated. Let us consider the case when the faulty nodes do not corrupt the message maliciously, i.e., non-Byzantine faults[9]. In this case, the original message can be identified from the received copies by using simple majority voting, i.e., the information in a received message is considered correct if the receiving node has $\lceil s/2 \rceil$ copies of that information. However, since all the copies do not arrive at the same time, majority voting is not as simple as in the tightly synchronous situation. The receiving processor can assume the broadcasting is complete and perform the necessary voting only if they can establish a quorum. There are two alternative ways of establishing a quorum [13]. A simple way to establish a quorum is to wait until $\lceil 2s/3 \rceil$ copies of the broadcasting message arrive before majority voting. Since copies passing through faulty nodes may not arrive at all, $\lceil 2s/3 \rceil$ or more copies will arrive only if there are fewer than or equal to Îs/3° faults in the system. Therefore, with this approach for establishing quorum, algorithm *Broadcasting* can tolerate a maximum of $\lfloor s/3 \rfloor$ faults.

An alternative approach[13] for establishing a quorum is to maintain a count of identical copies received. This approach can tolerate more faults than the first approach but has additional overhead for determining the establishment of a quorum. A quorum is established when there are at least $\lceil s/2 \rceil$ identical messages. As a result, with approach, algorithm *Broadcasting* can tolerate a maximum of $\lfloor s/2 \rfloor$ faults.

Finally, the worst situation is when the faulty processors can exhibit Byzantine behaviour, i.e., they can behave in any arbitrary manner including omitting, corrupting, rerouting, and even lying[9]. This case is not similar to the non-Byzantine case if we are interested in satisfying only condition C1. However, if we are interested in using the proposed algorithm for broadcasting in distributed agreement or clock synchronization algorithms, then we can tolerate a maximum of $\lfloor s/3 \rfloor$ faults [9-10].

As shown above, the algorithm *Broadcasting* can tolerate the number of faults and the receiving mechanism depends on the fault model used.

4.4 Performance of Algorithm *Broadcasting*

In this section, we will evaluate the performance of algorithm *Broadcasting* in terms of the number of steps required to complete the delivery mechanism. It is shown in Theorem 2 and 3 below that the delivery of the multiple copies can be completed in either $\lfloor s/2 \rfloor + (2s-1) + \lfloor s/2 \rfloor$ or 4s steps depending on the communication capability of each node.

If each node can send a message through at most one outgoing link at a time, then the algorithm requires a total of $\{(h-1)(s+2)+(s+1)\}$ steps. The node initiating the broadcasting sends the message to all its neighbors in the same ring in the first h-1 steps. The neighbors start their coordinated recursive doubling immediately after receiving the message. The last ring to receive the message uses the last s+1 steps for its recursive doubling. To prove that is is feasible to complete the algorithm in $\{(h-1)(s+2)+(s+1)\}$ steps, we have to prove that in every step each node has at most one message to send out. Each node will have at most one message to send only if the node initiating the broadcasting sends the message to its neighbors in the same ring in the following order: $A(l,p)$ -> $A(l,p+1)$ -> $A(l,p+2)$ -> -> $A(l,p+(h-1))$. Each ring sends the message to its neighbor ring in the following order: $A(l,p)$ -> $A(\oplus_0 l, p)$ -> $A(\oplus_2 l, p)$ -> -> $A(\oplus_{s-1} l, p)$.

Theorem 4.3: Let $A(l,p)$ be the node initiating the broadcasting. If 1) each node can use a maximum of one outgoing link at a time, 2) $A(l,p)$ sends the message to its neighbors in the same ring in the following order: $A(l,p)$ -> $A(l,p+1)$ -> $A(l,p+2)$ -> -> $A(l,p+(h-1))$, 3) $A(l,p)$ sends the message to its neighbor ring in the following order: $A(l,p)$ -> $A(\oplus_0 l, p)$ -> $A(\oplus_2 l, p)$ -> -> $A(\oplus_{s-1} l, p)$, 4) the neighbors use coordinated recursive doubling as per Algorithm *broadcasting* immediately upon receiving the message from $A(l,p)$, then all nodes will receive the s copies of the message in 4s steps.

Proof: In the first phase, the node initiating the broadcasting sends the message to all its neighbors in the same ring in the first h-1 steps. It takes one step from the initiating ring to next ring, $A(l,p)$ -> $A(\oplus_0 l, p)$. Again it sends the message to its neighbor nodes in the same ring. So, the node initiating the broadcast sends the message to all its neighbor ring in h-1+h steps. In the second phase, there are s steps and s rings from the initial ring to final destination ring to receive the message, since order is followed as $A(\oplus_0 l, p)$ -> $A(\oplus_2 l, p)$ -> -> $A(\oplus_{s-1} l, p)$. Thus, the steps in the second phase are s+s(h-1). Therefore, the algorithm requires a total of 4s steps.

In contrast, if a node can send messages through all its outgoing links and also receive from all its incoming links simultaneously, then algorithm *Broadcasting* requires only $\lfloor s/2 \rfloor + (2s-1) + \lfloor s/2 \rfloor$ steps. In the first step, the source node A sends the message to all its neighbors in the same ring. In the next s steps, the neighbors use the "Coordinated" procedure in algorithm *Broadcasting* to deliver the message to all nodes. The following theorem proves that there is no contension for the same link at any node during the entire course of the algorithm.

Theorem 4.4: Let $A(l, p)$ be the node initiating the broadcasting. If a node can receive and send messages simultaneously in all its incoming and outgoing links, respectively, then algorithm *Broadcasting* requires $\lfloor s/2 \rfloor + (2s-1) + \lfloor s/2 \rfloor$ steps.

Proof : The node to send the message to its all neighbor nodes in a same ring takes $\lceil (h-1)/2 \rceil$ steps. There are s+1 steps to find the farest ring in algorithm *Broadcasting*. Suppose not. Then, there exists a step s in which a node p that has to send more than one message in the same direction, say, j. Without loss of generality, we can assume that the messages are from the recursive *Coordinated* initiated by rings $A(\oplus_0 l, p)$ and $A(\oplus_i l, p)$. This implies in step s-1 of recursive *Coordinated* both rings $A(\oplus_0 l, p)$ and $A(\oplus_i l, p)$ send the message in the same direction. Contraction. That implies that there are s+1 rings. Therefore, algorithm *Broadcasting* requires only $\lfloor s/2 \rfloor + (2s-1) + \lfloor s/2 \rfloor$ steps.

For example, Node (110,0) will receive the message through initiating node (000,1) -> (000,0) -> (001,0) -> (001,1) -> (011,1) -> (011,2) -> (111,2) -> (111,1) -> (110,1) -> (110,0) in Fig. 4.3.

4.4 An Optimal Fault-Tolerant Broadcasting Algorithm

In this section, we present and analyze an efficient broadcasting algorithm for the CCC network that will enable any processor to send the message to all other processors nonredundantly. Then we develop an optimal fault-tolerant broadcasting algorithm which can tolerate s-1 rings or s-1 processors in CCC.

In our algorithm, we use three types of information to control the flow of the message in the CCC.

- $s(\oplus_i l,p)$: initiating node $s(l,p)$ sends the message to *i*th ring neighbor node by complementing *i*th position bit.
- $s(\oplus_{(i,j)} l,p)$: initiating node $s(l,p)$ sends the message, " $s(\oplus_i l,p)$ is faulty" to $s(\oplus_j l,p)$ where $0 \le j \ne i \le s-1$.
- $s((l,\oplus_p l), (p \pm i,p))$. : initiating node $s(l,p)$ sends the message, "$s(l,p \pm i)$ is faulty" to $s(\oplus_p l,p)$. This case corresponds to the node failure in the same ring.

We will show the example using the above notation.

Example: Consider a CCC of 3-dimension with 24 nodes in Fig. 4.1. The initiating node (0,1) can send a message to its 2nd ring neighbor, i.e., processor (4,1) by executing $s(\oplus_2 0,1)$. Also, The processor (0,1) can send a message, "node (100,1) is faulty" to its neighbor ring node (010) by executing $s(\oplus_{(2,1)} 0,1)$. Thus, the ring node $\oplus_1 000$ gets the information that the ring node $\oplus_2 000$ is faulty. The node (0,1) sends the information, "node (0,2) is faulty", to the node (2,1) by executing $s((0,\oplus_1 000), (2,1))$.

The route that the broadcasted message will take can be shown using a tree where the nodes and arcs of the tree correspond to the nodes and links of the CCC respectively. Furthermore, the root of the tree represents the source.

Al-Dhelaan[12] have developed an algorithm for broadcasting in the CCC. This algorithm sends the message to all other nodes non-redundantly, meaning that broadcasted message is sent to each processor exactly once. Their fault-tolerant broadcasting algorithm works only if all the processors in one ring are faulty. The algorithm does not make explicit use of the properties of the CCC topology. We will show the fault-tolerant broadcasting example developed by Al-Dhelaan[12] in Fig. 4.4

.

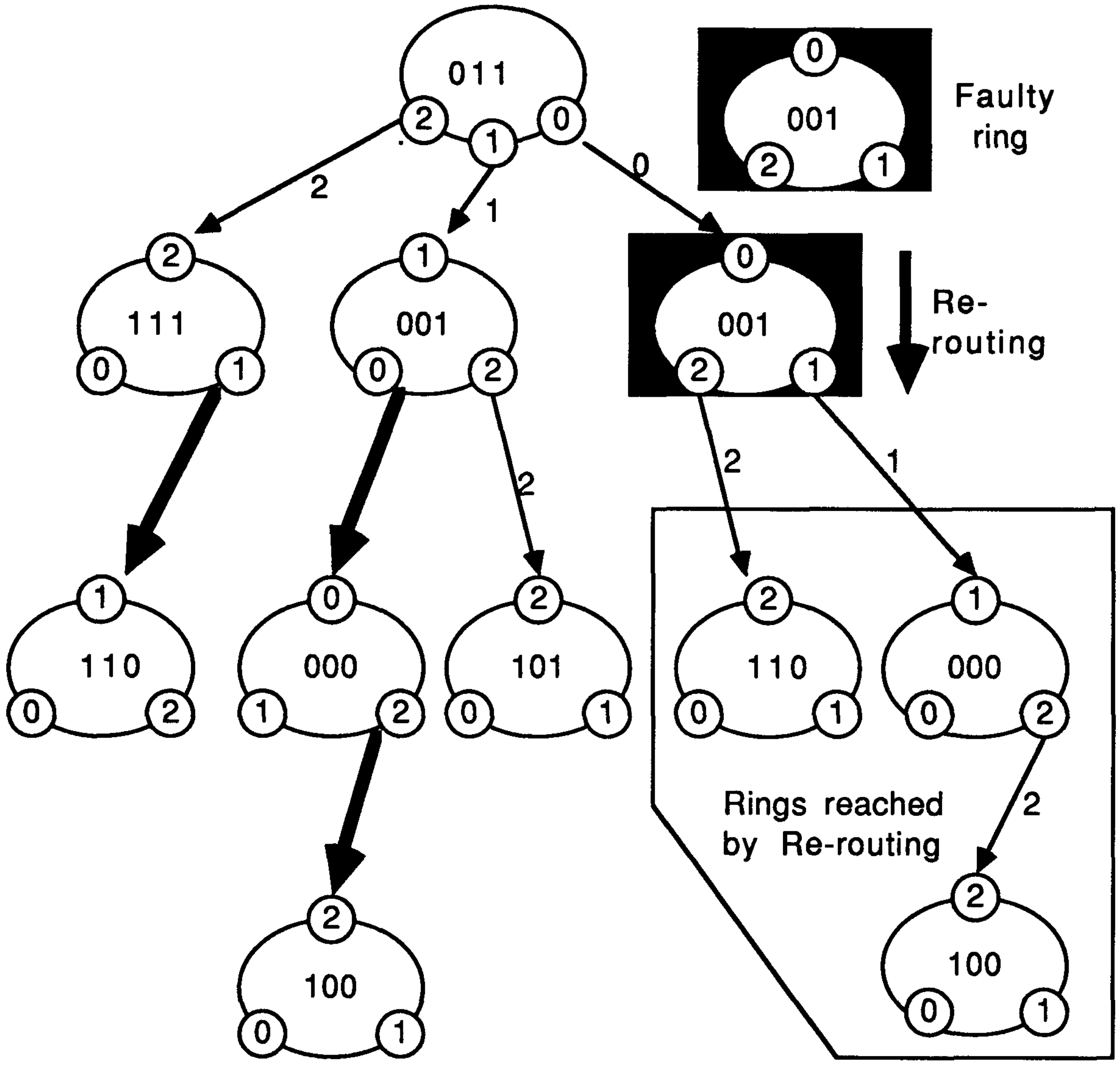


Fig. 4.4 Broadcasting in a faulty CCC with h=3,s=3

Let us explain the algorithm briefly. Initiating ring 011 finds the ring 001 faulty. So, ring 011 gives the information to ring 001 and 111, which send the message to ring 000 -> 100 and 110 respectively. Here when one ring is faulty, the two non-faulty rings are used to broadcast the message, while our algorithm to be shown in the next section uses only one ring.

We define $N=h*2^s$ to mean the total number of nodes in a CCC. Also for the broadcasting tree T, we define N(T) to be the number of nodes in such a tree.

Our broadcasting algorithm proceeds in two phases. In the first phase, the node initiating the broadcasting sends the message to all its neighbors in the same ring and sends the message to the node in the neighbor ring. In the second phase, the neighbor rings use procedure "coordinated" recursively to find the next neighbor ring according to the direction received from the initiating ring and broadcast the message to all the nodes in the subtree. The sequence of directions used by these neighbors in their "Coordinated" phases is coordinated in order to ensure that each ring gets the broadcasting message in sequence. A formal description of the algorithm is given below.

```
**Algorithm** Broadcast(A(l,p));          /* A: initiating node */
**begin**
      Generate the message
      **for** $1 \le i \le \lceil h-1/2 \rceil$ **do**
          send message from A($l,p$) to A($l,p$ + i mod h)  /* forward in the ring */
      **for** $1 \le i \le \lfloor h-1/2 \rfloor$ **do**
           send message from A($l,p$) to A($l,p$ - i mod h)  /* backward in the ring */
      **for** $0 \le i \le s-1$ **do begin**
            send message from A($l$,i) to A($\oplus_i l$, i);  /* send the neighbor ring */
            Coordinated (A($\oplus_i l$, i), i+1)
      **end;**
**end;**

**procedure** Coordinated (m(p,q),k)  /* m: initiating node;   $d_k$: starting direction */
**begin**
      **for** $0 \le i \le s-1$ **do begin**
            **for** $1 \le i \le \lceil h-1/2 \rceil$ **do**
                send message from m($p,q$) to m($p,q$ +i mod h)  /* forward in the ring */
            **for** $1 \le i \le \lfloor h-1/2 \rfloor$ **do**
                 send message from m($p,q$) to m($p,q$ -i mod h) /* backward in the ring */
            R := {m};  /* R: set of rings that have received the message */
            **for each** ring j $\in$ R
              **do**
```

```
                    for k+1≤i≤n-1 do begin
                        Adjacent_Ring = {$\oplus_{i \bmod s}(j)$}
                        Calculate the position(po) for the processor to connect two
                            rings using the definition of interconnection of CCC
                        send message from j.q to $\oplus_i(j)$.po;
                        for $1 \le i \le \lceil h-1/2 \rceil$ do
                            send message from $\oplus_i(j)$.po to $\oplus_i(j)$.(po+i mod h)
                        for $1 \le i \le \lfloor h-1/2 \rfloor$ do
                            send message from $\oplus_i(j)$.po to $\oplus_i(j)$.(po-i mod h)
                        if i < s-1 then
                          R := R ∪ {$\oplus_i(j)$}; /* all receiving rings are added */
                    end;
                    k := k+1;
                    R := R -{j}; /* source ring is deleted */
                until R = empty;
        end;
end;
```

Let us explain the algorithm briefly. In the first phase, the initiating node sends the message to its neighbor nodes in the same ring using "send message from A(*l,p*) to A(*l,p*±i mod h)" and its neighbor ring node using "send message from A(*l*,i) to A($\oplus_i l$, i)".

In the second phase, each neighbor ring node becomes initiating node and broadcasts the message to its children ring nodes according to new direction. Whenever new child ring is broadcasted, it is added to its father ring for another broadcasting. Finally initiating ring is deleted from the ring set, thus the remaining rings send the message to its neighboring rings until there is no ring to send the message. In case that a ring receives the direction *s-1*, there is no ring to broadcast.

For example, Fig. 4.5 shows the broadcasting of a message from source node (011,1) to all other processors in a CCC. Given below is an example to illustrate the basic idea of the algorithm.

Example : Consider the CCC in Fig 4.5. Let node (3,1) initiate the broadcasting. In the first phase, node (3,1) sends the its message to nodes (3,0) and (3,2). Then node (3,1), (3,0) and (3,2) sends the message to neighbor ring node (1,1), (2,0) and (7,2)

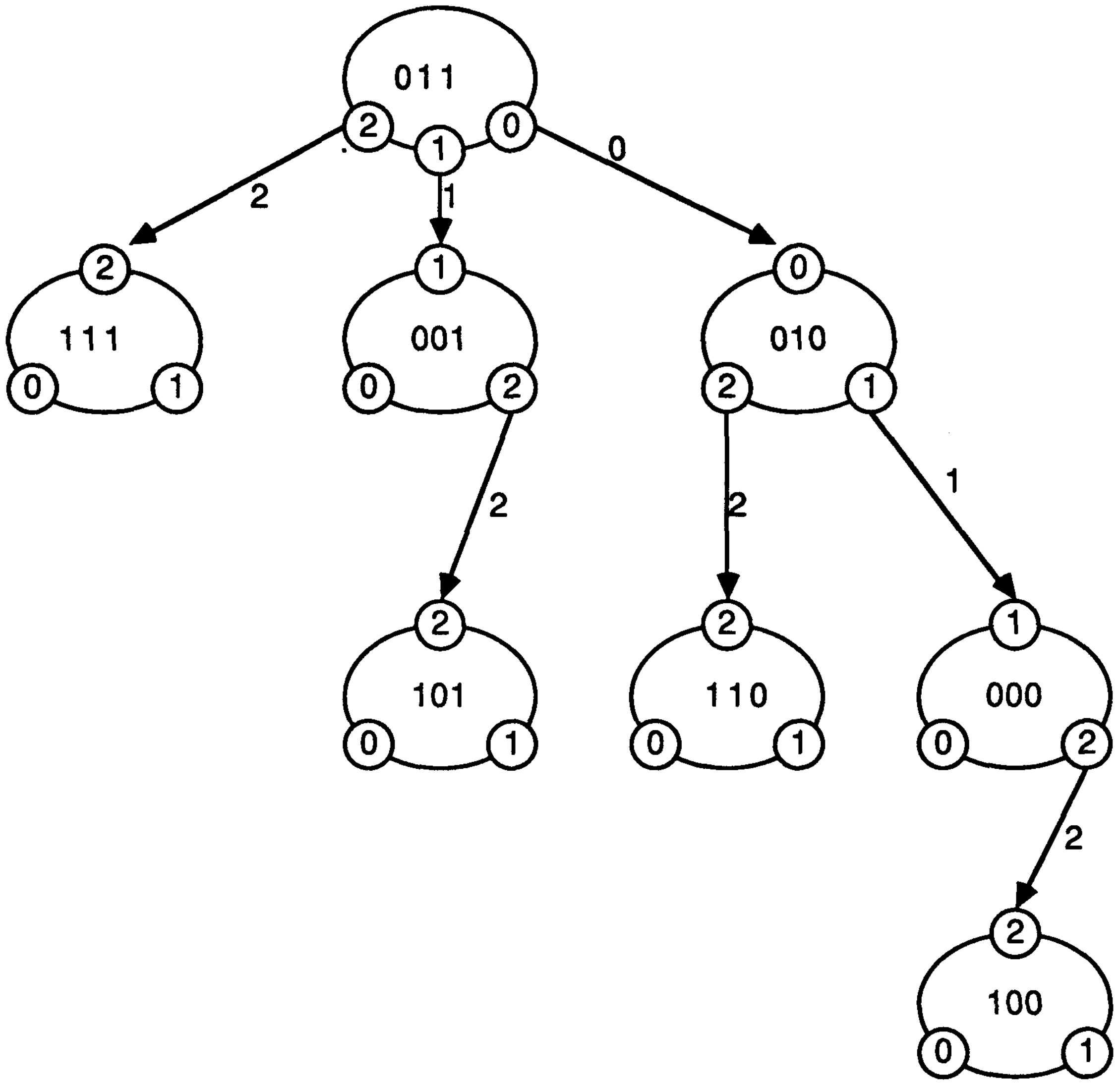


Fig 4.5 Broadcasting in a CCC with s=3, h=3 from the node (011,01)

respectively. In the second phase, node (2,0) sends to (2,1) and (2,2) and uses the procedure *coordinated* to broadcast the message it received from the ring 3 to ring 0 along d_1, then ring 6 along d_2 , and finally ring 0 sends the message to ring 4 along d_2. Similarily, ring 1 sends the message to ring 5 along d_2. This processing is done continually until there is no ring to send the message, which means that the ring set becomes the empty.

We describe some Lemmas that are needed to prove that the algorithm sends the message to each node non-redundantly. First, we will describe the binomial tree in [16].

Definition[16] : Binomial trees are defined as follows: For each $k \geq 0$, we define class B_k of ordered trees as follows:

1. Any tree consisting of a single nodes is a B_0 tree.

2. Suppose that Y and Z are disjoint B_{k-1} trees for $k \geq 1$. Then the tree obtained by adding an edge to make the root of Y become the the leftmost offspring of the root of Z is a B_k tree. All binomial trees having a given index are isomorphic in the sense that they have the same shape. Binomial tree has the following properties.

Lemma 4.1[16] : Let Z be a B_k tree. Then

1. Z has 2^k nodes.

2. Z has ${}_kC_l$ nodes on level l.

When we think all processors in a ring as one node, we have the following Lemmas.

Lemma 4.2[17-18] : The broadcasting tree developed by the algorithm *Broadcast* is binomial tree.

For this sake, we have the following corollary.

Corollary 4.1. Let N(T) be a B_s tree for CCC. Then

1. N(T) has $h*2^s$ nodes.

2. N(T_l) has $h*{}_sC_l$ nodes on level l.

Lemma 4.3[17-18]: The nodes in the broadcasting tree are distinct among themselves.

Now we want to prove the following Theorem.

Theorem 4.5[17-18]: The algorithm *Broadcast* sends the message to all nodes in the system nonredundantly.

Al-Dhelaan[12} also developed an fault-tolerant broadcasting algorithm which works if all processors in a ring are faulty. In the rest of this section we describe a fault-tolerant broadcasting algorithm in the presence of s-1 node failure in a CCC.

The basic idea of our algorithm is as follows. The node s(l,p) that wants to broadcast a message checks s($l, p \pm i$) in the same ring. If s($l, p \pm i$) are faulty, s(l,p) give the information,"s($l, p \pm i$) are faulty", to s($\oplus_p l, p$), that is, s(($l,\oplus_p l$), ($p \pm i,p$)). That node becomes be the new initiating node and sends the message to non-faulty nodes under s($l, p \pm i$). If s($\oplus_i l$, p) where $0 \le i \le s-1$ in different ring is faulty, s(l,p) give s($\oplus_{(i,j)} l$, p)) if s($\oplus_j l$, p) is non-faulty where $i+1 \le j \le s-1$. Then s($\oplus_j l$, p) becomes the new initiating node and sends the message to the other nodes in the broadcasting tree developed by s($\oplus_i l$, p) in the procedure *Coordinated.* The neighbor rings in the subtree also follow the same procedure as the initiating ring. Later, we consider one case that all his son-neighbor-rings are faulty, resulting in all its brother-rings to become the initiating rings and to send the message. A formal description of the algorithm is given below.

```
**Algorithm** Broadcast(A(1,p));          /* A: initiating node */
**begin**
     Generate the message
     **for** $1 \le i \le \lceil h-1/2 \rceil$ **do**
       **if** A($l,p$ - i mod h) is faulty **then**
          **begin**
             send the message from A($l,p$) to A((1,$\oplus_p l$), (p+i,p));
             send the message from A($\oplus_p l$,p) to A($\oplus_p l,p$ + i mod h);
             send the message from A($\oplus_p l,p$ + i) to A($\oplus_i(\oplus_p l),p$ + i);
             fault-tolerant-source(A($\oplus_i(\oplus_p l),p$ + i),i,p)
          **end**
       **else** send message from A($l,p$) to A($l,p$ + i mod h) /* forward in the ring */
     **for** $1 \le i \le \lfloor h-1/2 \rfloor$ **do**
       **if** A($l,p$ + i mod h) is faulty **then**
          **begin**
             finish normal broadcasting;
             find new ring by complementing pth position of final ring;
             broadcast the neighbors in the same ring;
          **end**
        **else** send message from A($l,p$) to A($l,p$ - i mod h) /* backward in the ring */
     **for** $0 \le i \le s-1$ **do begin**
       **if** A($\oplus_i l$, i) is faulty **then**
          **begin**
             p := i+1; true := 1;
             **while** ((p $\le$n-1) and (true)) **do**
                **begin**
```

```
                    if A(⊕pl, i) is non-faulty then
                       begin
                           if A(⊕i(⊕pl),i) is non-faulty then
                                  send the M from A(l,p) to A(⊕(i,p)l, i);
                                  Send the message from A(⊕pl, i) to A(⊕i(⊕pl),i);
                                  Fault-tolerant-source(A(⊕i(⊕pl),i));
                                  true := 0;
                       end;
                    else p := p+1;
               end;
           if (true = 1) then  /*  all ⊕i(s) rings  are faulty */
             begin
                    send the message from A(⊕il, i) to A(l,p);
                    for i+1≤j≤n-2 do begin
                           send the M from A(l,p) to A(⊕jl, i);
                           All-Sons-Dead(A(⊕jl, i),i);
               end;
          end;
        else
             send message from A(l,i) to A(⊕il, i); /* send the neighbor ring */
             alive := A(⊕pl, i);
             Coordinated (A(⊕il, i), i+1)
             if younger-brother-rings are dead then
                    Younger-Brother-Dead(alive,i);

      end;
end;
```

Let us explain this procedure briefly. First, the initiating node A(l,p) checks its neighbor A(*l*,*p*-i). If it is faulty, node A(l,p) sends this information to A((l,$\oplus_p l$), (p+i,p)). When A($\oplus_p l$,p) receives the information, "A(*l*,*p*-i) is faulty", it sends the message to its neighbor node, A($\oplus_p l$,*p*+i), in the same ring and finds its new neighbor ring, A($\oplus_i(\oplus_p l)$,*p*+i)), under A(*l*,*p*-i). Finally, A($\oplus_p l$,*p*+i) sends the message to A($\oplus_i(\oplus_p l)$,*p*+i)), which will be the new initiating ring and sends the message to all non-faulty nodes. Second, when its neighbor A(*l*,*p*+i) is faulty, the younger brother rings of faulty ring will not receive the message. Since all rings are in the binomial tree, the younger brother rings will receive the message from rings in the next level, so A($\oplus_p l$,p) will finish normal broadcasting then leaf ring can send the message to the younger brother rings of faulty ring by complementing *direction p*. Third, when its neighbor ring, A($\oplus_i l$, i), is faulty, A(l,p) checks A($\oplus_p l$, i) where $i+1 \le p \le s-1$. If one of them is non-faulty, send the message from A($\oplus_p l$, i) to A($\oplus_i(\oplus_p l)$,i), which becomes the new initiating ring and the sends the message to all non-faulty nodes. Here we have 2 more procedures to handle : 1) In case that all son-rings are faulty under the initiating ring. 2) In case that A($\oplus_i l$, i) is non-faulty but A($\oplus_j l$, i) is faulty where $j > i$.

Fig. 4.6 will show the example of fault-tolerant broadcasting when s-1 nodes are faulty in the same ring. Fig. 4.7 will show the example of fault-tolerant broadcasting when s-1 nodes are faulty in the different ring.

Now we can explain the *Fault-Tolerance-Source* procedure under the new initiating ring in case of ring failure. At that time, the broadcasting tree under the faulty ring is re-organized. The direction of the new initiating ring is different from fault-free broadcasting. The number of son-rings of the new initiating ring is one less than that of the faulty ring because the original initiating ring is faulty. Here we assume that there is no faulty ring under the new initiating node.

```
procedure Fault-Tolerance-Source(s',d,k);  /* s': ring, k : non-faulty direction
                                               d : faulty direction  */
begin
        if k ≠ n-1 then
          begin
               for  1≤i≤n  do
                    if (k-i) mod n > d then
                              Send the message from s' to $\oplus_{k-i \bmod n}(s')$;
          end;
        else begin
               for  1≤i≤n-1  do
                    if i>d  then
                              Send the message from s' to $\oplus_i(s')$;
             end;
        Fault-Coordinated($\oplus_{k-i}(s')$,k-i,k,d)
end;
```

We will describe the procedure *Fault-Coordinated*, which is the procedure under the new receiving ring in case of initiating ring failure. We assume that there is no faulty ring, since the fault checking routine is the same. Here, the direction is also different from the procedure *Coordinated.*

```
procedure fault-coordinated (m,k,bound,d)        /* bound: non-faulty direction,
                                                    d: faulty direction */
begin
    R := {m};  /* R: set of rings that have received the message */
    for each node j ∈ R
      do
        for (k-1) mod n ≤i≤bound  do begin        /* k-1, k-2, . . . */
            if i <> d then
            send the message from j to ⊕i(j);
            if i < bound then
                R := R ∪ {⊕i(j)};  /* all receiving nodes are added */
        end;
        k := k+1;
        R := R -{j};  /* initiating node is deleted  */
      until R = empty;
end;
```

It depends on that which non-faulty node becomes the new initiating node to send the message. Here we find an interesting direction of message in the procedure *fault-coordinated.* The initiating ring s sends the message to $\oplus_i(s)$ where $0 \le i \le n-1$. When the node $\oplus_j(s)$ receives the message, "the $\oplus_i(s)$ is faulty", we have the following 2 directions.

1) if j=n-1, then the direction is the same as the normal broadcasting: d_0 -> d_1 -> . . . -> d_{n-1} excluding the d_i.

2) if $j \ne n-1$, then the direction is $d_{(j-k) \bmod n}$ where $1 \le k \le n$ excluding the d_i

When the initiating ring is faulty, the subtree is reorganized under the new initiating ring. In the binomial tree, Hamming distance between source-ring and son ring is 1, while Hamming distance is 2 among son-rings. Since one of son-rings becomes the new initiating ring, all its brother-rings can't receive the message directly from the new initiating ring, leaving them the leaf rings.

Now we can explain the procedure that can handle in case of all son rings are faulty. Then all younger brother-rings become the initiating rings. After they finished normal broadcasting procedure, all rings except initiating ring send the message to $\oplus_p$(all-rings). This example corresponds that ring 000 and ring 110 are faulty in Fig 4.6. In this case ring 100 receives the message from ring 101.

```
**procedure** all-sons-dead (D(l,p));
**begin**
        finish normal operation using "coordinate" procedure;
        send the message from all-nodes to $\oplus_p$(all-nodes);
**end;**
```

We have another procedure that can handle in case of all brother nodes are faulty. This example corresponds that ring 001 and ring 111 are faulty in Fig 4.6. In this case ring 101 receives the message from ring 001 -> 010 -> 000 -> 100 -> 101.

```
**procedure** Younger-Brother-Dead(D'(m,i));
**begin**
        finish normal broadcasting using procedure *Coordinated*;
        follow *i* direction until leaf ring;
        send the message from leaf ring to $\oplus_i$(leaf ring);
**end;**
```

In the rest of the section we will show several examples that can tolerate s-1 processors failure and broadcast the message using our algorithm *Broadcast*. First of all, when s is 3 in a CCC, we can show the optimal fault-tolerant routing in the presence of 2 processors failure under the initiating node in Fig. 4.6.

Let us explain the basic idea of the algorithm in Fig. 4.7. We assume that node(1,1) and node(2,0) be faulty. Node s(011, 1) is the initiating node to send the message and sent the message to node (011,0) and (011,2). When it checks the neighbor ring node using s($\oplus_0$011, 0) and finds that it is faulty, it checks another neighbor using s($\oplus_1$011, 1) and finds that it is also faulty. So, s(011, 1) gives the information to s($\oplus_{(0,2)}$011, 2), which sends the message to node (7,1) and (7,0). Then those send the message to the neighbor rings {$\oplus$($_0$($\oplus_2$011))=110} and {$\oplus$($_1$($\oplus_2$011))=101}, which will become the new initiating rings and send the message to all the non-faulty rings, {100,010} and 001, originally developed by ring s($\oplus_0$011, 0) and s($\oplus_1$011, 0) respectively. Here, node(2,2) and node(1,2) sends the message to node(2,1) and node(1,0) respectively. Finally ring 100 sends the message to ring 000 according to *direction* 2.

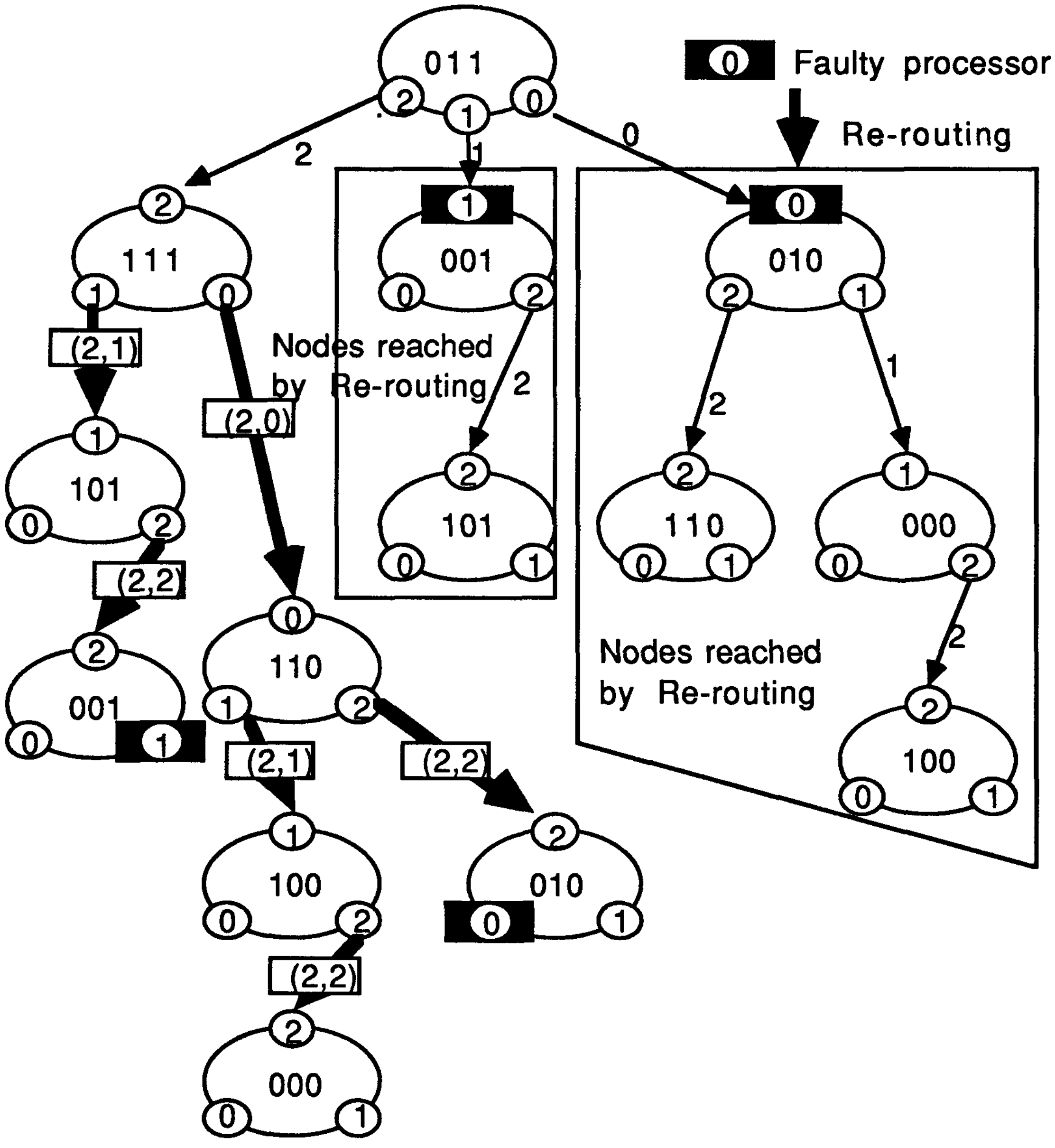


Fig. 4.6 Broadcasting in a CCC with faulty nodes (001,1) and (010,0)

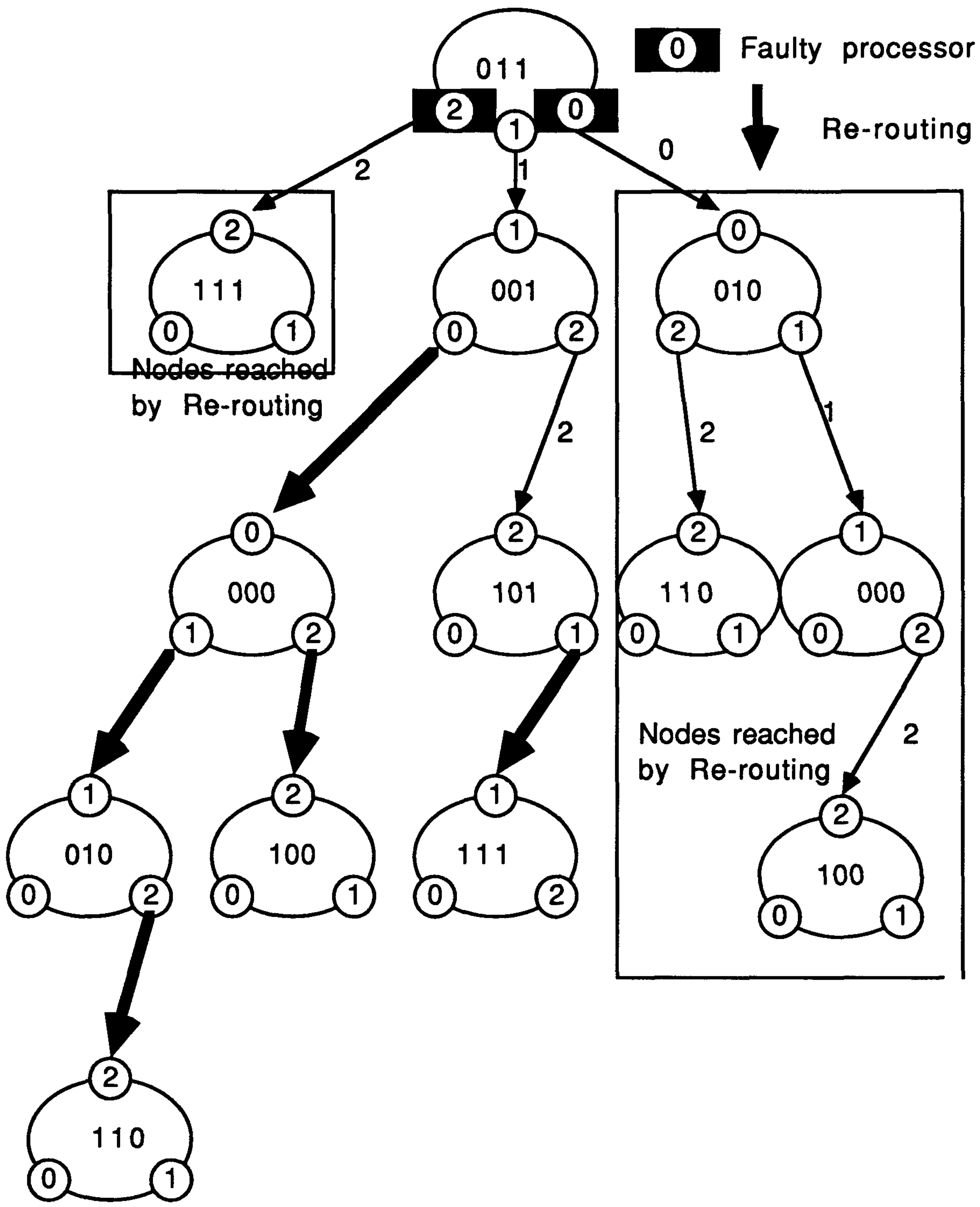


Fig. 4.7 Broadcasting in a CCC with faulty nodes (011,0) and (011,2)

Another example shown in Fig. 4.7 is to have 2 nodes failure in the same ring under the initiating node. Let node s(3,1) be the initiating node and node(3,0) and (3,2) be faulty. When the node(3,1) sends the message to node (011,0) and (011,2), it finds them to be faulty. So, node (011,1) give the information to s((011,$\oplus_1$(011), (0,0)) and s((011,$\oplus_1$(011), (2,2)). When the older brother ring is faulty, ring 001 sends the message to $\oplus_0$(001) according to *direction 0*, which will be the new initiating ring to send the message to ring $\oplus_1$(000) and $\oplus_2$(000) according to *direction 1* and *2* respectively in the procedure *Fault-Tolerant-Source*. Finally ring 010 sends the message to ring $\oplus_2$(010) according to *direction 2* in the procedure *Fault-Coordinated*. When the younger brother ring is faulty, follow the broadcasting tree until the direction is the same as the direction 2 and the ring 111 under the faulty node receives the message from the leaf ring by complementing $\oplus_1$(101). We will formally prove this in the next section.

4.6. Analysis of Broadcasting Algorithm

First, we will evaluate the performance of Algorithm *Broadcast* in terms of the number of steps required to complete the delivery mechanism. It is shown in Theorem 4.6 below that the delivery of the message can be completed in $\lceil (h-1)/2 \rceil + s(1+\lceil (h-1)/2 \rceil)$ steps.

Theorem 4.6 : Algorithm *Broadcast* takes $\lceil (h-1)/2 \rceil + s(1+\lceil (h-1)/2 \rceil$ steps to send the message to all nodes in a CCC.

proof : Let A(l,p) be the node initiating the message. In the first phase it sends the message to its neighbor node through A(l, $p \pm i$) in the same ring where $0 \le i \le h-1$. If it sends the message simultaneously, it will take $\lceil h-1/2 \rceil$ steps. In the second phase, A(l,p) sends the message to its neighbor ring in the following order: A($\oplus_0 l$, p), A($\oplus_1 l$, p), . . A($\oplus_s l$, p). It will take s steps from source ring to final destination ring and $\lceil (h-1)/2 \rceil$ steps is required in each ring. Therefore, the total number of steps from the source node to final destination node is $\lceil (h-1)/2 \rceil + s(1+\lceil (h-1)/2 \rceil)$ steps.

In the rest of this section we want to prove that our algorithm is optimal in case of s-1 nodes failure. We have the following Lemmas in order to prove. First we start with the initiating node to broadcast the message. We know that all nodes in a ring are faulty, it is the same as one ring failure. Our algorithm doesn't care how many nodes failure in a ring. The thing to be considered is that the faulty nodes are in a same ring or different ring. If we can reach the non-faulty ring, we can send the message to non-faulty nodes in a ring. Thus, we consider the ring faults.

Lemma 4.4 : In broadcasting tree, if s-1 ring are faulty under the initiating ring s, then we can send the message to one of son-node under the faulty node.

proof: Let A be the ring initiating broadcast. Ring A checks the $\oplus_i(s)$ where $0 \le i \le s-1$. let us assume that only $\oplus_j(s)$ where $0 \le j \ne i \le s-1$ is non-faulty. Let one son ring be $\oplus_j(\oplus_i(s))$ under $\oplus_i(s)$. When $\oplus_i(s)$ is faulty, s gives $\oplus_{(i,j)}(s)$ to $\oplus_j(s)$. Ring $\oplus_j(s)$ gives the message to $\oplus_i(\oplus_j(s))$ in the algorithm *Broadcast* and $\oplus_i(\oplus_j(s))$ becomes the initiating ring. If $\oplus_i(s)$ is non-faulty, the message is sent through s -> $\oplus_i(s)$ -> $\oplus_j(\oplus_i(s))$. However, when $\oplus_i(s)$ is faulty, the message is sent through s -> $\oplus_j(s)$ -> $\oplus_i(\oplus_j(s))$. Here, we find ring $\oplus_j(\oplus_i(s))$ is equal to ring $\oplus_i(\oplus_j(s))$. Thus we can reach one of son-rings in the faulty ring.

For example, if rings 010 and 001 are faulty, non-faulty ring 111 sends the message to 110 and 101, respectively in Fig. 4.6.

When the father ring is faulty, one of the son rings will receive the message from one of the father's brother rings and will be the new initiating ring and the broadcasting tree is reorganized. The message is sent to the non-faulty ring under the faulty father ring. Then we have the following Lemma.

Lemma 4.5 : In broadcasting subtree, if one of son rings is non-faulty, then it can be the initiating ring and send the message to all other rings in the same broadcasting subtree.

proof : Let the initiating ring s be faulty and $\oplus_i(s)$ be non-faulty. The $\oplus_i(s)$ receive the information, "s is faulty". Then $\oplus_i(s)$ calls the procedure *Fault-Tolerance-Source* and becomes the initiating ring s' and sends the message to its neighbors in the next level and calls the procedure *Coordinated.* At that time, since $H(s', \oplus_i(s')) = 2$, s' can't broadcast directly to $\oplus_i(s')$. Thus, when the broadcasting tree is reorganized, all the brother-rings of s' receives the message from their son-rings. That is, $\oplus_i(s')$ must be leaf rings when the procedure *Fault-Tolerance-Source* is called.

For example, ring 110 becomes the new initiating ring and sends the message to rings 100 and 010 in Fig. 4.7. Those nodes send the message to all the subtrees through procedure *fault-coordinated.* Here we can see that the ring 001 and 010, brother rings of ring 111, became the leaf rings. sends the message to ring 000.

Now we can prove the procedure that can handle in case that all son rings are faulty except the initiating ring. In that case, all younger brother rings become the initiating rings.

Lemma 4.6 : In broadcasting subtree, if all son-rings are faulty, then the initiating ring's all younger brother-rings can be initiating rings and send the message to all non-faulty rings under the faulty son-rings..

proof : Let s', m and m' be initiating ring, $\oplus_p(s')$ and $\oplus_i(s')$ respectively where i£p£n-1. Let $\oplus_i(m)$ where $0 \le i \le n-1$ be faulty. If $\oplus_i(m)$ is non-faulty, the son $\oplus_j(\oplus_i(m))$ of $\oplus_i(m)$ will receive the message through s' -> $\oplus_p(s')$ -> $\oplus_i(\oplus_p(s')$ -> $\oplus_j(\oplus_i(\oplus_p(s')))$. Since $\oplus_i(m)$ is faulty, m gives the information, "$\oplus_i(m)$ is faulty" to s' and s' to m'. Thus, m' sends the message to $\oplus_j(m')$ and finally to $\oplus_p(\oplus_j(m'))$. So, if $\oplus_i(m)$ is faulty, the the son $\oplus_j(\oplus_i(m))$ of $\oplus_i(m)$ will receive the message through s' -> $\oplus_i(s')$ -> $\oplus_j(\oplus_i(s')$ -> $\oplus_p(\oplus_j(\oplus_i(s')))$. Here, $\oplus_j(\oplus_i(\oplus_p(s')))$ is equal to $\oplus_p(\oplus_j(\oplus_i(s')))$. Thus, any son ring under all faulty ring can receive the message from the grand-father's younger brother-rings.

Lemma 4.7 : In broadcasting tree, if s-1 nodes are faulty in the same initiating ring $s(l,p)$, then we can send the message to one of grandson-rings under the faulty node.

Proof : Let $s(l,p-i)$ be faulty. Then $s(\oplus_{p-i}(l),p-i)$ will not receive the message. The $s(l,p)$ sends the message to $s(\oplus_p(l),p)$. Thus, $s(\oplus_p(l),p)$ will send the message to $s(\oplus_{p-i}(\oplus_p(l)),p))$ which is the one of son rings of $s(\oplus_{p-i}(l),p-i)$. So, $s(\oplus_{p-1}(\oplus_p(l)),p))$ sends the message to $s(\le_{p-i}(l),p-i)$. Therefore, $s(\oplus_p(\oplus_{p-i}(\oplus_p(l))))$ under faulty ring is equal to the $s(\oplus_{p-i}(l))$ under non-faulty ring.

For example, we have 2 nodes failure in a initiating ring in Fig. 4.7. If there is no faulty node, the ring 010 will receive the message from ring 011 directly, whereas ring 010 receive the message through 011 -> 001 -> 000 -> 010 under faulty node.

Lemma 4.8 : In a broadcasting tree, left(right) subtree rings of the initiating ring are mapped directly to right(left) subtree rings by one step.

proof : Let the ring in a CCC be the node in a Hypercube. Then according to Lemma 3.2, the broadcasting tree developed by the algorithm *Broadcast* is a binomial tree. All binomial trees having a given index are isomorphic in the sense that they have the same shape. Therefore, we can map left(right) subtree rings to right(left) subtree rings by one step.

All the above Lemmas 4.4, 4.5, 4.6, 4.7 and 4.8 lead to the following theorem.

Theorem 4.7: Our broadcasting algorithm tolerate s-1 processors or s-1 rings failure.

Proof: From Lemma 4.4, the initiating ring to send the message can tolerate s-1 rings failure. This Lemma corresponds to algorithm *Broadcast* and *Younger-Brother-Dead.* From Lemma 4.5, if one of son-rings is non-faulty, it becomes the new initiating ring and sends the message to all the non-faulty rings in the sub-broadcasting tree. This Lemma corresponds to procedure *Coordinated.* From the Lemma 4.6, if all son rings are faulty, all initiating node' younger brother rings can be the initiating rings and send the message to all the non-faulty rings in the sub-broadcasting tree. This corresponds to procedure *All-Sons-Dead.* Finally, according to Lemma 4.7, if s-1 nodes are faulty in a ring, we can send the message to non-faulty nodes in CCC. These Lemmas can be applied to any level of subtrees. Therefore, our algorithm can tolerate s-1 processors.

In the rest of this section, we will evaluate the performance of algorithm *Broadcast* in terms of the number of steps required to complete the message delivery. It is shown in Theorem 4.8 below that the fault-tolerant delivery can be completed in n+1 steps. We are not considering the steps to send the information, "the $\oplus_{(i,j)}(s)$".

Theorem 4.8: Our optimal fault-tolerant broadcasting algorithm needs $1+2*\lceil(h-1)/2\rceil + s(1+\lceil(h-1)/2\rceil)$ steps.

Proof : When the ring to initiate the broadcast (say s) send the message, it checks $\oplus_i(s)$ and give the $\oplus_j(s)$ to $\oplus_{(i,j)}(s)$. Then, $\oplus_j(s)$ gives the message to one of sons of $\oplus_i(s)$, which will be the new initiating ring and sends the message to all non-faulty rings as the same step as the normal broadcasting tree. And according to Lemma 4.8, we need one more step for non-faulty node to send the message to another non-faulty node and each ring takes $\lceil(h-1)/2\rceil$. Therefore, we need $1+2*\lceil(h-1)/2\rceil+s(1+\lceil(h-1)/2\rceil)$ steps in this optimal fault-tolerant broadcasting algorithm.

REFERENCES

1. F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles, A Versatile Network for Parallel Computation," Communication of ACM, pp. 30-39, May 1981.

2. J. D. Ullman, Computational Aspects of VLSI, Computer Science Press, 1984

3. P. Banerjee, S. Y. Kuo, W. K. Fuchs, "Reconfigurable Cube-Connected Cycles Architecture", 1986 IEEE

4. H. S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Computers, pp. 153-161, Feb. 1971.

5. R. M. Chamberlain, "Gray codes, Fast Fourier Transformations and Hypercubes,"Parallel Computing, 6, 1988, pp. 458-473.

6. D. Bitton , D. DeWitt, D. Hsiao and J. Menon, "A taxonomy of Parallel Sorting," Computing Surveys, Vol. 16, Sep 1884, pp. 458-473

7. C. L. Seitz, "The Cosmic Cube," Commun. Ass. Comput. Mach. Vol. 28, Jan 1985, pp. 22-33.

8. M. Pease, "The Indirect Binary n-Cube Microprocessor Array," IEEE Trans. on Computers May 1977, pp. 458-473.

9. L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," ACM Trans.Programming language System, pp. 382-401, Jul. 1982.

10. T. K. Srikanth and S. Toueg, "Optimal clock synchronization," J. ACM pp.626-645, Jul. 1987.

11. A. Al-Dhelaan and B. Bose, "Efficient Fault Tolerant Broadcasting Algorithm for the Hypercube," Fourth Conf. on Hypercube Concurrent Comp. and Applications, Mar. 1989, pp. 123-128.

12. A. Al-Dhelaan and B. Bose, "Efficient Fault Tolerant Broadcasting Algorithm for the Cube-Connected Cycles Network", Proc. IEEE Pacific Rim Conference, May 1989, pp. 161-164.

13. P. Ramanathan and K. G. Shin, "Reliable Broadcast in Hypercube Multiprocessor", IEEE Trans. on Computers, Dec. 1988, pp. 1654-1657.

14. T. K. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerating algorithms," Tech. Rep. 84-623, Dep. Comp. Cornell Univ., Jul. 1984.

15. A. Al-Dhelaan and B. Bose, "A New strategy for Processor Allocation in an N-cube Multiprocessor", Phoenix Conference on Computer and Communication, Mar 1989. pp. 114-118.

16. M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms", SIAM J. Comput, Vol. 7, Aug. 1978, pp 298-319.

17. J. E. Jang,"Optimal Fault Tolerant Broadcasting Algorithm foe a Hypercube Multiprocessor", Proc.1990 ACM Computer Science Conference, Feb, 1990. pp. 96-102.

18. J. E. Jang,"Optimal Fault Tolerant Broadcasting Algorithm in an Cube-Connected Cycles Network", Proc. Int'l Conference on Database, Parallel Architectures, and their Applications(PARBASE-90), Mar, 1990. pp. 206-215.

19. J. E. Jang,"Reliable Broadcasting Algorithm for a Cube-Connected Cycles Netwoork", Proc. the 9th Annual Phoenix Int'l Conference on Computers and Communications, Mar, 1990

Chapter 5

Masking Adjacent Asymmetric Line Faults

5.1. Introduction

Error correcting/detecting codes have been extensively discussed for improving the reliability of computer systems and communication networks and memory units[1-16].

The type of error statistics which occur in memory, logic, and arithmetic units are many and varied. We can broadly classify them as symmetric, asymmetric, and unidirectional errors.

Symmetric errors: The error statistics are said to be symmetric when both 1 -> 0 and 0 -> 1 errors can occur simultaneously in a data word.

Asymmetric errors: When the errors in a data word are only one type, say 1 -> 0, these error statistics are called asymmetric. In this case the other type of errors, say 0 -> 1, will never occur in any data word.

Unidirectional errors: When the error statistics in a data word are 0 -> 1 or 1 -> 0 errors, but both types of errors do not appear simultaneously in a word, these are called unidirectional errors , but the decoder does not know *a priori* the type of errors.

These unidirectional and asymmetric error codes have been proposed for power supply failure, stuck-at fault in shift register memories[10] and self-checking logic systems[14]. According to the fault analysis of the ROMs[15], the probability of short-circuits faults to adjacent bus lines is larger than the probability of open-circuit faults. When *adjacent asymmetric error masking codes* are used for short-circuit faults, they are capable of masking single adjacent asymmetric error in bus lines in LSIs.

This paper was initiated by Kazumitsu[15] which announced that theoretic analysis and the systematic derivation of *adjacent asymmetric error masking codes*(AAEMC) remain as future studies. In this paper, when the weight is 2, 3 and 4 in the constant weight codes, we develop the AAEMC using systematic methods and analyzed those codewords and derived an equation to get those codes, especially formally proved the maximality of AAEMC when weight is 2. We want to minimize the number of transistors in the decoder of the bus line circuits.

This chapter is organized as follows. Section 5.2 describes the definition of masking asymmetric line faults. Section 5.3 describes the background and the notation used in this chapter. Section 5.4 describes the method used to develop the adjacent asymmetric error masking codes when the weight is 2, 3 and 4 and the formula to derive the number of codewords. Section 5.5 evaluates the performance and compares our results with the previous results.

5.2. Masking Asymmetric Line Faults

In this section we will briefly describe the masking asymmetric line faults mentioned in [15]. In recent microprocessor LSIs, the area of bus lines has been increasing as word bit length increase. These lines commonly connect circuit elements, e.g., processing elements and memory elements. Thus, defects or faults in these lines seriously influence the LSI yield and reliability.

Fig. 5.1 shows a typical model of the bus line circuit. Information signals are sent on parallel bus lines. Several circuits elements, shown in Fig. 5.1 as circuit A to circuit X, which operate function such as AND and selection, work at positions on the bus lines to obtain the information from all or part of these lines. For example, in a memory address decoder of RAM or ROM LSIs, address information is sent on address bus lines, and only one decoder gate (AND gate), i.e., only one circuit element, is activated according to the information for a memory unit. In this model. since the bus lines are often very long and occupy large chip areas in LSIs, the bus lines are vulnerable to manufacturing defects or noises. Therefore, technologies which tolerate these defects or faults are necessary to improve LSI yield and reliability.

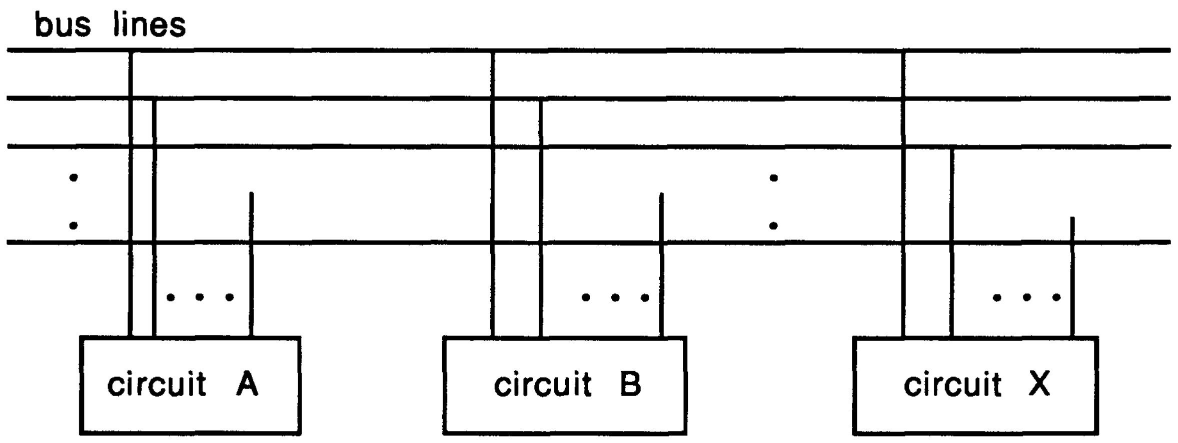


Fig. 5.1 A typical model of bus line circuit

Fig. 5.2 shows a bus line circuits with defect masking, which consists of lines L, encoder E, and circuits elements, d_0 to d_{n-1}. Input information I is encoded into a code C by the encoder E, and then C is sent to the bus line L. The code C is designed to tolerate bus line faults. Each circuit element acts to mask these line faults as well as to operate the function that is originally required. Therefore, we call set of these circuit elements as decoder D. Finally, we can get correct outputs of this bus line circuits.

In the bus line L, short-circuit and open-circuit defects often occur. These defects change the signal line into several levels, i.e., high, low, or medium. However, by controlling the bus driver and the bus terminal gate the level of faulty line can always be made either high or low. For example, for short-circuit defects in the bus lines, the bus drivers are designed to maintain a high level on all bridged lines.

For a short-circuit defects, the bus driver is designed to work in two steps. In the first step, the driver discharges the bus line. In the second step, the driver charges the bus line when the input signal level is high.

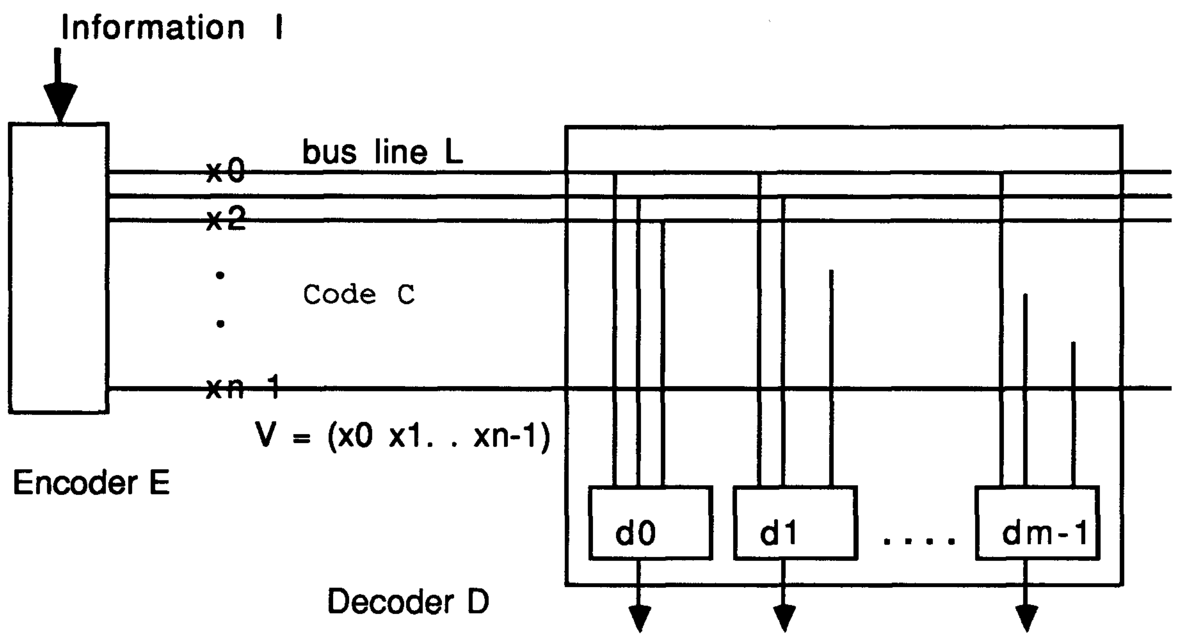
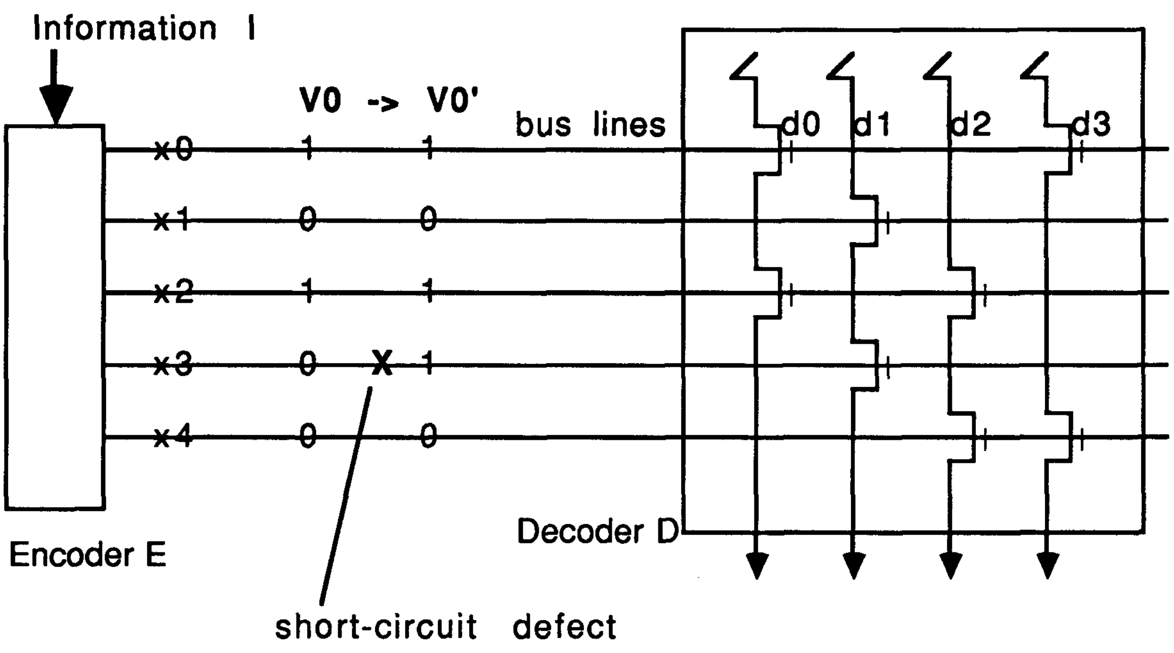


Fig. 5.2 A bus line circuit with defect masking coding

When the levels is low, the driver makes its output impedence high to maintain the bus line at a low level. Therefore, even if there is a short-circuit defect in the bus lines, the bridged lines are always charged up by the driver, i.e., maintained at a high level.

Such a fail-safe design achieves asymmetry in errors. The probability of 1-errors (1 is changed to 0) is made extremely small compared with 0-errors (0 is changed to 1). These defects can also make the line level only low by controlling the construction of the bus driver and the bus terminal gates. In this paper, we will mainly concentrate on 0-errors.

These asymmetric faults can be masked by new coding techniques. Fig. 5.3 illustrates an example of a bus line circuit which can mask single asymmetric 0-errors in the bus lines. The decoder D consists of AND gates d_0 to d_3 corresponding to codewords V_0 to V_3 in code C, i.e., $\{V_0,V_1,V_2,V_3\} \in$ C. Because each gate has transistors at the bus line where the element of the codeword is '1', the gate is only activated by receiving the corresponding codeword. Here we consider the circuit elements of this bus line circuit, e.g., decoder gates d_0 to d_3, as AND gates.



Codeword

$V_0 = (10100)$ $V_1 = (01010)$ $V_2 = (00101)$ $V_3 = (10001)$

$\{V_0, V_1, V_2, V_3\} \in C$

Fig. 5.3 An example of a masking single asymmetric fault

This circuit can work correctly, even if there is a single asymmetric 0-error in the bus lines. For example, we assume that information I is given and then encoded into codeword $V_0 = (10100)$. Furthermore, we assume that one 0-error occurs in the fourth line X_3. The codeword V_0 is changed into $V_0' = (10110)$. However, only one AND gate d_0, which would originally be activated only by V_0 if there is no fault, is activated, because V_0' has 1's at the position where V_0 has '1's. The other AND gates d_1 to d_3 can't be activated for V_0', because V_0' has at least one '0' at the position where V_1 to V_3 have '1'. Hence one asymmetric line fault never causes faulty activation and therefore it can be masked.

Generally, since the bus lines are coded into an asymmetric error masking code, no wrong circuit output is given even in the presence of bus faults. This masking technique has the big advantage that no additional circuits, except for bus terminal gates and additional bus lines, are needed for masking these faults. That is, the output of the bus line circuit is always correct without explicitly using an error correction circuit.

5.3. Preliminaries and Definition

We briefly review the error correcting capabilities of binary block codes for symmetric and asymmetric errors. We start with the following concepts.

Let X and Y be two n-tuples over $GF(2) = \{0,1\}$. We denote the number of 1 -> 0 crossovers from X to Y by $N(X,Y)$.

For example, when $X = (110110)$ and $Y = (001110)$, then $N(X,Y) = 2$ and $N(Y,X) = 1$. Note that in general $N(X,Y) \neq N(Y,X)$.

It is well known that the concept of Hamming distance is useful in discussing the symmetric error correcting/detecting abilities of codes. This is defined below. Without loss of generality, we will always assume the type of the asymmetric error to be 0 -> 1.

Definition : The Hamming distance between two n-tuples X and Y, denoted by $D(X,Y)$, is defined as the number of positions in which the two words differ.

In terms of 1 -> 0 crossovers, we can express the Hamming distance between two n-tuples X and Y as

$$D(X,Y) = N(X,Y) + N(Y,X).$$

Definition : A vector $X = (x_1x_2 \ldots x_n)$ is said to cover another vector $Y = (y_1y_2 \ldots y_n)$ whenever $y_i = 1$, $x_i = 1$ for all $i = 1,2, \ldots n$. When neither covers the other, they are called unordered.

Definition[15] : A code is defined as an error masking code, asymmetric 0-error masking code, if for all $X \in C$, the erroneous word X', i.e., $X' = X + E$, E : 0-error pattern, never covers any other codewords in C.

Definition : A code is defined as an adjacent asymmetric error masking code if for all X $\in$ C, the erroneous word X', i,e., X'= X + E, E:0-error pattern adjacent to 1, never occurs any other codewords in C, i.e., a 0-error can occur only if that 0 is adjacent to a '1'.

For example, the following set of codewords express an adjacent asymmetric error masking codes with n = 6 and w = 2.

$V_0 = (1\ 0\ 1\ 0\ 0\ 0)$
$V_1 = (0\ 1\ 0\ 1\ 0\ 0)$
$V_2 = (0\ 0\ 1\ 0\ 1\ 0)$
$V_3 = (0\ 0\ 0\ 1\ 0\ 1)$
$V_4 = (1\ 0\ 0\ 0\ 1\ 0)$
$V_5 = (0\ 1\ 0\ 0\ 0\ 1)$

If there is a bridging 0-error at the first and second bits in V_0 then the erroneous word will be V_0' = (1 1 1 0 0 0), the second bit '0', adjacent to the first bit '1', is changed to '1'. The erroneous word V_0' never covers any other code words.

Definition : Crossover positions, $C_p(X,Y)$, are defined to be the set of positions in which 1 -> 0 or 0 -> 1 crossover occurs. The starting position is 1 from the leftmost in a codeword.

For example, Let X be (101000) and Y be (100010). Then $C_p(X,Y)$ = (3,5).

Definition : When $C_p(X,Y)$ is (i,j), $C_p(X,Y)+1$ is (i+1,j+1) and $C_p(X,Y)-1$ is (i-1,j-1). For example, $C_p(X,Y) \pm 1$ = (2,4,6)

A code C is defined as an error masking code, more exactly as an asymmetric '0'-error masking code, if " X $\in$ C, the erroneous word X', i.e., X' = X $\oplus$ E, E:'0'-error pattern, never covers any other codewords in C.In case of masking single asymmetric error, we can use the constant weight codes proposed by Graham[16].

For completeness we prove the following theorem which gives the necessary and sufficient conditions for an adjacent asymmetric error masking codes.

Theorem 5.1 : A code C is capable of masking an adjacent asymmetric error codes iff it satisfies the following condition :

for all $X,Y \in C$ with $X \neq Y$ implies

either $N(X,Y) \geq 2$ and $N(Y,X) \geq 2$

or the value of $C_p(X,Y)\pm1$ is '0' where $C_p(X,Y)+1 \leq n$ and $C_p(X,Y)-1 \geq 1$

proof: 1) When $N(X,Y) \geq 2$ and $N(Y,X) \geq 2$, then X and Y have the following type of codewords.

```
X = . . . . 1 . . . . .1. . . . . 0. . . . . 0 . . .
Y = . . . . 0 . . . . .0. . . . . 1 . . . .  1 . . .
```

If there is single 0-error occurs in X, X can't cover Y. Also if there is single 0-error occurs in Y, Y can't cover X. So the condition holds according to the definition.

2) When the value of $C_p(X,Y) + 1$ and $C_p(X,Y) - 1$ is '0' , then X and Y have the following type of codewords.

```
X = . . . . .0 1 0 . . . . .0 0 0 . . .
Y = . . . . .0 0 0 . . . . .0 1 0 . . .
               i             j
```

Here $C_p(X,Y) = (i,j)$. So, $C_p(X,Y) + 1$ is $(i+1,j+1)$ and $C_p(X,Y) - 1$ is $(i-1,j-1)$. The '0' in the position j in X can't change to 1 according to the definition. So, X can't cover Y. The '0' in the position i in Y can't change to 1. So, Y can't cover X. Therefore the condition holds.

In the previous example, if there is 0-error in V_4, V_4' will be one of the following codewords; (1 1 0 0 1 0), (1 0 0 1 1 0) and (1 0 0 0 1 1). However, V_4' never covers any other code words.

5.4. Code Construction

The adjacent asymmetric error masking codes developed in this section are constant weight codes. We can derive the AAEMC when the weight is 2, 3 and 4 in the constant weight codes. First let us define some notations used in this section.

Let E and O be the position of 1 is even and odd from the rightmost in the codeword respectively. We say that code word X is the type E^i if X has exactly i ones in even positions and codeword X is the type O^j if X has exactly j ones in odd positions. Let i and j be the number of 1 bits in even positions E^i and the number of 1 bits in the odd positions O^j, respectively. We represent a sequence of 1's in a codeword by EO, OEO, EEEE and EOEO, etc., depending on the weight. If the number of 1's are neighbored in the codeword, we can express it as OE, EO, EOE, etc., and EO*EO* if they are neighbored in EO and another EO position.

We can design the asymmetric error masking codes using the 1 or 3 level partition of w-out-of-n code when w is 2 or (3 and 4) respectively. Let all the constant codes with weight w be group G. We can partition this group G into G_i subclass where $0 \leq i \leq \lceil w/2 \rceil$. We can repartition this subclass into $G_{i,j}$ subclasses for each subclass. Finally we can repartition this subclass into $G_{i,j,k}$ subclasses. The hierarchical structure of these partitions is shown Fig. 5.4.

First we can partition w-out-of-n codes as follows: In level 1, the codewords are divided into 2 or 3 subgroups according to the weight. When the weight is 2 and 3, we have only two subclasses, E^0 and E^2. But when the weight is 4, we have three subclasses, E^0, E^2 and E^4.

We can't have the codewords with E^1 or E^3 among codewords with E^0, E^2 and E^4, because they can cover each other when the adjacent asymmetric error occurs. For example, codeword X=(10101000) has 0 number of even position, and codeword Y=(11100000) has 1 number of even position. If X becomes (11101000), then X covers Y. Thus, they can't be the same codewords. We will prove this in Lemma 2 below. In level 2, when the weight is 3 or 4, we can repartition this subclass E^2 according to the position of remaining 1's. When the weight is 3, we repartitions the subclass, E^2, into {EEO, EOE, OEE} subclasses. When the weight is 4, we repartitions the subclass, E^2, into {EEOO, EOEO, OEEO, EOOE, OEOE, OOEE} subclasses. In the third level, we can repartition this subclasses according as the numbers of 1's are neighbored in the codeword.

For example, EEO subclass is divided into 2 subclasses, EEO and E$\underline{EO}$. We can describe the third level more clearly in the next subsection.

Let us examine some properties at each level. First of all, We have the following Lemmas when the group G is partitioned in level 1.

Lemma 5.1: The Hamming distance between code with E^{2i} and code E^{2j} in level 1 is at least 4 where $0 \le i,j \le \lceil w/2 \rceil$ and $i \ne j$.

proof: We partition the w-out-of-n codes according to E^{2i} in level 1. Without loss of generality, we assume i is 0. We have the codewords with E^0 and E^{2j}. Here, $H(E^0, E^{2j}) \ge 4$, since $O^k = W - E^0 \ge 2$, which means at least there are 2 ones in odd position in E^0.

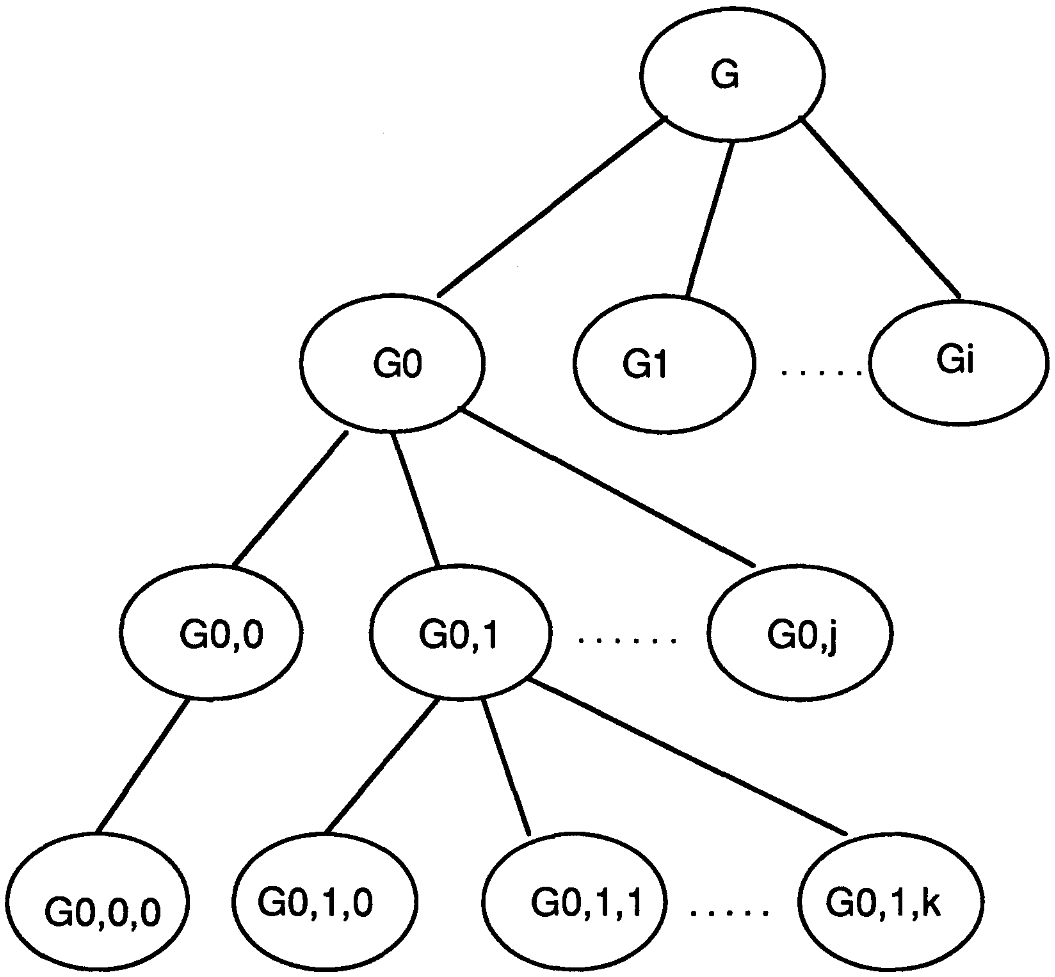


Fig. 5.4. Hierarchical partition of constant weight codes

Lemma 5.2: The codeword G_i with E^{2i} covers the code G_j with $E^{2i\pm1}$ in level 1 when adjacent asymmetric error(AAE) occurs.

proof: We have that $H(G_i,G_j) = 2$. When AAE occurs in G_i, G_i + AAE covers G_j. When AAE occurs in G_j, G_j + AAE covers G_i. That is, when $H(G_i,G_j) = 2$, the codewords are as follows;

G_i : . . . 1 0 . . .
G_j : . . . 0 1 . . .

When AAE occurs in G_i, G_i+ AAE covers G_j: the codewords are as follows.

G_i + AAE : . . . 1 1 . . .
G_j : . . . 0 1 . . .

When AAE occurs in G_i, G_j + AAE_i covers G_i: the codewords are as follows.

G_{ia} : . . . 1 0 . . .
G_{ib} +AAE . . . 1 1 . . .

Lemma 5.1 and 5.2 lead the following important Theorem.

Theorem 5.2 : we can take only E^{2i} subclass for AAEMC in level 1.

In level 2, each subclass has the same numbers of even position and odd position, but the order is different. Here, there is some relationship among the subclass $G_{1,j}$, that is $G_{1,m} \cap G_{1,n} \neq 0$, where $0 \leq m,n \leq j$.

Lemma 5.3: Even though $H(G_{1,m}, G_{1,n})$ is 2, they can't cover each other if they are not adjacent.

proof: When $H(G_{1,m}, G_{1,n}) = 2$, we have the following two cases.

1) When they are adjacent, they cover each other if AAE occurs.

G_i : . . . 1 0 . . . .
G_j : . . . 0 1 . . .

2) when they are not adjacent, they don't cover each other even though AAE occurs.

G_i : . . . 1. . . 0 . . .
G_j : . . . 0 . . . 1 . . .

According the Lemma 5.2, we have to repartition again subclasses in level 2, depending on that '1' in the odd position and '1' in the even position is adjacent or not. Here, we will prove that each codewords in $G_{i,j,k}$ is the adjacent asymmetric error masking codes in Theorem 5.3.

Theorem 5.3: Each codewords in the $G_{i,j,k}$ is the AAEMC.

proof: According to Lemma 5.3, when the number of 1's in odd and even positions are not adjacent, they don't cover each other even though AAE occurs. This satisfies the condition 1 of Theorem 5.1. Thus, we can choose those codewords for AAEMC. Also, we can choose the subclass for AAEMC with same form of positions, which have the adjacent positions. This satisfies the condition of Theorem 5.2.

We have the following example to satisfy Theorem 5.3.

case 1) If $H(G_{i,j,m}, G_{i,j,n}) = 2$ and the position of 1 is not adjacent, then we have the following codewords. Therefore, they don't cover each other even if we have AAE.

```
. . . . . 11 . . . .01000 . . .
. . . . . 11 . . . .00010 . . .
```

case 2) If $H(G_{i,j,m}, G_{i,j,n}) = 4$ and position of 1 is different, then we have the following codewords. Therefore, they don't cover each other even if we have AAE.

```
. . . . . . . . . .01110 . . .
. . . . . . . . . . . . 0111 . . .
```

Section IV-A, IV-B and IV-C describe the AAEMC when the weight is 2 , 3 and 4 respectively.

A. When weight = 2

In constant weight codes the total number of codewords with length *n* and weight 2 are $\binom{n}{2}$. Among them we must choose some codewords to satisfy Theorem 1. we can choose codewords with E^0 and E^2 from the Theorem 2. Let us describe the algorithm to generate the adjacent asymmetric codes using systematic method; The procedure is as follows.

```
step 1. start   000 . . . 0101 ; i = 3 (second '1' position)
step 2. rotate to the left direction until second '1' arrives at the first position
step 3. advance second '1' to the i = i + 2 position
step 4. check  whether  i = n        when n is odd
                        i = n - 1 when n is even
step 5. if not, go to step 2.
```

For example, we will construct the AAEMC with n = 8 and w = 2.

```
0 0 0 0 0 1 0 1
0 0 0 1 0 0 0 1
0 1 0 0 0 0 0 1
0 0 0 0 1 0 1 0
0 0 1 0 0 0 1 0
1 0 0 0 0 0 1 0
0 0 0 1 0 1 0 0
0 1 0 0 0 1 0 0
0 0 1 0 1 0 0 0
1 0 0 0 1 0 0 0
0 1 0 1 0 0 0 0
1 0 1 0 0 0 0 0
```

Let us check these codes generated by above method. All the codewords are E^0 and E^2 from the rightmost position. That satisfies the Theorem 1 and Theorem 2.

The following theorem gives the total number of codewords.

Theorem 5.4 : The total number of codewords generated using above method are $\binom{\lceil n/2 \rceil}{2} + \binom{\lfloor n/2 \rfloor}{2}$.

proof : Let the code length be n and the weight be 2. We have the following codewords for each case.

1) E^0 : $\binom{\lceil n/2 \rceil}{2}$. The number of cases when two '1' are located in odd position.

2) E^2 : $\binom{\lfloor n/2 \rfloor}{2}$. The number of cases when two '1' are located in even position.

For example, when n= 9 and w = 2 we have the following codewords.

1) E^0 : $\binom{\lceil n/2 \rceil}{2} = \binom{5}{2} = 10$ 2) E^2 : $\binom{\lfloor n/2 \rfloor}{2} = \binom{4}{2} = 6$.

Therefore there are 16 codewords.

We have an recurrence relation of codewords between the code lengths. Let n be the length of codeword and the codewords be F(n,2). Let n-2 be the length and the codewords be F(n-2,2). Then we can derive F(n,2) from F(n-2,2). We have the following recurrence equation form the codewords.

Theorem 5.5: Let F(n,2) be the total number of codewords. Then we have the following recurrence relation.

$$F(n,2) = F(n-2,2) + (n-2)$$

proof : As the code length is increased by 2 from the code length n-2, we can have 1 more codeword whenever we take step 2 in the above algorithm. This process repeats until 10100...0. this case corresponds n-2 times. Therefore, we have n-2 more codewords than F(n-2,2).

For example, when n = 11 and n = 8 we have the following codewords.

$$(11,2) = (9,2) + (11 - 2) = 16 + 9 = 25$$

$$(8,2) = (6,2) + (8 - 2) = 6 + 6 = 12$$

The maximality of the codewords are considered in the following Theorem.

Theorem 5.6 : The codewords described above are maximum number of codewords.

Proof : Let us analyze the total codewords with $\binom{n}{2}$. We can divide the codewords into 6 subgroups according to the position of 1's: EE, OO, EO, OE, $\underline{\text{EO}}$, $\underline{\text{OE}}$. According to Theorem 5.2, we have chosen subgroups with EE and OO. If not, we can add at least one codeword from EO, OE, $\underline{\text{EO}}$, $\underline{\text{OE}}$ However, any codeword from EO, OE, $\underline{\text{EO}}$, $\underline{\text{OE}}$ can't join the subgroups with EE and OO, because if the AAE occurs in the codeword, it violates the Theorem 5.1. Contradiction.

B. When weight = 3

First of all, we considers when code length n is greater than 9. When n is less than 9, it corresponds to the special case. According to Theorem 2, We have the following subclasses for the 3-out-of-n code in level 1 and level 2.

G_0 : $G_{00} = OOO$

G_1 : $G_{10} = EEO$ $G_{11} = EOE$ $G_{12} = OEE$

We have the following subclasses in level 3 according to the neighboring position.

G_0 : $G_{00} = OOO$

G_1 : $G_{10} = EEO$: $G_{100} = EEO$, $G_{101} = E\underline{EO}$

$G_{11} = EOE$: $G_{110} = \underline{EOE}$, $G_{111} = \underline{EO}E$ $G_{112} = E\underline{OE}$

$G_{113} = EOE$

$G_{12} = OEE$: $G_{120} = \underline{OE}E$ $G_{121} = OEE$

Here we are faced with the problem to find the more AAEMC. So, we can divide $G_{1,j,k}$ group into 2 subclasses, odd and even, according to the number of neighboring in the codewords. The Odd classes is composed with 1 neighboring in the codeword.

Odd subclass		Even subclass	
$G_{1,0,0} = E\underline{EO}$	$G_{1,1,0} = \underline{EOE}$	$G_{1,0,1} = EEO$	$G_{1,1,3} = EOE$
$G_{1,1,1} = \underline{EO}E$	$G_{1,1,2} = E\underline{OE}$	$G_{1,2,1} = OEE$	
$G_{1,2,0} = \underline{OE}E$			

From the above table, we can see all the codewords in the odd subclass can be covered by those of even subclasses if the AAE occurs. Also we know more codewords in even subclass. Therefore, we can choose the codewords in the even subclass.

<u>Theorem 5.7:</u> We can choose $G_{0,0,0}$, $G_{1,0,1}$, $G_{1,1,3}$ and $G_{1,2,1}$, subclasses for AAEMC.

proof: According to Theorem 5.1, we know that $G_{0,0,0}$ is the AAEMC, which is unrelated with $G_{1,x,y}$ where x,y means don't care symbol. We have the following relations to satisfy the Theorem 1 among the subclasses. $H(EEO, EOE) \geq 4$, $H(EEO, OEE) \geq 4$, $H(EOE, OEE) \geq 4$. Therefore, $G_{0,0,0}$, $G_{1,0,1}$, $G_{1,1,3}$ and $G_{1,2,1}$ are AAEMC

Next, we are considering the number of codewords in each subclass. When we add up all of them, they will be the total number of codewords to satisfy AAEMC.

Theorem 5.8 : We can have the equation to have AAEMC for each subclass.

1. $G_{0,0,0}$ = OOO : $\binom{\lceil n/2 \rceil}{3}$.

2. $G_{1,0,1}$ = EEO : $\sum_{j=6by2}^{n} \sum_{i=jby2}^{n} (\lceil \frac{n-i}{2} \rceil + 1)$

3. $G_{1,1,3}$ = EOE : $\sum_{j=8by2}^{n} \sum_{i=jby2}^{n} (\lceil \frac{n-i}{2} \rceil + 1)$

4. $G_{1,2,1}$ = OEE : $\sum_{j=7by2}^{n} \sum_{i=jby2}^{n} (\lfloor \frac{n-i}{2} \rfloor + 1)$

proof: In case of OOO : The number of cases to locate the odd positions are $\lceil n/2 \rceil$. Therefore, we can choose $\binom{\lceil n/2 \rceil}{3}$. In the case of EEO : First codeword will be . . . 101001. That means, the minimum number of bits are 6. The leftmost 1 in the codeword is shifted to left even position, which will give $\sum_{i=jby2}^{n} (\lceil \frac{n-i}{2} \rceil + 1)$ codewords. then middle 1 shifted, finally leftmost 1 is shifted. Therefore we have $\sum_{j=6by2}^{n} \sum_{i=jby2}^{n} (\lceil \frac{n-i}{2} \rceil + 1)$ codewords. In case of EOE and OEE same procedure is applied.

Therefore we have the following corollary.

Corollary 1: if n≥9, the total number of codewords of AAEMC is as follows:

Total number of AAEMC = Codewords with (OOO + EE0 + EOE + OEE).

For example, when n=16 and w=3, we have the following codewords.

$\binom{\lceil n/2 \rceil}{3} = 56$, $\sum_{j=6by2}^{n} \sum_{i=jby2}^{n} (\lceil \frac{n-i}{2} \rceil + 1) = 56$,

$$\sum_{j=8by2}^{n} \sum_{i=jby2}^{n} (\lceil \frac{n-i}{2} \rceil + 1) = 35, \quad \sum_{j=7by2}^{n} \sum_{i=jby2}^{n} (\lfloor \frac{n-i}{2} \rfloor + 1) = 35.$$

Therefore, 56 + 56 + 35 + 35 = 182 codewords.

Now we are considering the other case which n is less than 9.

1. n = 6 : OOO, OOE
2. n = 7 : OOO, OOE
3. n = 8 : OOO, OOE, OEO

We show the number of codewords in each subclass when n is 6-11 in Table 5.1.

G_i	$G_{i,j}$	$G_{i,j,k}$	Min.	code length					
				6	7	8	9	10	11
0	OOO	OOO	5	1*	4*	4*	10*	10*	20*
1	EEO	$\mathrm{O}\underline{\mathrm{OE}}$	4	3*	3*	6*	6	10	10
		OOE	6	1	1	4	4*	10*	10*
	EOE	$\underline{\mathrm{EOE}}$	4	2	2	3	3	4	4
		$\underline{\mathrm{EO}}\mathrm{E}$	6	1	1	3	3	6	6
		$\mathrm{E}\underline{\mathrm{OE}}$	6	1	1	3	3	6	6
		EOE	8	0	0	1*	1*	4*	4*
	OEE	$\underline{\mathrm{OE}}\mathrm{E}$	5	1	3	3	4	6	10
		OEE	7	0	1	1	4*	4*	10*
2	EEE	EEE	6	1	1	4	4	10	10
TOTAL AAEMC				4	7	11	19	28	44

Table 5.1. number of codewords in each subclass when w =3.

We can show the AAEMC codewords in Appendix 5-I when n=12 and w=3.

C. When weight = 4

First of all, we consider when the code length n is greater than 12. When n is less than 12, it corresponds to the special case. According to Theorem 5.2, We have the following subclasses for the 4-out-of-n code in level 1 and level 2.

G_0 : G_{00} = 0000

G_1 : G_{10} = EE00 G_{11} = EOEO G_{12} = OEEO G_{13} = EOOE

G_{14} = OEOE G_{15} = OOEE

G_2 : G_{20} = EEEE

We have the following subclasses in level 3 according to the neighboring position.

$G_{1,0}$:	$G_{1,0,0} = \mathrm{E\underline{EO}O}$	$G_{1,0,1} = \mathrm{EEOO}$	
$G_{1,1}$:	$G_{1,1,0} = \mathrm{\underline{EOEO}}$	$G_{1,1,1} = \mathrm{E\underline{OEO}}$	$G_{1,1,2} = \mathrm{\underline{EOE}O}$
	$G_{1,1,3} = \underline{\mathrm{EO}\mathit{EO}}$	$G_{1,1,4} = \mathrm{EO\underline{EO}}$	$G_{1,1,5} = \mathrm{\underline{EO}EO}$
	$G_{1,1,6} = \mathrm{E\underline{OE}O}$	$G_{1,1,7} = \mathrm{EOEO}$	
$G_{1,2}$:	$G_{1,2,0} = \underline{\mathrm{OE}\mathit{EO}}$	$G_{1,2,1} = \mathrm{OE\underline{EO}}$	$G_{1,2,2} = \mathrm{\underline{OE}EO}$
	$G_{1,2,3} = \mathrm{OEEO}$		
$G_{1,3}$:	$G_{1,3,0} = \underline{\mathrm{EO}\mathit{OE}}$	$G_{1,3,1} = \mathrm{EO\underline{OE}}$	$G_{1,3,2} = \mathrm{\underline{EO}OE}$
	$G_{1,3,3} = \mathrm{EOOE}$		
$G_{1,4}$:	$G_{1,4,0} = \mathrm{\underline{OEOE}}$	$G_{1,4,1} = \mathrm{O\underline{EOE}}$	$G_{1,4,2} = \mathrm{O\underline{EOE}}$
	$G_{1,4,3} = \underline{\mathrm{OE}\mathit{OE}}$	$G_{1,4,4} = \mathrm{OE\underline{OE}}$	$G_{1,4,5} = \mathrm{\underline{OE}OE}$
	$G_{1,4,6} = \mathrm{O\underline{EO}E}$	$G_{1,4,7} = \mathrm{OEOE}$	
$G_{1,5}$:	$G_{1,5,0} = \mathrm{O\underline{OE}E}$	$G_{1,5,1} = \mathrm{OOEE}$	

We can divide $G_{1,j,k}$ group into 2 subclasses according to the number of neighboring in the codewords.

Odd subclass			Even subclass	
$G_{1,0,0} = \mathrm{E\underline{EO}O}$	$G_{1,1,1} = \mathrm{E\underline{OEO}}$		$G_{1,0,1} = \mathrm{EEOO}$	$G_{1,1,0} = \mathrm{\underline{EOEO}}$
$G_{1,1,2} = \mathrm{\underline{EOE}O}$	$G_{1,1,4} = \mathrm{EO\underline{EO}}$		$G_{1,1,3} = \underline{\mathrm{EO}\mathit{EO}}$	$G_{1,1,7} = \mathrm{EOEO}$
$G_{1,1,5} = \mathrm{\underline{EO}EO}$	$G_{1,1,6} = \mathrm{E\underline{OE}O}$		$G_{1,2,0} = \underline{\mathrm{OE}\mathit{EO}}$	$G_{1,2,3} = \mathrm{OEEO}$
$G_{1,2,1} = \mathrm{OE\underline{EO}}$	$G_{1,2,2} = \mathrm{\underline{OE}EO}$		$G_{1,3,0} = \underline{\mathrm{EO}\mathit{OE}}$	$G_{1,3,3} = \mathrm{EOOE}$
$G_{1,3,1} = \mathrm{EO\underline{OE}}$	$G_{1,3,2} = \mathrm{\underline{EO}OE}$		$G_{1,4,0} = \mathrm{\underline{OEOE}}$	$G_{1,4,3} = \underline{\mathrm{OE}\mathit{OE}}$
$G_{1,4,1} = \mathrm{O\underline{EOE}}$	$G_{1,4,2} = \mathrm{O\underline{EOE}}$		$G_{1,4,7} = \mathrm{OEOE}$	$G_{1,5,1} = \mathrm{OOEE}$
$G_{1,4,4} = \mathrm{OE\underline{OE}}$	$G_{1,4,5} = \mathrm{\underline{OE}OE}$			
$G_{1,4,6} = \mathrm{O\underline{EO}E}$	$G_{1,5,0} = \mathrm{O\underline{OE}E}$			

From the above Table, we can see all the codewords in the odd subclass can be covered by those of even subclasses if the AAE occurs. Therefore we can choose only subclasses in the even subclasses. Among all the even subclasses, we can choose subclasses which don't have the neighboring 1's. Then we have the following Theorem.

Theorem 5.9 : We can choose $G_{1,0,1}$, $G_{1,1,7}$, $G_{1,2,3}$, $G_{1,3,3}$, $G_{1,4,7}$, $G_{1,5,1}$ subclass for AAEMC.

proof: We have the following relations to satisfy the Theorem 5.1 among the subclasses. Hamming distance among all the subclasses is at least 4. Thus, they are AAEMC.

After we choose some subclasses according to above Lemma, we have only $G_{1,1,0}$, $G_{1,1,3}$, $G_{1,2,0}$, $G_{1,3,0}$, $G_{1,4,0}$, $G_{1,4,3}$ subclasses left. In order to get more codewords, we have chosen the following subclasses.

{ EOEO & OEOE }, { EO*EO* & EO*OE* }, { OE*OE* & OE*EO* } are neighbored each other. So, if AAE occurs, they will cover each other. Thus, we can't take both. When we check the number of codewords in each case, { EOEO > OEOE }, { EO*EO* > EO*OE* }, { OE*OE* > OE*EO* } So we have chosen EOEO EO*EO* and OE*OE*

Next, we are considering the number of codewords in each subclass. When we add up all of them, they will be the total number of codewords to satisfy AAEMC.

Theorem 5.10: We can have the equation for each subclass to satisfy AAEMC.

1. $G_0 \quad = \text{EEEE} : \binom{\lfloor n/2 \rfloor}{4}$
2. $G_{1,0,1} = \text{EEOO} : \sum_{j=8\text{by}2}^{n} \sum_{i=j\text{by}2}^{n} (\lfloor \frac{n-i}{2} \rfloor + 1)$
3. $G_{1,1,7} = \text{EOEO} : \sum_{j=10\text{by}2}^{n} \sum_{i=j\text{by}2}^{n} (\lfloor \frac{n-i}{2} \rfloor + 1)$
4. $G_{1,2,3} = \text{OEEO} : \sum_{j=9\text{by}2}^{n} \sum_{i=j\text{by}2}^{n} (\lfloor \frac{n-i}{2} \rfloor + 1)$
5. $G_{1,4,7} = \text{OEOE} : \sum_{j=11\text{by}2}^{n} \sum_{i=j\text{by}2}^{n} (\lfloor \frac{n-i}{2} \rfloor + 1)$
6. $G_{1,5,1} = \text{OOEE} : \sum_{j=9\text{by}2}^{n} \sum_{i=j\text{by}2}^{n} (\lfloor \frac{n-i}{2} \rfloor + 1)$

7. $G_{1,1,0}$ = EOEO : $\lfloor \frac{n-4}{2} \rfloor + 1$

8. $G_{1,1,3}$ = EO*EO* : $\sum_{i=6by2}^{n} (\lfloor \frac{n-i}{2} \rfloor + 1)$

9. $G_{1,4,3}$ = OE*OE*: : $\sum_{i=7by2}^{n} (\lfloor \frac{n-i}{2} \rfloor + 1)$

10. $G_{1,3,3}$ = EOOE : $\sum_{j=10by2}^{n} \sum_{i=jby2}^{n} (\lfloor \frac{n-i}{2} \rfloor + 1)$

11. G_2 = OOOO : $\binom{\lceil n/2 \rceil}{4}$

proof: The same procedure as the weight is 3.

Therefore we have the following corollary.

Corollary 2: if n≥12, the total number of codewords of AAEMC is as follows:

Total number of AAEMC = Codewords with (EOEO+ EO*EO* + OE*OE*: + EEEE + EEOO + EOEO + EOOE + OEEO + OEOE + OOEE + OOOO)

For example, when n=16 and w=3, we have the following codewords.

EEEE = 70, EEOO = 70, EOEO = 7, EO*EO* = 21, EOEO = 35

EOOE = 35, OEEO = 35, OE*OE*: = 15, OEOE = 15, OOEE = 35, OOOO = 70.

Therefore, the total number of codewords are 408.

case 2) When n < 12, we select the following codewords.

1. n = 8 : OOOO, EEOO, EOEO, EO*EO*, EOEO, OEEO, EOOE, OE*OE*, OOEE, EEEE
2. n = 9 : OOOO, EOEO, EO*EO*, OEEO, EOOE, OOEE, EEEE
3. n = 10 : OOOO, EOEO, EO*EO*, OEEO, EOOE, OOEE, EEEE
4. n = 11 : OOOO, EEOO, EOEO, EO*EO*, EOEO, OEEO, OE*OE*, OEOE, EOOE,
   OOEE, EEEE

We show the number of codewords when n is 8 - 13 in Table 5.2.

G_i	$G_{i,j}$	$G_{i,j,k}$	Min.	code length					
				8	9	10	11	12	13
0	OOOO	OOOO	7	1*	5*	5*	15*	15	35
1	EEOO	EEOO	6	4	4	10	10	20	20
		EEOO	8	1*	1	5	5*	15	15
	EOEO	EOEO	4	3*	3*	4*	4*	5	5
		EOEO	6	3	3	6	6	10	10
		EOEO	6	3*	3*	6*	6*	10	10
		EOEO	8	1	1	4	4	10	10
		EOEO	6	3	3	6	6	10	10
		EOEO	8	1*	1	4	4	10	10
		EOEO	6	1	1	4	4	10	10
		EOEO	10	0	0	1	1*	5	5
	OEEO	OEEO	5	3	6	6	10	10	15
		OEEO	7	1*	4*	4*	10	10	20
		OEEO	5	1	4	4	10	10	20
		OEEO	9	0	1	1	5*	5	15
	EOOE	EOOE	6	3	3	6	6	10	10
		EOOE	8	1	1	4	4	10	10
		EOOE	8	1*	1*	4*	4	10	10
		EOOE	10	0	0	1	1*	5	5
	OEOE	OEOE	5	2	3	3	4	4	5
		OEOE	7	1	3	3	6	6	10
		OEOE	7	1*	3	3	6*	6	10
		OEOE	9	0	1	1	4	4	10
		OEOE	9	1	3	3	6	6	10
		OEOE	9	0	1	1	4	4	10
		OEOE	9	0	1	1	4	4	10
		OEOE	11	0	0	0	1*	1	5
	OOEE	OOEE	7	1*	4*	4*	10	10	20
		OOEE	9	0	1	1	5*	5	15
2	EEEE	EEEE	8	1*	1*	5*	5*	15	15
TOTAL CODE				14	21	35	54	87	135

Table 5.2. number of codewords in each subclass when w =4.

For example, we can show the AAEMC codewords in Appendix 5-II when n is 12 and weight is 4.

5.5. Comparison and Application

From the codewords obtained from the section 5.4, we can compare our results with previous results obtained by simulation[15] in Table 5.3.

	Graham's table			New result			ref[15]		
n\w	2	3	4	2	3	4	2	3	4
4	2			2					
5	2	4		4	2		4	2	
6	3	4		6	4		6	4	
7	3	7		9	7	7	9	7	7
8	4	8	14	12	11	14	12	11	14
9	4	12	18	16	19	21	16	15	20
10	5	13	30	20	28	35	20	23	33
11	5	17	35	25	44	54	25	33	50
12	6	20	51	30	60	87	30	49	73
13	6	26	65	41	85	135			
14	7	28	91	53	110	201			
15	7	35	103	66	146	291			
16	8	37	140	80	182	408			

Table 5.3 Comparison between Graham and ref[15] and new result

From the table 5.3, we can get a large number of codewords compared to those of random asymmetric masking codes and those of previous result.

For masking single asymmetric faults, the adjacent asymmetric error code is needed. The number of codewords should be equal to or larger than the number of circuit outputs. i.e., 16 and 32 in the bus line circuits. The code length and the weight should be small in order to reduce the number of bur lines and transistors. According to the fault analysis of the ROM, the probability id short-circuit faults to adjacent bus lines is larger that the probability of random short-circuit faults or open-circuit faults. Therefore, the adjacent asymmetric error code is needed.

We have to clarify the code conditions necessary for masking line faults economically and with better yield.

1. To minimize the number of transistors in the decoder of the bus line circuit shown in Fig. 5.3 the codes should have minimum weight.
2. To minimize the number of bus lines, the number of codewords should be maximum.
3. Consideration is needed about the fault cases of bridging faults in the bus lines that cause to adjacent asymmetric errors.

From the above conditions, we can compare the number of bus lines in ordinary bus line circuits with the number of bus lines in fault-tolerant bus line circuits having adjacent error masking codes with weight w in Table 5.4. It should be noted that there are cases, marked with * in the table, in which the number of bus lines in fault-tolerant bus line circuits is smaller than that in ordinary bus line circuits.

Number of circuits outputs M (k=1024)		16	32	64	128	256	512	1k	2k	4k
Number of bus line in ordinary bus line circuit		8	10	12	14	16	18	20	22	24
Number of bus lines	(w=2)	9	13	15	19	26				
in fault- tolerant	(w=3)	9	11	13	15					
bus line circuit	(w=4)	9	10	12	13*	15*	17*	19*	21*	23*

Table 5.4 Comparison of the number of bus lines

Also we can reduce a lot of transistors with AAEMC codes. For example, When M = 256, we need 256 * 8 transistors in ordinary bus line circuit, while 256*4 transistors in adjacent fault-tolerant bus line circuit when the weight is 4.

REFERENCES

1. W.W. Peterson and E. J. Weldon, Error Correcting Codes, Cambridge, MA: M.I.T. Press, 1972.

2. E. R. Berlekamp, Algebraic Coding Theory, New-York: McGraw-Hill, 1968

3. C. V. Freiman, "Optimal Error Detection Codes for Completely Asymmetric Binary Channels,", Infor. & Contr., vol 5, pp 64-71, 1962

4. J. M. Berger, "A note on Error Detecting Codes for Asymmetric Channels.", Infor. & Contr., vol 4, pp 68-73, Mar. 1961

5. B.Bose and D.J. Lin, "Systematic Unidirectional Error-Detecting Codes", IEEE Trans. on Computers, vol. C-34, pp. 1026-1032, Nov. 1985.

6. H. Dong, "Modified Berger Codes for Detection of Unidirectional Errors," IEEE Trans. on Computers, pp. 572-575, Jun 1984.

7. N. K. Jha and M. B. Vora, "A Systematic Code for Detecting t-unidirectional Errors.", 17th FTCS., pp. 96-10, 1987

8. B. Bose, "Burst Unidirectional Error-Detecting Codes," IEEE Trans. on Computers, pp. 350-353. Apr. 1986

9. B. Bose, "Burst Unidirectional Error-Correcting Codes," 19th FTCS pp. 350-353. Apr. 1989

10. B. Bose and T.R.N.Rao, "Unidirectional Error Codes for Shift-Register Memories", IEEE Trans. on Computers, pp 575-578, Jun. 1984

11. B. Bose and D. K. Pradhan, "Optimal Unidirectional Error Detecting/Correcting Codes," IEEE Trans. on Computer. pp. 564-568, Jun. 1982

12. D.J. Lin and B.Bose, "Systematic Unidirectional Error-Detecting Codes", IEEE Trans. on Computers, vol. C-34, pp. 1026-1032, Nov. 1989.

13. S.D. Constantin and T.R.N.Rao, "On the Theory of Binary Asymmetric Error Correcting Codes", Infor. & Contr. pp 20-26, Jan. 1979

14. M.A. Marouf and A.D. Friedman, "Design of Self-Checking Checkers for Berger Codes", Dig. 8th Annu. Int. Sym. Fault-Tolerant Comput., pp 179-184, 1978

15. Kazumitsu. Matsuzawa and Eiji. Fujiwara, "Masking Asymmetric Line Faults using Semi-distance Codes", 18th FTCS, pp. 354-359

16. R.L.Graham and N.J.A. Slone, "Lower Bounds for Constant Weight Codes", IEEE Trans. on Information Theory, pp 37-41, Jan. 1980

Chapter 6

Conclusion

6.1 Summary

In chapter 2, we proposed a new strategy to recognize the maximum subcube in an n-cube multiprocessor. This subcube recognition algorithm can be done in both serial and parallel and analyzed. This strategy will enhance the performance drastically so that our algorithm will outperform the buddy system by a factor ${}_nC_k$, the gray strategy by ${}_nC_k/2$ and Al-Dhelaan strategy by ${}_nC_k/(k(n-k)+1)$ in cube recognition. We present a very efficient processor allocation strategy which makes larger contiguous spaces for the new coming job than buddy, gray strategy and Al-Dhelaan strategy do. Furthermore, this new strategy is suitable for static as well as dynamic processors allocation and it results in a less fragmentation and higher fault tolerance. Also we describe an efficient procedure for task migration under the new strategy: 1) goal configuration under the new strategy 2) node-mapping between source and destination node 3) the shortest deadlock-free routing algorithm.

In chapter 3, we developed a new broadcasting algorithm in an N-cube multiprocessors using a binomial tree. This algorithm takes $\log_2(N)$ steps to broadcast all the processors. Our broadcasting algorithm is a procedure by which a processor can pass a message to all other processsors in the network non-redundantly: this message can either be information or control. We describe an optimal fault tolerant broadcasting algorithm when n-1 processors are faulty in Q_n. And we proved that this algorithm is optimal formally. This algorithm takes $\log_2(N)+1$ steps to broadcast the message to all non-faulty processors.

In chapter 4, a simple yet efficient algorithm to broadcast in a in a Cube-Connected Cycles Network containing faulty nodes/links was proposed. The algorithm is particularly useful in critical real-time systems that cannot tolerate the time overhead of identifying the faulty processors on-line. The algorithm delivers multiple copies of the broadcast message through disjoint paths to all the nodes in the system. The salient feature of the proposed algorithm is that the delivery of the multiple copies is transparent to the processes receiving the message and does not require the processes to know the identity of the faulty processors. The processes on nonfaulty nodes that receive the message identify the original message from the multiple copies using some scheme appropriate for the fault model used. The algorithm completes in $\lfloor s/2 \rfloor + (2s-1) + \lfloor s/2 \rfloor$ steps if each node can simultaneously use all of its outgoing links. But if each node cannot use more than one outgoing link at a time, then the algorithm requires 4s-2 steps.

In chapter 5, we developed the AAEMC using systematic methods and analyze those codewords when the weight is 2, 3 and 4 in the constant weight codes. We derived an equation to get those codes, especially proved the maximality of AAEMC and found an interesting recurrence relation between code length when weight is 2.

When these codes are used for short-circuit faults, they are capable of masking a single adjacent asymmetric error in bus lines in LSIs. This can be used in minimizing the number of transistors in the decoder of the bus line circuits, i.e., the codes have the minimum weight. Also the bus lines can be minimized.

6.2 Future Research

A few problems are generated from this thesis and are left open. For future research, we have the following problems to explore:

From Chapter 2:

- We will try to develop some processor allocation strategies for other interconnection networks.
- We will find a new approach for dynamic processor allocation in hypercube machine.

From Chapter 3 and 4:

- We will develop an efficient fault tolerant broadcasting algorithm in the incomplete hypercube.
- Better routing and Broadcasting Algorithm in Incomplete hypercube

From Chapter 5:

- We will try to find the adjacent asymmetric error masking codes when the weight is greater than 4 in constant code.

Bibliography

A. Al-Dhelaan and B. Bose, "A New strategy for Processor Allocation in an N-cube Multiprocessor", Phoenix Conference on Computer and Communication, Mar 1989. pp. 114-118.

A. Al-Dhelaan and B. Bose, "Efficient Fault Tolerant Broadcasting Algorithm for the Hypercube", Proc.The fourth Conf. on Hypercube Concurrent Comp. and Applications, Monterey, Mar 1989, pp. 123-128.

A. Al-Dhelaan and B. Bose, "Efficient Fault Tolerant Broadcasting Algorithm for the Cube-Connected Cycles Network", Proc. IEEE Pacific Rim Conference, May 1989, pp. 161-164.

J. R. Armstrong and F.G. Gray, "Fault Diagnosis in a Boolean n Cube Array of Microprocessors", IEEE Trans. on Computers Aug. 1981, pp. 587-590.

P.Banerjee, S.Y. Kuo, W. K. Fuchs, "Reconfigurable Cube-Connected-Cycles Architecture", Proc, of IEEE, 1986, pp. 286-291.

B. Becker and H.U. Simon, "How robust is the n-cube?", in Proc. 27th Ann. Symp. Foundations of Comp. Sci. Oct. 1986 pp. 283-291.

J. M. Berger, "A note on Error Detecting Codes for Asymmetric Channels.", Infor. & Contr., vol 4, pp 68-73, Mar. 1961

E. R. Berlekamp, Algebraic Coding Theory, New-York: McGraw-Hill, 1968

D. Bitton , D. DeWitt, D. Hsiao and J. Menon, "A taxonomy of Parallel Sorting", Computing Surveys, Vol. 16, Sep 1984, pp. 458-473

B.Bose and D.J. Lin, "Systematic Unidirectional Error-Detecting Codes", IEEE Trans. on Computers, vol. C-34, pp. 1026-1032, Nov. 1985.

B. Bose, "Burst Unidirectinal Error-Detecting Codes," IEEE Trans. on Computers, pp. 350-353. Apr. 1986

B. Bose, "Burst Unidirectinal Error-Correcting Codes," 19th FTCS pp. 350-353. Apr. 1989

B. Bose and T.R.N.Rao, "Unidirectional Error Codes for Shift-Register Memories", IEEE Trans. on Computers, pp 575-578, Jun. 1984

B. Bose and D. K. Pradhan, "Optimal Unidirectional Error Detecting/Correcting Codes," IEEE Trans. on Computer. pp. 564-568, Jun. 1982

M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms", SIAM J. Comput, Vol. 7, Aug. 1978, pp 298-319.

M. Chen and K.G. Shin, "Processor Allocation in an N-cube Multiprocessor Using Gray Codes", IEEE Trans. Computer, Dec. 1987 pp. 1396-1407.

M. Chen and K.G. Shin, "Task Migration in Hypercube Multiprocessor", Proc. 16th Annual Int'l Symp. on Computer Architecture. Jun 1989, pp. 105-111

M. Chen and K.G. Shin, "Adaptive Fault-Tolerant Routing in Hypercube Multicomputers" To appear in IEEE Trans. on Computers, 1989

R. M. Chamberlain, "Gray codes, Fast Fourier Transformations and Hypercubes", Parallel Computing, 6, 1988, pp. 458-473.

S.D. Constantin, T.R.N.Rao, "On the Theory of Binary Asymmetric Error Correcting Codes", Infor. & Contr. pp 20-26, Jan. 1979.

N. Deo, Graph Theory with applications to Engineering and Computer Science, Prentice-Hall, 1974.

H. Dong, "Modified Berger Codes for Detection of Unidirectional Errors," IEEE Trans. on Computers, pp. 572-575, Jun 1984.

C. V. Freiman, "Optimal Error Detection Codes for Completely Asymmetric Binary Channels,", Infor. & Contr., vol 5, pp. 64-71, 1962

R.L.Graham and N.J.A. Slone, "Lower Bounds for Constant Weight Codes", IEEE Trans. on Information Theory, pp 37-41, Jan. 1980

J. E. Jang, S. W. Choi and W. K. Cho, "A New Approach to Processor Allocation and Task Migration in an N-cube Multiprocessor", Proc. International Conference on Supercomputing, Nov, 1989. pp. 314-325.

J. E. Jang and W. K. Cho, "Maximulity of Subcube Recognition and Fault Tolerance in an N-cube Multiprocessor", Proc. 4th SIAM conference on Parallel Processing, Dec 1989.

J. E. Jang,"An Optimal Fault Tolerant Broadcasting Algorithm for a Hypercube Multicomputers", Proc.1990 ACM Computer Science Conference, Feb. 1990, pp. 96-102.

J. E. Jang, "Reliable Broadcasting Algorithm in an Cube-Connected Cycles Network", Proc. 9th International Pheonic Conference onComputers and Communications, Mar. 1990, pp. 3-9.

J. E. Jang, "Optimal Fault Tolerant Broadcasting Algorithm in an Cube-Connected Cycles Network", Proc. Int'l Conference on Databases, Parallel Architecures and their applications (PARBASE-1990), Mar, 1990. pp. 206-215.
(To appear as a chapter in a book published by IEEE)

J. E. Jang and B. Bose, "Efficient Broadcasting Algorithm for an Incomplete Cube-Connected Cycles Network", To appear in the Proc. 5th Distributed Memory Computing Conference, Apr, 1990.

J. E. Jang and B. Bose, "A New Approach to Fault Tolerant Broadcasting Algorithm for an Cube-Connected Cycles Network" To appear in the Proc. 5th Distributed Memory Computing Conference, Apr, 1990.

J. E. Jang and B. Bose, "Masking Adjacent Asymmetric Line Faults", To be published.

N. K. Jha and M. B. Vora, "A Systematic Code for Detecting t-unidirectional Errors", 17th FTCS., pp. 96-10, 1987

H. P. Kattesff, "Incomplete hypercubes", IEEE Trans. Computer, May 1988, pp. 604-608.

Dongseung Kim, "Supercube: A Generalized Hypercube with Shared and Private Memories Using Multiple Spanning Buses", The 4th Conf. on Hypercube Concurrent Computers and Applications, Mar. 1989.

K. Matsuzawa and Eiji. Fujiwara, "Masking Asymmetric Line Faults using Semi-distance Codes", 18th FTCS, pp. 354-359, Jun. 1988.

L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," ACM Trans. Programming language System, pp. 382-401, Jul. 1982.

D.J. Lin and B.Bose, "Systematic Unidirectional Error-Detecting Codes", IEEE Trans. on Computers, vol. C-34, pp. 1026-1032, Nov. 1989.

M. Livingston andQ. F. Stout, "Fault Tolerance of Allocation Schemes in Massively Parallel COmputers", Proc. 2nd Symp. Frontiers of Massively Parallel Computation, pp 491-494. Oct. 1988.

M. Livingston and Q. F. Stout, "Parallel Allocation Algorithms for Hypercubes and Meshes", Proc. 4th Conf. on Hypercube Concurrent Computers and Applications, Monterey, CA, Mar 1989.

M.A. Marouf and A.D. Friedman, "Design of Self-Checking Checkers for Berger Codes", Dig. 8th Annu. Int. Sym. Fault-Tolerant Comput., pp 179-184, Mar 1978.

NCUBE Corp, NCUBE/10: An Overview, Beverton, OR, Nov 1985.

R. Negrini and M. G. Sami, Fault Tolerance Through Reconfiguration in VLSI and WSI Arrays, MIT Press, 1989.

M. Pease, "The Indirect Binary n-Cube Microprocessor Array," IEEE Trans. on Computers May 1977, pp. 458-473.

D. K. Pradahan, Fault-Tolerant Computing: Theory and Techniques, Prentice-Hall, 1986.

W.W. Peterson ans E. J. Weldon, Error Correcting Codes, Cambridge, MA: M.I.T. Press, 1972.

F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles, A Versatile Network for Parallel Computation," Communication of ACM, pp. 30-39, May 1981.

P. Ramanathan and K.G. Shin, "Reliable Broadcasting in Hypercube Multicomputers", IEEE Trans. on Comp. Dec 1988, pp. 1654-1657.

Y. Saad and M.H. Schultz, "Topological Properties of Hypercubes", IEEE Trans. on Computers, Jul 1988, pp 867-872.

C. L. Seitz, "The Cosmic Cube," Commun. Ass. Comput. Mach. Vol. 28, Jan 1985, pp. 22-33.

T. K. Srikanth and S. Toueg, "Optimal clock synchronization," J. ACM pp.626-645, Jul.1987.

H. S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Computers, pp. 153-161, Feb. 1971.

M. Sultan and Rami Melhem, "Fault Tolerance and Reliable Routing in Augmented Hyercube Architecture", Phoenix Conference on Computer and Communication, Mar, 1989. pp. 19-23.

H. Sullivan and T. R. Baskow, "A large scale homogeneous, fully distributed parallel machine," Proc. Fourth Symp. Comp. Architecture, Mar. 1977, pp. 105-117.

J. D. Ullman, Computational Aspects of VLSI, Computer Science Press, 1984.

Appendices

Appendix A

AAEMC when n=12 & w=3

1	000000010101	51	000001001010
2	000001000101	52	000100001010
3	000100000101	53	010000001010
4	010000000101	54	000100100010
5	000001010001	55	010000100010
6	000100010001	56	010010000010
7	010000010001	57	000100101000
8	000101000001	58	010000101000
9	010001000001	59	010010001000
10	010100000001	60	010010100000
11	000001010100		
12	000100010100		
13	010000010100		
14	000101000100		
15	010001000100		
16	010100000100		
17	000101010000		
18	010001010000		
19	010100010000		
20	010101000000		
21	000000101001		
22	000010001001		
23	001000001001		
24	100000001001		
25	000010100001		
26	001000100001		
27	100000100001		
28	001010000001		
29	100010000001		
30	101000000001		
32	001000100100		
33	100000100100		
34	001010000100		
35	100010000100		
36	101000000100		
37	001010010000		
38	100010010000		
39	101000010000		
40	101001000000		
41	000010010010		
42	001000010010		
43	100000010010		
44	001001000010		
45	100001000010		
46	100100000010		
47	001001001000		
48	100001001000		
49	100100001000		
50	100100100000		

Appendix B

AAEMC when n=12 & w=4

1	000010101010	46	000000001111
2	001000101010	47	000000111100
3	100000101010	48	000011110000
4	001010001010	49	001111000000
5	100010001010	50	111100000000
6	101000001010	51	000000110011
7	001010100010	52	000011000011
8	100010100010	53	001100000011
9	101000100010	54	110000000011
10	101010000010	55	000011001100
11	001010101000	56	001100001100
12	100010101000	57	110000001100
13	101000101000	58	001100110000
14	101010001000	59	110000110000
15	101010100000	60	110011000000
16	000001010101	61	001001001001
17	000100010101	62	100001001001
18	010000010101	63	100100001001
19	000101000101	64	100100100001
20	010001000101	65	100100100100
21	010100000101	66	000001100110
22	000101010001	67	000110000110
23	010001010001	68	011000000110
24	010100010001	69	000110011000
25	010101000001	70	011000011000
26	000101010100	71	011001100000
27	010001010100	72	010010010010
28	010100010100	73	000101001010
29	010101000100	74	010001001010
30	010101010000	75	010100001010
31	000010100101	76	010100100010
32	001000100101	77	010100101000
33	100000100101	78	000100101001
34	001010000101	79	010000101001
35	100010000101	80	010010001001
36	101000000101	81	010010100001
37	001010010001	82	010010100100
38	100010010001	83	001001010010
39	101000010001	84	100001010010
40	101001000001	85	100100010010
41	001010010100	86	100101000010
42	100010010100	87	100101001000
43	101000010100		
44	101001000100		
45	101001010000		