Nanyang Technological University

SCE 02-434

Analysis of Fuzzy-Neuro Network Communications

Zhang Xinhua

School of Computer Engineering
2003

Nanyang Technological University

SCE 02-434

Analysis of Fuzzy-Neuro Network Communications

Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor of Computer Engineering

by

Zhang Xinhua

School of Computer Engineering
2003

ABSTRACT

Highly parallel computers are playing a central role in high-performance computing. In addition to network topology, reliable and efficient message routing is becoming increasingly critical with the rapidly growing system scale. Although many fault-tolerant routing strategies have been proposed for various specific networks, there lacks a general algorithm that applies well to a wide variety of topologies.

Fuzzy Neural Networks (FNNs) are a group of hybrid systems that incorporate fuzzy logic into Artificial Neural Network (ANN) architectures. The fuzzy characteristic provides interpretable human-like IF-THEN reasoning rules while ANN supplies the learning ability to the traditional fuzzy systems by deriving membership function and/or rule base automatically. These traits make FNN a promising tool for designing efficient general-purpose routers and the feasibility and difficulties are explored in the project.

On the other hand, research in traditional routing algorithm is still not complete enough to encompass all interconnection networks. Due to sparse connectivity and low node availability, there is no existing fault-tolerant routing strategy for node/link diluted hypercubic networks. Among these networks, Gaussian Cubes (*GC*s) use a common parameter to link the interconnection density and algorithmic efficiency. The variation of it can scale routing performance according to traffic loads without changing the routing algorithm. Fibonacci-class Cubes use fewer links than the corresponding binary hypercube, with the scale increasing slower, allowing more choices of network size.

To make these types of networks with such desirable properties more fault-tolerant, the project investigates the approaches of divide-and-conquer and fault classification so as to tolerate more faults than node availability. To facilitate our discussion, a new type of interconnection network named Exchanged Hypercube (*EH*) is proposed. It reduces the number of links to only 1/*n* of binary hypercubes with the same number of nodes (*n* is the network's dimension) with little lose of structural advantage. New auxiliary topologies are also proposed for illustrating *EH*'s desirable emulation and communication properties.

Finally, as a new prototype for efficient simulation of incomplete networks, a software simulator is built and the results about the performance of our algorithms are shown to be reasonable. FPGA implementation is also completed to demonstrate the feasibility of physical manufacture.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the following people.

A/P Peter, K K, Loh

for directing the research and development of this project

A/P Quek Hiok Chai

for guiding the choice and use of Fuzzy Neural Network

Mr. Tan Swee Huat

for providing hardware and software technical support in the Intelligent Systems laboratory.

Mr. Tung W. L., Mr. Ang K. K., and Mr. Ting C. W.

for sharing programming experiences and problem solving techniques.

Administrators of *Nanyang Technological University*

and *Shanghai Jiao Tong University*

for providing this exchange program opportunity

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

1.1 Background

With the growing demand for high-performance computing power in more and more software applications, highly parallel computers have attracted increasing interest in recent years. Multicomputers, which are based on message–passing for interprocessor communications, can scale up to hundreds of thousands of processors, providing the capability of massive parallel processing. Hypercube Multicomputers [37], considered one of the most extensively studied topology due to their structural regularity, easy construction and high potential for parallel execution of various algorithms, have been used in several experimental and commercial machines including NCUBE-2 [35] and Intel iPSC [36]. Many variations of the hypercube topology have been proposed to improve certain parameters, such as diameter, node degree, emulation and communication efficiency, etc [1][12-15][39- 43][52][53].

Unicasting, the focus of this project, is a one-to-one communication between a source node and a destination node. Unicasting in fault-free hypercubes and its variations have been extensively studied in [44-47]. As in [54], when the scale of parallel computer systems grows, the probability of component failure (processors and/or links) increases. Reliable and efficient message routing is thus becoming more and more critical, requiring the routing algorithm to be capable of tolerating high probability of component failures. There have been a number of fault-tolerant unicasting schemes proposed [6][19][48-51].

In designing fault tolerant communication strategies in large networks, there are many issues deserving special attention. Firstly, besides having fault-tolerant mechanism, an adaptive routing algorithm, which makes more efficient use of network bandwidth and

provides resilience to failure [54], is also necessary for routing in faulty communication networks.

Secondly, besides reliability, efficiency is also an important consideration. As in [55], the fault-tolerant communication mechanism should not degrade the performance gained by parallelism and at the same time guarantee delivery of messages to their destinations in the presence of faulty network components. It also should not incur message routing overheads in a fault-free network.

Thirdly, scalable and space efficient schemes [33][34] should be used. A fault tolerant routing algorithm should not require excessive space to store status information in the network. It should maintain or update status information efficiently so as to ensure high performance under fault-free condition, be free from deadlock and livelock, and guarantee specified levels of reliability and efficiency in its performance.

1.2 Purpose of Project

A large variety of interconnection network topologies have been proposed, each with its possible unique fault-tolerant routing algorithm. However, there is no general algorithm that can apply to all types of topologies. In the exploration of a general-purpose router, the technology of Fuzzy Neural Network (FNN) is looming as a promising tool. FNN is equipped with outstanding learning and clustering capability that have found successful applications in many areas. It can also provide human-like interpretable rules that overcome the problem of black-box in ordinary artificial neural networks. In this project, efforts are taken to evaluate the potential and feasibility of fuzzy logic routing, to investigate the possibility of unifying the membership functions and rules learned from

different topologies of networks. In the best case, framework of software tools is to be studied so as to measure and compare the communications performance of fuzzy logic routing against existing fault-tolerant routing strategies.

On the other hand, even in the realm of classic fault-tolerant routing strategies, there is a void for link/node diluted hypercubic networks. The intrinsic problem lies in the sparse connectivity that brings about susceptibility to the occurrence of faults. Attracted by their other desirable properties, we attempt to design fault-tolerant routing algorithms to make *Gaussian Cubes* and Fibonacci-class Cubes more fault-tolerant topologies. Later on, these algorithms will be implemented by software simulator and FPGA, so that their performance can be benchmarked and the feasibility of physical manufacture can be assessed. If fuzzy routing is proved a practicable approach, the simulation result of the performance of both FNN and classic methods can be compared as well.

1.3 Objectives

In order to fulfill the purpose of the project, the following objective are defined:

- To explore fuzzy neural network applied in network communications.
- To design a fault-tolerant routing algorithm for Gaussian cube.
- To design a fault-tolerant routing algorithm for Fibonacci-class Cubes.
- To propose a new interconnection topology: Exchanged Hypercube.
- To write software simulation tools for implementation and benchmark.
- To implement the routing algorithm of Fibonacci-class Cubes and fuzzy routing strategy on FPGA with Handel-C.

1.4 Overview of Report Organization

The report is organized into 10 chapters.

In Chapter 2, the preliminaries of fault-tolerant interconnection network routing are presented. Basic terms are defined and the requirements for the routing algorithm in question are also given.

In Chapter 3, the fundamentals of Fuzzy Neural Network (FNN) are reviewed. The possibility and difficulty in applying FNN to the interconnection network routing are explored.

In Chapter 4, a new fault-tolerant routing strategy is presented for Fibonacci-class Cube. We also designed a generic approach for cycle-free routing.

In Chapter 5, a new interconnection topology named 'Exchanged Hypercube' is proposed based on link dilution from binary hypercube. Its structural features and emulation, communication properties are discussed.

In Chapter 6, a new fault-tolerant routing strategy for *Gaussian Cube* is described. The major merits and general significance are emphasized.

In Chapter 7, a software simulator is constructed to test the performance of the two fault-tolerant routing strategies presented in Chapter 4 and 6. The architecture and many features of the simulator are discussed.

In Chapter 8, the simulation results are illustrated. Detailed analysis is also carried out to investigate the result, including comparisons between different topologies and some seemingly irregularities.

In Chapter 9, we discuss the FPGA hardware implementation of the routing strategy proposed in Chapter 4, as well as routing with fuzzy neural network. Many suggestions are listed for future development.

Chapter 10 concludes the report with discussion of findings in this project and provides a recommendation for future work.

Chapter 2 Preliminaries

2.1 Communications network

For many parallel applications, the interconnection network determines overall performance [58]. The most commonly used topology is binary hypercube.

An *n*-dimensional hypercube can be modeled as a graph $G(V_n, E_n)$, with the node set V_n and edge set E_n, where $|V_n| = 2^n$, $|E_n| = n2^{n-1}$. Each node represents a processor and its memory. Each edge represents a communication link between a pair of processors. The 2^n nodes are distinctly addressed by *n*-bit binary numbers, with values from 0 to $2^n - 1$. Each node has links at *n* dimensions, ranging from 0 (lowest dimension) to $n-1$ (highest dimension), connecting to *n* neighbors. An edge connecting nodes *u* and *v* is said to be at dimension *j* or to be the j^{th} dimensional edge if their binary addresses *u* and *v* differ at bit position *j* only. Figure 2.1 shows a 4-dimensional binary hypercube.

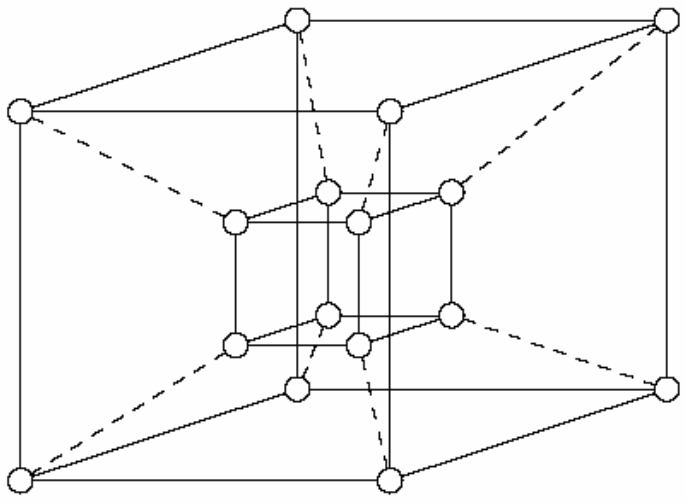


Figure 2.1 4-dimensional binary hypercube (16 PEs)

The *length* of a path is equal to the number of links contained in the path. The *distance* between two nodes u_0 and u_d is equal to the hamming distance between their binary address, denoted by $H(u_0, u_d)$. A path between u_0 and u_d is called an *optimal path* if its length is equal to the distance between the two nodes. A *shortest path* is a path of minimal length among all possible paths between the two nodes when constrained by the presence of faulty components. A shortest path may or may not be an optimal one.

2.1.1 Switching Techniques

Switching refers to the means of transferring a packet from the input channel to the output channel. Four switching techniques, *store-forward*, *circuit switching*, *wormhole routing* and *virtual cut-through*, are discussed here. The choice of switching technique has a great bearing on the network performance, especially on deadlock and livelock freeness.

In store-forward, the received packet is stored in a buffer and then forwarded to the selected neighboring node based on the routing decision made by the routing algorithm. After the packet is forwarded, it waits for an acknowledgement from the receiver. The whole process of storing and forwarding a packet is referred to as a hop.

In circuit switching, a physical connection path between the source and destination nodes must be established. After the path is established, the packet is allowed to move through the path without any buffering. During the transmission of a packet along this path, the connection is not switched and thus no other packets are allowed to move along this path. This physical connection path is torn down after the packet has reached its destination.

In wormhole routing [59], the packet to be routed is divided into chunks called flits. These flits spread over the entire path between the source and destination nodes where

each node along the path has a queue for each of its adjacent links to hold the flit. If there is space in the next node or when flits are consumed by the destination node, the head will move and the entire packet can move by moving to the free space created.

In virtual cut-through, if there is free space in the next node, the received packet is forwarded without buffering. Otherwise, the received packet is stored in a queue that can hold the entire packet.

2.1.2 Flow Control

Flow control refers to the allocation of channels and buffers to a packet as it moves along the path between the source and destination nodes. An appropriate flow control policy should be used for different switching techniques. For store-forward and virtual cut-through, flow control policy is applied on packet, whereas for wormhole routing, each flit will have a unique flow control. The flow control policy determines whether packet will be discarded, buffered, blocked or rerouted through another channel.

2.1.3 Routing

In multiple hop topologies, routing determines the path by which a message packet generated by an arbitrary source is to traverse in order to reach its destination. Routing can be classified into source routing and distributed routing.

In source routing, the entire path for a message packet to traverse is determined by the source node based on the current network condition. Once the packet leaves the source node, it will follow the selected path till it reaches its destination. In distributed routing, when a node receives a packet, it will determine whether the packet has reached destination. If packet reaches destination, this packet is delivered to the local processor.

Otherwise, the routing algorithm is used to determine which neighboring node to forward the packet to.

A disadvantage of using source routing is larger packets size where routing information is included in every packet. In distributed routing, the routing algorithm will normally produce a path with lower network latency. Thus, distributed routing is the major focus of this project.

2.2 Fault-tolerant routing

In the presence of faulty components in the interconnection network, it is desired that alternative paths can be found and used to bypass the faults. The following concepts are important in fault-tolerant routing.

2.2.1 Types of Faults

Component faults in a communication network can be either node faults or link faults or both. A node faults will incur the breakdown of all links incident to that node.

2.2.2 Types of links / dimensions

Let the current node be u and destination be d. The relative address r is defined as $r = u \oplus d$, where $\oplus$ denotes the bitwise exclusive OR (XOR). All the dimensions whose corresponding bit in r equals 1 are called preferred dimensions, while all the rest dimensions whose corresponding bit in r equals 0 are called spare dimensions. A faulty dimension refers to either a faulty neighboring node or a faulty link at that dimension.

2.2.3 Adaptiveness

Routing algorithm can be either classified as *static* (deterministic) or *adaptive*. In static

or deterministic routing algorithm, a fixed path is used to send messages between a given pair of source and destination nodes. At the source node, the selected path is determined based on the destination node and the current network conditions. As for adaptive routing algorithm, alternative paths between the source and destination nodes are used to route messages. Each node can only determine the next node to forward a message based on the local or global information that it contains.

In the context of minimal routing, dynamic adaptive routing algorithm can dynamically adjust its adaptivity based on fault distribution in the neighborhood [54]. This dynamic adaptivity can be further categorized as *fully adaptive*, *partial adaptive*, *one-adaptive* and *zero-adaptive* (also called *infeasible*). Fully adaptive algorithm can use all possible minimal paths between the source and destination node. As for partially adaptive algorithm, a subset of available minimal paths between the source and destination nodes is used. Only a single minimal path is available for one-adaptive algorithm. For zero-adaptive, there is no available minimal path at an intermediate node.

Adaptive algorithm can be characterized as progressive, backtracking, profitable and derouting (or misrouting). Progressive algorithm will wait, deroute or abort if no preferred link is available at an intermediate node. Backtracking refers to messages using the input link to route when they are at deadend nodes. In order for a message to move closer to the destination, preferred links are used. In contrast, spare links move a message farther away from the destination. Profitable algorithms only consider profitable links. Derouting or misrouting algorithm can use both preferred and spare links.

2.2.4 Deadlock

A deadlock maybe defined as a cyclic dependency of ungranted packet requests for buffer or channel resources [57]. It refers to the situation where a packet is blocked forever in the network. Deadlock occurs when a packet is holding some resources while

requesting for other resources that other packets are holding and these other packets are requesting for those resources that are held by this packet which results in a circular wait. An example where deadlock occurs is shown in Figure 2.2.

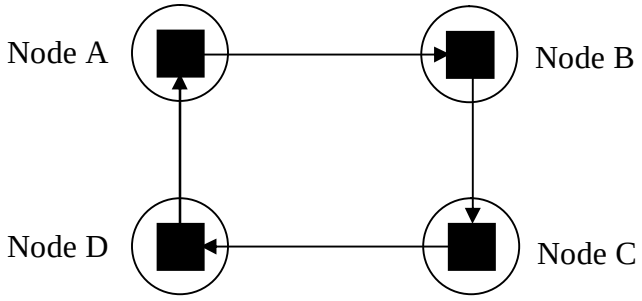


Figure 2.2: Four packets in circular waiting using store-forward

In Figure 2.2, there are four packets each holding a packet buffer represented by black square and four nodes represented by circles. Each node has a packet buffer. The packet in node A is requesting buffer in node B. Packet in node B is requesting buffer from node C. Packet in node C is requesting buffer from node D and packet in node D is requesting buffer from node A. As a result, a circular wait is formed.

2.2.5 Livelock

Livelock refers to the situation where a packet is circulating in the network without reaching the destination. Livelock usually occurs when misrouting is allowed in the routing algorithm in order to tolerate faults. An example where livelock occurs is shown in Figure 2.3.

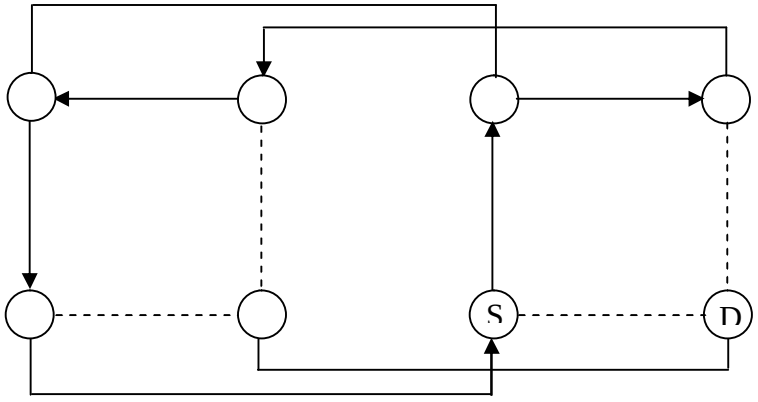


Figure 2.3 Livelock with four link faults

There are four link faults represented by dashed lines. Source and destination nodes are represented by a circle with 'S' and 'D', respectively. The arrows represent the path by which a packet generated by the source node traverses. These arrows form a cycle which means that the packet is circulating in the network without reaching its destination. Hence, livelock arises.

2.2.6 Types of information for routing decision

An adaptive algorithm requires either *local* or *global* information to make routing decision. However, there is *limited global* information based approach which is a compromise between local information based and global information based approaches.

In local information based model [6], each node exchanges information with its adjacent neighbors and it only knows the status of its neighbors. This model can only achieve local optimization and is heuristic in nature. However, it can be proved for some special network topologies that routing strategies based on local information is enough for tolerating faults with satisfactory performance.

As for global information based model, such as the Shortest Path Routing in [20], each node exchanges information with its adjacent neighbors as similar to local information based model. But this information is propagated throughout the entire network. Hence, each node knows the status of all the nodes and this model can normally achieve optimal or suboptimal result. The problem here is the huge task of gathering and exchanging global information, which is usually in large size.

Limited global information based approach [54] requires a relatively simple process to collect and maintain fault information in the neighborhood (such information is called limited global information) and is more cost effective than local or global information based approaches.

2.2.7 Types of Communication

Three types of communication are generally discussed: *unicasting, multicasting and broadcasting*. Unicasting is a one-to-one communication between two nodes; one is called source node and the other the destination node. Multicasting and broadcasting involve communication between several nodes, but the difference is that multicasting is a one-to-many communication that involves only one source node and several destination nodes whereas broadcasting is a one-to-all communication that involve one source node and all other nodes in a network.

2.2.8 Optimality

A routing algorithm can be categorized as optimal or suboptimal or both based on the path that a message traverses from source to reach its destination. In optimal or minimal routing, a message moves along a minimal path (also called a *Hamming* distance path) to its destination node. This means that each link along the minimal path is a preferred link. As for suboptimal or nonminimal routing, a path (where a message traverses) with the length more than the *Hamming* distance between the source and destination is generated. This means that nonpreferred or spare links are used for deroute or misroute when faulty component is encountered.

Chapter 3: Fuzzy Neural Network for Routing

3.1 Overview of Fuzzy Neural Network

Fuzzy Neural Networks (FNNs) are a group of hybrid systems that incorporate fuzzy logic into Artificial Neural Network (ANN) architectures. The fuzzy characteristic overcomes the problem of black box in ANN by providing interpretable human-like IF-THEN reasoning rules while ANN supplies the learning ability to the traditional fuzzy systems by deriving fuzzy rule base and/or membership function automatically. Such hybrid systems can be deployed in clustering, time series or stock market prediction, as well as automated control of large, complex systems.

The main advantage of a fuzzy logic is its ability to model a problem domain using a linguistic model instead of complex mathematical models. Zadeh proposed fuzzy logic as a new method to manage vagueness and uncertainty [60-63]. When modeling vagueness, fuzzy predicates without well-defined boundaries concerning the set of objects may be applied. The rationale for using fuzzy logic is that the denotations of vague predicates are fuzzy sets rather than probability distributions. In many situations, vagueness and uncertainty are simultaneously presented since any precise or imprecise fact may be uncertain as well. Fuzzy set and possibility theories provide a unified framework to deal with vagueness and uncertainty.

However, the fuzzy logic itself does not have learning ability, i.e. the parameters of fuzzy rules and membership functions can not be self-adjusted, but must be set by expert knowledge. As such, fuzzy neural networks are adopted due to their recognized learning ability. Generally, FNNs perform cluster analysis on each dimension of the inputs and

outputs of training data to determine the fuzzy sets and subsequently derive the fuzzy rules by connecting the input and output fuzzy sets.

In this chapter, we explore the possibility of applying Fuzzy Neural Network (FNN) to interconnection network routing, though the result is pessimistic.

3.2 Fuzzy Inference System

A fuzzy inference system is composed of following components:

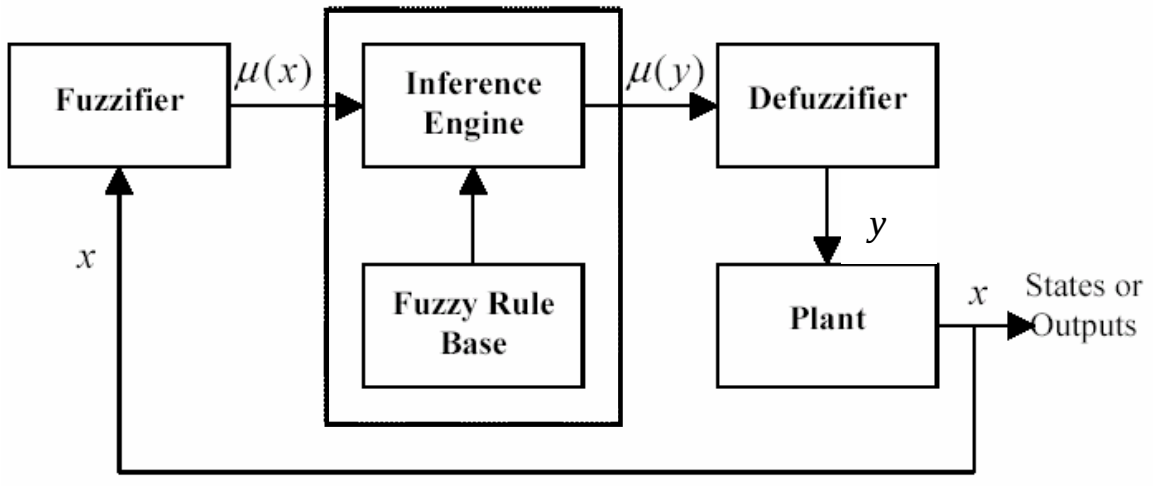


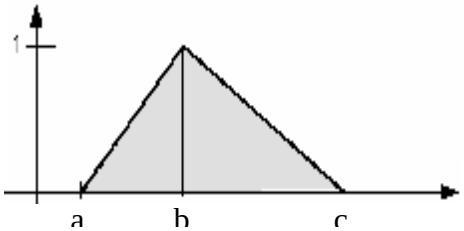
Figure 3.1 Fuzzy Inference System

The specification of fuzzy inference system encompasses the five blocks in Fig. 3.2. The following components are important:

3.2.1 Fuzzifier

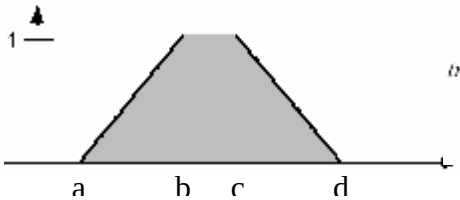
This part focus on the shape of membership function: Gaussian, Trapezoidal, Triangular, Bell-shape, etc).

Triangular:



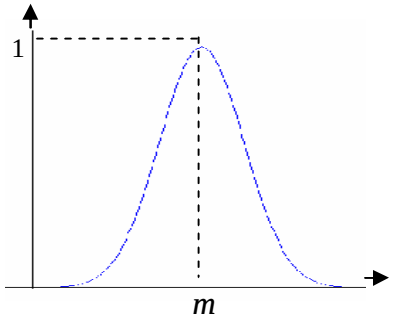
$$triangle(x:a,b,c)=\begin{cases}0 & x<a\\ (x-a)/(b-a) & a\le x\le b\\ (c-x)/(c-b) & b\le x\le c\\ 0 & x>c\end{cases}$$

Trapezoidal:



$$trapezoid(x:a,b,c,d)=\begin{cases}0 & x<a\\ (x-a)/(b-a) & a\le x<b\\ 1 & b\le x<c\\ (d-x)/(d-c) & c\le x<d\\ 0 & x\ge d\end{cases}$$

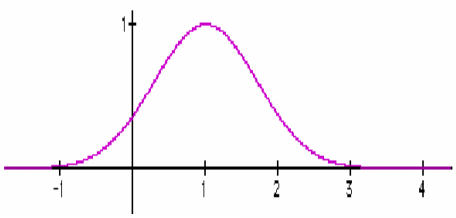
Gaussian:



$$gaussian(x:m,\sigma)=\exp\left(\frac{(x-m)^2}{-\sigma^2}\right)$$

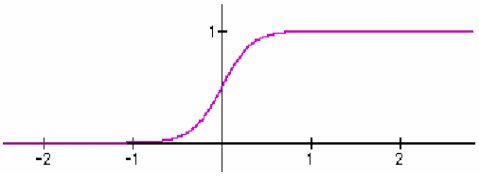
Other less frequently used functions include:

Bell-shaped Membership Function



$$bell(x:a,b,c)=\frac{1}{1+\left|\frac{x-c}{a}\right|^{2b}}$$

Sigmoidal Membership Function



$$sigm(x:a,c)=\frac{1}{1+e^{-a(x-c)}}$$

Still less frequently used shapes are S membership function, π membership function.

The simplest forms of membership function are trapezoid and triangle. They can provide high speed inference and fairly good accuracy. The two slopes belonging to [a, b] and [c, d] makes fuzzy logic different from classic two-value logic. But they are not ideal if high accuracy is desired. In such cases, Gaussian membership function is preferred because of its soft shape and long 'tail', which is different from the hard cut-off in trapezoid and triangle.

3.2.2 Fuzzy rule-based models for function approximation

How the rules are represented is very important for the compactness and effectiveness of the fuzzy system. There are three types of fuzzy rule-based models for function approximation: (a) the Mamdani model [23], (b) the Takagi-Sugeno-Kang (TSK) model [24][25][26], and (c) Kosko's Standard Additive Model (SAM) [27].

i) Mamdani model is one of the most widely used fuzzy models in practice, which consists of the following linguistic rules that describe a mapping from $U_1 \times U_2 \times \cdots \times U_r$ to W.

$$R_i: \ IF\ x_1\ is\ A_{i1}\ and\ \ldots\ and\ x_r\ is\ A_{ir}\ THEN\ y\ is\ C_i$$

where,

$X_j (j = 1,2, \ldots, r)$	input variables
y	output variable
A_{ij}	fuzzy sets for x_j
C_i	fuzzy sets for y.

The contribution of rule R_i to a Mamdani model's output is a fuzzy set whose membership function is computed by

$$m_{C_i'}(y) = (a_{i1} \wedge a_{i2} \wedge ... \wedge a_{in}) \wedge m_{C_i}(y)$$

where $$a_{ij} = \sup_{x_j}(m_{A_j'}(x_j) \wedge m_{A_{ij}}(x_j))$$

α_i is the matching degree of rule R_i

α_{ij} is the matching degree between x_j and R_i's condition about x_j

The final output of the model is the aggregation of outputs from all rules using the max operator:

$$m_C(y) = \max\{m_{C_1'}(y), m_{C_2'}(y), ..., m_{C_L'}(y)\}$$

ii) The Takagi-Sugeno-Kang (TSK) model was introduced in 1984. The main motivation of this model is to reduce the number of rules required by Mamdani model, especially for high-dimensional problems. It consists of rules in the form of:

$$R_i: \ IF\ x_1\ is\ A_{i1}\ and\ ...\ and\ x_r\ is\ A_{ir}$$

$$THEN\ y = f_i(x_1, x_2, ..., x_r) = b_{i0} + b_{i1}x_1 + ... + b_{ir}x_r$$

where

f_i is the linear model

$b_{ij}(j = 0,1,...,r)$ are real-valued parameters

The total output of the model is given as

$$y = \frac{\sum_{i=1}^{L} a_i f_i(x_1, x_2, ..., x_r)}{\sum_{i=1}^{L} a_i} = \frac{\sum_{i=1}^{L} a_i (b_{i0} + b_{i1}x_1 + ... + b_{ir}x_r)}{\sum_{i=1}^{L} a_i}$$

The inputs to a TSK model are crisp (nonfuzzy) numbers. Therefore, the degree of input $x_1 = a_1, x_2 = a_2, ..., x_r = a_r$ that matches the i^{th} rule is typically computed using the min operator:

$$a_i = \min\{m_{A_{i1}}(a_1), m_{A_{i2}}(a_2), ..., m_{A_{ir}}(a_r)\}.$$

TSK seems to be more effective (as in ANFIS) in the use of the number of rules in a fuzzy rule-based system as compared to CRI (as in POPFNN and GenSoFNN). But CRI inference is more intuitive and readable.

iii) The Standard Additive Model (SAM) was introduced by B. Kosko in 1996. The structure of fuzzy rules in SAM is identical to that of the Mamdani model. The rules is in the form of

$$IF\ x\ is\ A_i\ and\ y\ is\ B_i\ THEN\ z\ is\ C_i$$

Given crisp inputs $x = x_0$, $y = y_0$, the output of the model is

$$z = Centroid\ (\ \sum_i m_{A_i}(x_0) \times m_{B_i}(y_0) \times m_{C_i}(z)\ )$$

3.2.3 Definition of operators on fuzzy sets including: union, intersection, and complement.

There are multiple choices for the fuzzy conjunction and fuzzy disjunction operators. The choice of a fuzzy conjunctions operator determines the choice of the fuzzy disjunction, and vice versa. This is due to the principle of duality between the two operators. A *fuzzy conjunction* operator, denoted as $t(x,y)$ and *fuzzy disjunction* operator, denoted as $s(x,y)$, form a dual pair if they satisfy the following condition:

$1 - t(x, y) = s(1 - x, 1 - y)$, so as to ensure $\overline{A \cap B} = \overline{A} \cup \overline{B}$.

Here, the set of candidate fuzzy conjunction operators called *triangular norms* or *t-norms* is defined as a mapping T: $[0, 1] \times [0, 1] \rightarrow [0, 1]$ which is symmetric, associative, non-

decreasing in each argument and $T(a,\ 1) = a$, for all $a \in [0,\ 1]$. In other words, any t-norm T satisfies the properties:

$T(x, y) = T(y, x)$	symmetricity
$T(x, T(y, z)) = T(T(x, y), z)$	associativity
$T(x, y) \le T(x',\ y')$ *if* $x \le x'$ and $y \le y'$	monotonicity
$T(x,\ 1) = x$, $\forall x \in [0,\ 1]$	one identity

Basic *t*-norms include the following:

minimum	$MIN(a,\ b) = \min\{a,\ b\}$
Lukasiewicz	$LAND(a,\ b) = \max\{a + b - 1,\ 0\}$
Probabilistic	$PAND(a,\ b) = ab$
week	$WEEK(a,b) = \begin{cases} \min\{a,\ b\} & \text{if } \max\{a,\ b\} = 1 \\ 0 & \text{otherwise} \end{cases}$
Hamacher	$HAND_g(a,\ b) = \dfrac{ab}{g + (1-g)(a + b - ab)},\quad g \ge 0$
Dubois and Prade	$DAND_a(a,\ b) = \dfrac{ab}{\max\{a,b,a\}} \qquad a \in (0,\ 1)$
Yager	$YAND_P(a,b) = 1 - \min\{1, [(1-a)^p + (1-b)^p]^{1/p}\},\ p > 0$

Likewise, we can define *t*-conorm. The only difference between *t*-norm and *t*-conorm is that in *t*-conorm *S*, $S(a,\ 0) = a$, for all $a \in [0,\ 1]$. Basic *t*-conorm include the following:

maximum	$MAX(a,\ b) = \max\{a,\ b\}$
Lukasiewicz	$LOR(a,\ b) = \min\{a + b,\ 1\}$
Probabilistic	$POR(a,\ b) = a + b - ab$
strong	$STRONG(a,b) = \begin{cases} \max\{a,\ b\} & \text{if } \min\{a,\ b\} = 0 \\ 1 & \text{otherwise} \end{cases}$
Hamacher	$HOR_g(a,\ b) = \dfrac{a + b - (2-g)ab}{1 - (1-g)ab},\qquad g \ge 0$
Yager	$YOR_p(a,b) = \min\{1,\ [a^p + b^p]^{1/p}\},\qquad p > 0$

3.2.4 Definition of fuzzy inference schemes.

The operations of fuzzy neural network need to be clearly defined and mapped to formal fuzzy inference schemes. There are several such schemes such as Compositional Rule of Inference (CRI) [30], Approximate Analogous Reasoning Schema (AARS) [28] or the Ttruth Value Restriction (TVR) [29]. The most commonly used is CRI which works as follows.

Knowledge: If x is A then y is B

Fact: x is A'

Conclusion: y is B'

Here, $B' = A' \circ R$. $B'(v) = \sup_{u \in U} T\{A'(u), R(u,v)\}$, $v \in V$. There are a number of definitions of R.

Zadeh:

min-max rule:

$$R_m = (A \times B) \bigcup (\neg A \times V) = \int_{U \times V} (m_A(u) \wedge m_B(v)) \vee (1 - m_A(u)) / (u,v)$$

$$B'_m = A' \circ R_m = A' \circ [(A \times B) \bigcup (\neg A \times V)]$$

$$m_{B'_m}(v) = \bigvee_{u \in U} \{m_{A'}(u) \wedge [(m_A(u) \wedge m_B(v)) \vee (1 - m_A(u))]\}$$

arithmetic rule:

$$R_m = (A \times B) \bigcup (\neg A \times V) = \int_{U \times V} (m_A(u) \wedge m_B(v)) \vee (1 - m_A(u)) / (u,v)$$

$$B'_a = A' \circ R_a = A' \circ [(\neg A \times V) \oplus (U \times B)]$$

$$m_{B'_a}(v) = \bigvee_{u \in U} \{m_{A'}(u) \wedge [1 \wedge (1 - m_A(u) + m_B(v))]\}$$

Mamdani: $$R_c = A \times B = \int_{U \times V} m_A(u) \wedge m_B(v) / (u,v)$$

Mizumoto: $$R_s = A \times V \underset{s}{\Rightarrow} U \times B = \int_{U \times V} [m_A(u) \underset{s}{\rightarrow} m_B(v)] / (u,v)$$

where $$m_A(u) \underset{s}{\rightarrow} m_B(v) = \begin{cases} 1, & m_A(u) \le m_B(v) \\ 0, & m_A(u) > m_B(v) \end{cases}$$

$$R_g = A \times V \underset{g}{\Rightarrow} U \times B = \int_{U \times V} [m_A(u) \underset{g}{\rightarrow} m_B(v)] / (u, v)$$

$$m_A(u) \underset{g}{\rightarrow} m_B(v) = \begin{cases} 1, & m_A(u) \le m_B(v) \\ m_B(v), & m_A(u) > m_B(v) \end{cases}$$

$$R_{sg} = (A \times V \underset{s}{\Rightarrow} U \times B) \bigcap (\neg A \times V \underset{g}{\Rightarrow} U \times \neg B)$$
$$= \int_{U \times V} \{[m_A(u) \underset{s}{\rightarrow} m_B(v)] \wedge [(1 - m_A(u)) \underset{g}{\rightarrow} (1 - m_B(v))]\} / (u, v)$$

$$R_{gg} = (A \times V \underset{g}{\Rightarrow} U \times B) \bigcap (\neg A \times V \underset{g}{\Rightarrow} U \times \neg B)$$
$$= \int_{U \times V} \{[m_A(u) \underset{g}{\rightarrow} m_B(v)] \wedge [(1 - m_A(u)) \underset{g}{\rightarrow} (1 - m_B(v))]\} / (u, v)$$

$$R_{gs} = (A \times V \underset{g}{\Rightarrow} U \times B) \bigcap (\neg A \times V \underset{s}{\Rightarrow} U \times \neg B)$$
$$= \int_{U \times V} \{[m_A(u) \underset{g}{\rightarrow} m_B(v)] \wedge [(1 - m_A(u)) \underset{s}{\rightarrow} (1 - m_B(v))]\} / (u, v)$$

$$R_{ss} = (A \times V \underset{s}{\Rightarrow} U \times B) \bigcap (\neg A \times V \underset{s}{\Rightarrow} U \times \neg B)$$
$$= \int_{U \times V} \{[m_A(u) \underset{s}{\rightarrow} m_B(v)] \wedge [(1 - m_A(u)) \underset{s}{\rightarrow} (1 - m_B(v))]\} / (u, v)$$

$$R_b = (\neg A \times V) \bigcup (U \times B) = \int_{U \times V} [(1 - m_A(u)) \vee m_B(v)] / (u, v)$$

$$R_* = A \times V \underset{*}{\Rightarrow} U \times B = \int_{U \times V} [m_A(u) \underset{*}{\rightarrow} m_B(v)] / (u, v)$$

where $$m_A(u) \underset{*}{\rightarrow} m_B(v) = 1 - m_A(u) + m_A(u) \times m_B(v)$$

There is no principle to judge which one is best on a general basis because the system's performance is closely related to the specific application. We can use experiment to choose the best fit one.

3.2.5 Defuzzification

Defuzzification is a process to select a representative element from the fuzzy output

inferred from the fuzzy control algorithm. There are three common defuzzification techniques:

i) Mean of Maximum (MOM): It calculates the average of those output values that have the highest possibility degrees. It can be expressed formally as:

$$MOM(A) = \frac{\sum_{y^* \in P} y^*}{|P|}$$

ii) Center of Area (COA): The center of area (COA), also referred to as center of gravity or centroid, is the most commonly used defuzzification technique.

$$COA(A) = \frac{\sum_{x} m_A(x) \times x}{\sum_{x} m_A(x)}$$

iii) Height Method: First, convert the consequent membership function C_i into crisp consequent $y=c_i$ where c_i is the center of gravity of C_i. The centroid defuzzification is then applied to the crisp consequents. It can be expressed formally as:

$$y = \frac{\sum_{i=1}^{M} w_i c_i}{\sum_{i=1}^{M} w_i}$$

3.3 Architecture of fuzzy neural networks

There are many architectures of fuzzy neural network in existence. One typical kind of architecture is what is used in Generic Self-Organizing Fuzzy Neural Network (GenSoFNN) [31] and Pseudo Outer Product based Fuzzy Neural Network (POPFNN) [32]. It is actually a Multi-Input Multi-Output (MIMO) system is a five-layer neural network as shown in Figure 3.2. For simplicity, only the interconnections for the output y_m are shown [32].

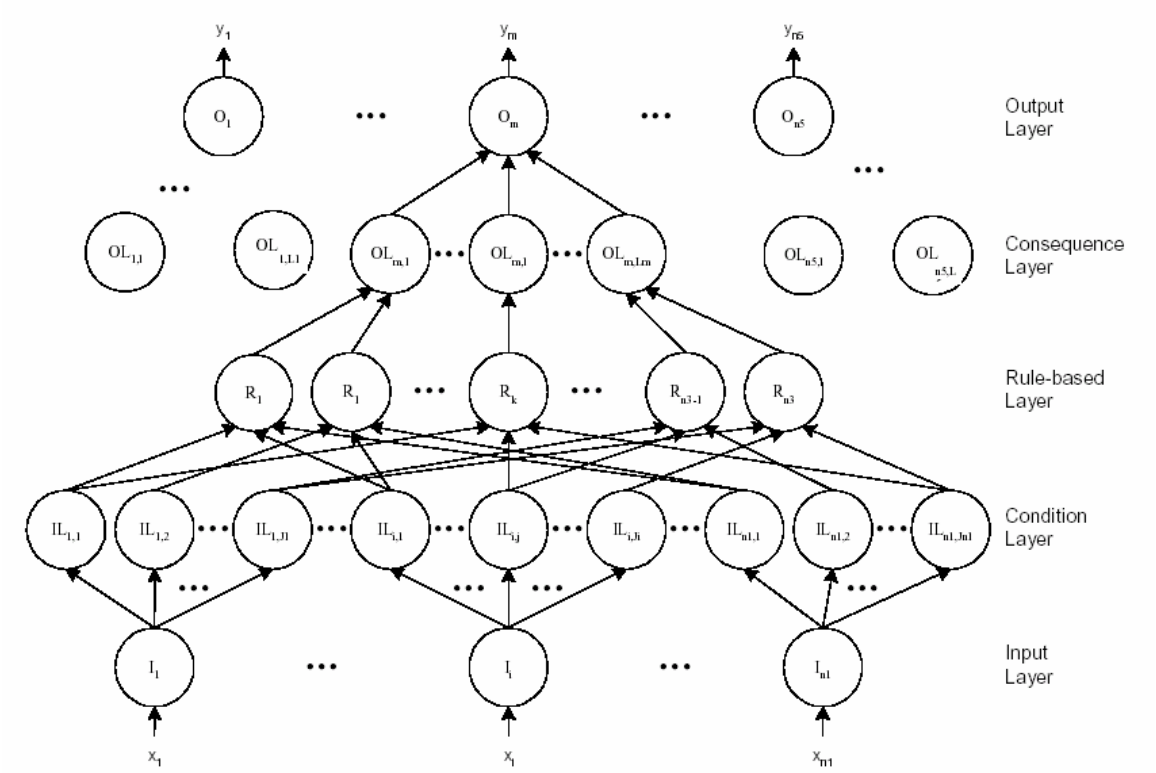


Figure 3.2 Structure of POPFNN-CRI(S)

Each layer in POPFNN-CRI(S) performs a specific fuzzy operation. The inputs and outputs of the POPFNN-CRI(S) are represented as non-fuzzy vector $\mathbf{X}^T=[x_1, x_2, \ldots x_i, \ldots x_{n1}]$ and nonfuzzy vector $\mathbf{Y}^T=[y_1, y_2, \ldots y_l, \ldots y_{n5}]$ respectively. Fuzzification of the input data and defuzzification of the output data are respectively performed by the input and output linguistic layers, while the fuzzy inference is collectively performed by the

rule-base and the consequence layers. The number of neurons in the condition and the rule-base layers are defined in as:

$$n_2 = \sum_{i=1}^{n_1} J_i \qquad n_4 = \sum_{m=1}^{n_5} L_m \qquad n_3 = n_2 \times n_4 .$$

where

J_i is the number of linguistic labels for the i^{th} input,

L_m is the number of linguistic labels for the m^{th} output,

n_1 is the number of inputs,

n_2 is the number of neurons in the condition layer,

n_3 is the number of rules or rule-based neurons,

n_4 is the number of linguistic labels for the output, and

n_5 is the number of outputs.

A detailed description of the functionality of each layer is given as follows:

i) Input linguistic layer:

net input: $f_i^I = x_i$, and

net output: $o_i^I = f_i^I$

where: x_i = value of the i^{th} input

ii) Condition layer:

Each input-label node $IL_{i,j}$ represents the j^{th} linguistic label of the i^{th} linguistic node from the input layer. The input-label nodes constitute the antecedent of the fuzzy rules. Each node is represented by a trapezoidal membership function $m_{i,j}(x)$ described by a fuzzy interval formed by four parameters $(a_{i,j}, b_{i,j}, g_{i,j}, d_{i,j})$ and a centroid $v_{i,j}$ as shown in Fig. 3.3.

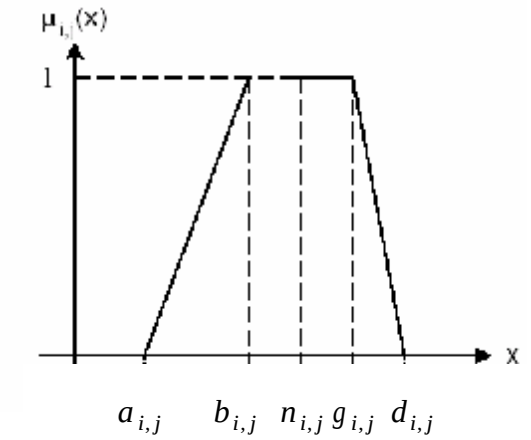


Figure 3.3 Trapezoidal-shaped membership function

net input: $f_{i,j}^{II} = o_i^I$, and

net output:
$$o_{i,j}^{II} = \begin{cases} 0 & \text{if } f_{i,j}^{II} < a_{i,j}^{II} \text{ or } f_{i,j}^{II} > d_{i,j}^{II} \\ \dfrac{a_{i,j}^{II} - f_{i,j}^{II}}{a_{i,j}^{II} - b_{i,j}^{II}} & \text{if } a_{i,j}^{II} \le f_{i,j}^{II} \le b_{i,j}^{II} \\ 1 & \\ \dfrac{d_{i,j}^{II} - f_{i,j}^{II}}{d_{i,j}^{II} - g_{i,j}^{II}} & \text{if } b_{i,j}^{II} \le f_{i,j}^{II} \le g_{i,j}^{II} \end{cases}$$

where

$[a_{i,j}^{II}, d_{i,j}^{II}]$ is the kernel of the fuzzy interval for the j^{th} linguistic label of the i^{th} input,

$[b_{i,j}^{II}, g_{i,j}^{II}]$ is the support of the fuzzy interval for the j^{th} linguistic label of the i^{th} input, and

o_i^l is the output of i^{th} input node.

iii) Rule-base layer

net input: $f_k^{III} = \min_i (o_{i,j}^{II})$, and

net output: $o_k^{III} = f_k^{III}$.

where

$o_{i,j}^{II}$ = output of the input-label node that forms the antecedent conditions for the i^{th} input to the k^{th} fuzzy rule R_k.

iv) Consequence layer

net input: $f_{m,l}^{IV} = \max_k (o_k^{II})$, and

net output: $o_{m,l}^{IV} = f_{m,l}^{IV}$.

where

o_k^{III} = output of the rule node R_k whose consequence is $Ol_{m,l}$.

v) Output Linguistic layer

net input: $$f_m^V = \begin{cases} \sum_{l=1}^{L(m)} (v_{m,l}^{IV} \times (g_{m,l}^{IV} - b_{m,l}^{IV}) \times o_{m,l}^{IV}) & \text{if } g_{nmk}^{IV} > b_{m,l}^{IV} \\ \sum_{l=1}^{L(m)} (v_{m,l}^{IV} \times o_{m,l}^{IV}) & \text{if } g_{nmk}^{IV} = b_{m,l}^{IV} \end{cases}$$

net output: $$o_m^V = \begin{cases} \dfrac{f_m^V}{\sum_{l=1}^{L(m)} (v_{m,l}^{IV} \times (g_{m,l}^{IV} - b_{m,l}^{IV}))} & \text{if } g_{nmk}^{IV} > b_{m,l}^{IV} \\ \dfrac{f_m^V}{\sum_{l=1}^{L(m)} v_{m,l}^{IV}} & \text{if } g_{nmk}^{IV} = b_{m,l}^{IV} \end{cases}$$

where

$v_{m,l}^{IV}$ = the centroid of the output-label node $OL_{m,l}$, and

$g_{m,l}^{IV}$, $b_{m,l}^{IV}$ = the width of the membership function for output-label node $OL_{m,l}$.

3.4 Self-organizing (*Clustering*) techniques in FNN

Generally FNNs perform cluster analysis on each dimension of the inputs and outputs of training data to determine the fuzzy sets, which are subsequently used to derive the fuzzy rules by connecting the input and output fuzzy sets. After the fuzzy inference system is chosen, several parameters need to be learned from training data. The challenges lie in:

i) Required prior knowledge such as number of clusters for different sets of training data, such as in Pseudo Outer Product based Fuzzy Neural Network (POPFNN).

ii) No principled method to configure the parameters of membership functions or parameters for learning process, e.g. set support parameter and STEP in Discrete Incremental Clustering (DIC).

iii) How to make the number of clusters as small as possible so that the rule number can be effectively reduced. This is also known as horizontal reduction.

iv) How to be resistant to noisy/spurious training data and overcome the stability-plasticity dilemma. Most partition-based clustering techniques, such as fuzzy *C*-means (FCM), Linear Vector Quantization (LVQ) and LVQ-inspired technique such as modified LVQ, fuzzy Kohonen partitioning (FKP) and pseudo FKP, are all susceptible to noisy data and lack the flexibility to incorporate new clusters of data after the training has completed. This is called stability-plasticity dilemma, making online learning difficult.

There are many fuzzy clustering techniques, such as: DIC, Fuzzy Kohonen Partition (FKP), Pseudo Fuzzy Kohonen Partition (PFKP), fuzzy *C*-means (FCM), LVQ, modified LVQ, self-organizing map (SOM), fuzzy adaptive resonance theory (fuzzy ART), etc.

As the rules used for implementing FPGA routing is generated by POPFNN, we take a look at the fuzzy membership learning algorithms in POPFNN: FKP and PFKP. The difference between FKP and PFKP is that the latter produces pseudo fuzzy partitions while the former only produces fuzzy partitions. The former is a supervised learning algorithm, while the latter is unsupervised.

Step 1: Define *c* as the number of classes, $l < \frac{1}{\Omega}$ as the learning constant, η as the learning width and a small positive number ε as a stopping criterion; where Ω = number of data vectors in a cluster, *n*=total number of data vectors.

Step 2: Initialise the training iteration $T = 0$ and the weights $v_i^{(0)}$ with

$$v_i^{(0)} = \min_k(x_k) + \frac{i+1/2}{c}(\max_k(x_k) - \min_k(x_k)) \quad \text{for } i = 1,\ldots,c,\ k = 1,\ldots, n.$$

Step 3: Initialize $v_i^{(T+1)} = v_i^{(T)}$ for $i = 1,...,c$.

Step 4: For k = 1..n:

FKP: Determine the i^{th} cluster the data x_k belongs to from the training data.

PFKP: Find the winner using:

$$| x_k - v_i^{(T+1)} | = \min_j(| x_k - v_j^{(T+1)} |) \text{ for } j = 1, ...,c.$$

Update weights v_i of

FKP: the i^{th} cluster

PFKP: the winner *i*

with $$v_i^{(T+1)} = v_i^{(T)} + l\,(x_k - v_i^{(T)})$$

Step 5: Compute $e^{(T+1)}$ using $e^{(T+1)} = \sum_{k=1}^{n} | x_k - v_i^{(T+1)} |$

Step 6: Compare $e^{(T+1)}$ and $e^{(T)}$ where $e^{(0)} = 0$, using $de^{(T+1)} = e^{(T+1)} - e^{(T)}$.

Step 7: If $de^{(T+1)} \le e$, stop, otherwise, repeat step 3-7 for $T = T + 1$.

Step 8: Initialize $a_i = b_i = d_i = g_i = j_i = v_i^{(T+1)}$ for $i = 1, \ldots, c$.

Step 9: For $k = 1, \ldots, n$

FKP: Determine the i^{th} cluster the data x^k belongs to from the training data.

PFKP: Find the winner using $| x_k - j_i | = \min_j (| x_k - j_j |)$ for $j = 1, \ldots, c$.

Update pseudo weights j_i of

FKP: the i^{th} cluster

PFKP: the winner i

the i^{th} cluster using $j_i = j_i + h(x_k - j_i)$

Update the four points of the Trapezoidal Fuzzy Number (T_rFN) with

FKP: $a_i = \min(a_i, x_k)$

PFKP:

$$\alpha_i = \begin{cases} \min(\alpha_i, x_k) & \text{for } i = 1 \\ d_{i-1} & \text{for } i > 1 \end{cases}$$

$$b_i = \min(b_i, j_i)$$

FKP: $g_i = \max(g_i, x_k)$

PFKP:

$$\alpha_i = \begin{cases} \max(g_i, x_k) & \text{for } i = c \\ b_{i+1} & \text{for } i < c \end{cases}$$

3.5 Rule formulation techniques in FNN

The rule formulation techniques are different between TSK-based and CRI-based models. Even in CRI-based models, different approaches might be adopted. In GenSoFNN, RuleMap is used while the method used in POPFNN to identify the fuzzy rules is the Pseudo Outer-Product (POP) learning algorithm. The POP learning algorithm is a simple one-pass learning algorithm. In POPFNN-CRI(S), each node in the condition and consequence layers represents a linguistic label once the membership functions have been identified. Under the POP learning algorithm, the set of training data {**Xp**, **Yp**}, where **Xp** is the input vector and **Yp** is the output vector, is simultaneously fed into both the input linguistic and output linguistic layers. The membership values of each input-label node oII are then determined. These values are subsequently used to compute the firing strength fIII of the rule nodes in the rule-base layer. Similarly, the membership values of each output-label node are determined by feeding the output value back from the output layer to the consequence layer. The weights of the consequence layer linking the rule-based layer are then determined using: $w_{k,m,l} = \sum_{p=1}^{n} f_k^{III}(X^p) \times m_{m,l}(y_m^p)$ (*)

$w_{k,m,l}$ = weight of the link between the k^{th} rule node and the l^{th} linguistic label for the m^{th} output, and

$f_k^{III}(X^p)$ = firing strength of k^{th} rule node when presented with input vector **Xp**, and

$m_{m,l}(y_m^p)$ = membership value of the m^{th} output of **Yp** with the fuzzy subset $Y_{m,l}$ that semantically represents the l^{th} linguistic label of the m^{th} output.

The weights in Equation (*) are initially set to zero. After performing POP learning, these weights represent the strength of the fuzzy rules having the corresponding output-label nodes as their consequences. Among the links between a rule node and the output-

label nodes, the link with the highest weight is chosen and the rest are deleted. The links with zero weights to all output-label nodes are also deleted. The remaining rule nodes after this link selection process subsequently represent the rules used in the POPFNN-CRI(S).

3.6 Problems in applying FNN to network routing

Although FNN is a powerful data analysis and prediction tool, it is very difficult to apply FNN to interconnection network due to the following reasons:

3.6.1 Exponentially growing number of rules

With careful re-examination of the Virus Infection Clustering and clustering techniques in POPFNN and GenSoFNN, it is clear that the clustering process is related only to the input of training examples, with no relationship with the respective output. To make routing decision, it is indispensable to take the binary address of nodes into consideration. So they are selected as part of FNN's inputs. However, in a *n*-dimension network, if in the training set, we feed all the 2^n combinations to FNN, then obviously, each input *i* will be assigned two linguistic labels, namely H_i centering on 1 and L_i centering on 0. Recall the process of rule formulation. No matter whether Mamdani or TSK, SAM model is used, the rule antecedent is always in the form of *IF* x_1 *is* A_{i1} *and ... and* x_r *is* A_{ir}. So the rule number is always in the magnitude of 2^n with each tuple $(x_1, x_2, ..., x_n)$ ($x_i \in \{H_i, L_i\}$) corresponding to a rule. In other words, the FNN is just memorizing each case without any intelligence demonstrated. In practice, this number is intolerable.

A trial to circumvent this problem is to convert the *n*-digit binary number into its corresponding decimal value for input. This is supported by the fact that the *n* bits are

independent. However, as what counts is the bit pattern of the node address, this attempt suffers from the following problem. For example, at current node, a packet is to be sent to 10000 and another packet is to be sent to 01111. In decimal value, their difference is only 1. However, the routing decisions for them are quite different. Experiments also show that this conversion will not reduce the number of rules effectively because more linguistic labels are needed for each input.

Another attempt to overcome the problem is to use pure CMAC and feed in the decimal value. The result still shows that unless the resolution grows exponentially with dimension, the error rate is intolerable. The test is run on learning the function of bitwise XOR. The inputs are two integers ranging from 0 to 127. The output is the bitwise XOR of the inputs. In the training set, all 128*128 combinations of inputs are enumerated and the testing set is same as the training set. The following Figure 3.4 demonstrates the trend of error rate with respect to the resolution *r*. The resolution *r* applies to both inputs simultaneously.

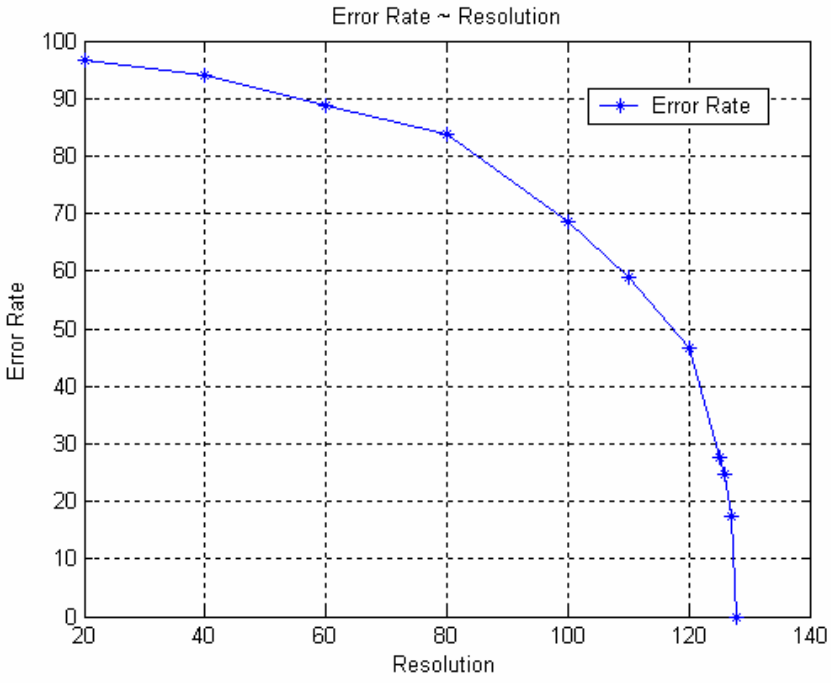


Figure 3.4 Error rate versus Resolution for learning bitwise XOR

Fig. 3.4 shows that the error rate decreases slowly when the resolution is far from 128. Actually, the error rate goes below 50% only when CMAC is nearly memorizing all individual maps from input space to output space. This rote is not acceptable due to its large space cost for storing rule base.

3.6.2 Too long off-line training time

Suppose the input for FNN is three *n*-bit binary strings: current node address, destination address, and safety vector of current node [54]. Then for a 5-dimension network, if we use all the $2^{2\times5}$ combinations of (source, destination) pair, the training time is about 2 minutes with POPFNN on a 1.7GHz CPU computer. For networks of practical size, say 11 dimensions, the training time will be intractable. The problem in nature is that the application of routing in interconnection network is based on binary discrete numbers. The FNN is heavily dependent on the clustering of each dimension of input, also called horizontal reduction. So the range of each input can be very large but the number of input can not be too high because the algorithm's time complexity is $O(\prod_{i=1}^{n} I_i \cdot \prod_{j=1}^{m} O_j \cdot T)$, where I_i stands for the number of linguistic labels for the i^{th} input, O_j stands for the number of linguistic labels of the j^{th} output, and T stands for the number of training examples. However, our binary application makes $I_i = 2$ for $i \in [1, n]$ and n linear to network dimension, so that the complexity is exponential to the dimension.

Besides converting binary numbers into one decimal number, another way to tackle this problem is to reduce the number of training examples. If we provide all possible cases of input in the training set, then as the training time is linear to the training set size, it is inevitable to suffer from $O(2^n)$ time complexity where n is the dimension of the network. We have noted that in most cases, the routing decision in a network with faults is the

same as that in the fault-free setting. The proportion of those decisions affected by faults is so small that an FNN even neglecting them will also achieve a very high percentage of correctness (asymptotically approaching 100%). Thus, to prepare the training set, we choose a small proportion of those cases that are not affected by faults while recording all the cases that are affected. The choice of the former is just by random. However, the harvest is not significant. And the new problem is what proportion of the former cases need to be preserved in order to reach the best performance.

3.6.3 Difficulty in discussion of non-fuzzy metrics

In network routing by FNN, the most important problem in theory is the discussion of metrics of performance. For example, there can be no theoretical deduction of whether the routing strategy is deadlock free or livelock free. We can't prove how many faults can be tolerated. It is also hard to derive in theory the upper bound of path found.

One way to deal with the problem is by simulation. But to compare with other routing strategies, such an approach is not appropriate, because currently no routing strategy is measured by how likely it will lead to deadlock or livelock. The occurrence of deadlock and livelock might result from the routing decision of many packets at many nodes. So such a benchmark is not easy. More importantly, there is no way to predict how many faults can be tolerated. This will put the routing strategy at a disadvantage when high and predictable reliability are desired.

3.7 A possible method for using FNN

For low dimensional networks, the FNN can be applied. But we have to be careful with designing inputs and outputs of the fuzzy neural network. For example, at 000, if a packet is to be sent to 111, then it can use any of the 3 dimensions. However, which one

is adopted in training example is important because choosing randomly will lead to inconsistent training examples.

The final approach used in implementing FPGA is not a direct routing strategy based on FNN. At each node, it uses the FNN to estimate the distance of each neighbor to the destination. And then choose the best one together with such considerations as not immediately backtracking to the sender, and not using a faulty link. In other words, the input of the FNN is:
(*n* bits for current address), (*n* bits for destination address), (*n* bits for current node's safety vector)

Output of FNN is the real distance between current node and the destination in the presence of faulty components.

Note here, when using trained FNN to route, the 'current address' above is actually fed by the neighbors address and 'current node's safety vector' is actually fed by the neighbor's safety vector.

Chapter 4: A Fault-tolerant Routing Strategy for Fibonacci-Class Cubes

4.1 Introduction

Fibonacci-class Cubes originate from *Fibonacci Cube* (*FC*) proposed by Hsu [12][13][16], and its extended forms are *Enhanced Fibonacci Cube* (*XFC*) by Qian [14] and *Extended Fibonacci Cube* (*XFC*) by Wu [15]. This class of interconnection network uses fewer links than the corresponding binary hypercube, with the scale increasing slower because Fibonacci number is of order $O((\frac{1+\sqrt{3}}{2})^n) < O(2^n)$. That allows more choices of network size. In structural aspects, these two extensions virtually maintain all desirable properties of *FC* and improve it by ensuring the *Hamiltonian* property [14][15]. Besides, there is an ordered relationship of containment between the series of *XFC* and *EFC*, together with binary hypercube and regular *FC* [15] as shown in Fig. 4.1 and 4.2:

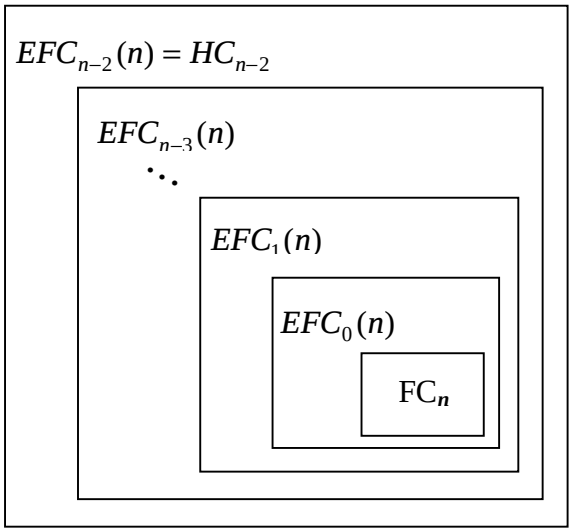


Figure 4.1 Relationship between binary hypercube, regular Fibonacci Cube and Enhanced Fibonacci Cubes

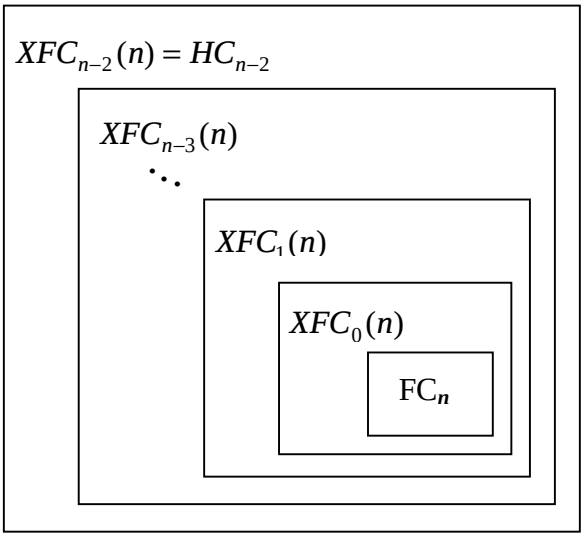


Figure 4.2 Relationship between binary hypercube, regular Fibonacci Cube and Extended Fibonacci Cubes

Lastly, they all allow efficient emulation of other topologies such as binary tree (including its variants) and binary hypercube. In essence, Fibonacci-class Cubes are superior to binary hypercube for low growth rate and sparse connectivity, with little loss of its desirable topological and functional (algorithmic) properties.

Though Fibonacci-class Cubes provide more options of incomplete hypercubes to which a faulty hypercube can be reconfigured and thus tend to find applications in fault-tolerant computing for degraded hypercube computer systems, there are no existing fault-tolerant routing algorithms. This is a common shortcoming of link-diluted hypercubic variants. In this chapter, we propose a unified fault-tolerant routing strategy for Fibonacci-class Cubes, named **F**ault-**T**olerant **F**ibonacci **R**outing (**FTFR)**. It has the following properties:

- It can be applied to all Fibonacci-class Cubes in a unified fashion, with only minimal modification of structural representation.
- The maximum number of faulty components tolerable is the network's node availability [18] (the maximum number of faulty neighbours of a node that can be tolerated without disconnecting the node from the network).
- Each node requires only one round of fault status exchange with its neighbours.
- For a n-dimension Fibonacci-class Cube, each node, with degree deg, maintains and updates at most $(deg+2)$ n-bit vectors, among which: 1) a n-bit availability vector indicates the local non-faulty links, 2) a n-bit input link vector indicates the input message link, 3) deg copies of its deg neighbors' n-bit availability vector indicate dimension availability of its neighbors.
- Provided the number of component faults in the network does not exceed the network's node availability, and the source and destination nodes are not faulty, **FTFR** guarantees a message path length not exceeding $n+H$ empirically and $2n+H$ theoretically, where n is the dimension of the network and H is the Hamming distance between source and destination.

- Generates deadlock-free and livelock-free routes.
- Can be implemented almost entirely with simple and practical routing hardware requiring minimal processor control (refer to Chapter 7 for the FPGA implementation).

The rest of this chapter is organized as follows. Section 4.2 reviews several versions of definitions of Fibonacci-class Cube, together with comments and initial analysis. Section 4.3 presents a Generic Approach for Cycle-free Routing (GACR), which is used as a component of the whole strategy. Section 4.4 develops the fault-tolerant routing algorithm **FTFR** and Section 4.5 illustrates its application with an example. The design of a simulator and simulation results will be presented in the Chapter 4 and 5 respectively. Finally, the routing strategy is implemented on an FPGA chip. This is described separately in Chapter 7.

4.2 Definition and analysis

Though Fibonacci-class Cubes are very similar and are all based on a sequence with specific initial conditions, they do have some different properties that call for special attention.

4.2.1 Definitions of Fibonacci-class Cubes

We first quote the definition *Fibonacci Cube* proposed by Hsu [12].

(*Definition* 4.1) *Fibonacci number*

The well-known *Fibonacci number* is defined by: $f_0 = 0,\ f_1 = 1,\ f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$.

(*Definition* 4.2) *order-n Fibonacci code*

The *order-n Fibonacci code* of integer $i \in [0,\ f_n - 1]\ \ (n \geq 3)$ is defined as $(b_{n-1}, \cdots, b_3, b_2)_F$ where b_j is either 0 or 1 for $2 \leq j \leq (n-1)$ and $i = \sum_{j=2}^{n-1} b_j \cdot f_j$.

(*Definition* 4.3) *Fibonacci Cube of order n* ($n \geq 3$)

Fibonacci Cube of order n ($n \geq 3$) is a graph $FC_n = <V(f_n), E(f_n)>$, where $V(f_n) = \{0,\ 1, \cdots, f_n - 1\}$ and $(i, j) \in E(f_n)$ if and only if $H(I_F, J_F) = 1$, where I_F, J_F are the Fibonacci codes of *i* and *j*, respectively. $H(I_F, J_F)$ stands for the *Hamming* distance between I_F and J_F.

Another equivalent definition which is more unified with *Enhanced Fibonacci Cube* and *Extended Fibonacci Cube* is:

(*Definition* 4.3') *Fibonacci Cube of order n* ($n \geq 3$) [12][14]

Let $FC_n = (V_n, E_n)$, then $V_n = 0 \| V_{n-1} \bigcup 10 \| V_{n-2}$ for $n \geq 5$, where $\|$ denotes the concatenation operation. $V_3 = \{1,\ 0\}$, $V_4 = \{01,\ 00,\ 10\}$. Two nodes in FC_n are connected by an edge in E_n if and only if their labels differ in exactly one bit position.

(*Theorem* 4.1)

Fibonacci Cube of order n ($n \geq 3$) can be equivalently defined as a graph whose node addresses are $(n-2)$-bit binary number in which there are no two consecutive 1's. Edges exist between nodes whose *Hamming* distance is 1.

Proof:

Let $V_n^{'} = \{a_{n-3}a_{n-4}...a_1a_0 \mid a_i \in \{0,\ 1\}$, for $i \in [0, n-3]$ and for $\forall j \in [0, n-4],\ a_{j+1}a_j \neq 11\}$.

Obviously, to prove *Theorem* 4.1, it is sufficient to prove that $V_n = V_n^{'}$ because the definition of link in *Theorem* 4.1 is the same as that in *Definition* 4.3'. First, it is obvious that $V_n \subseteq V_n^{'}$. We prove $V_n \supseteq V_n^{'}$ inductively. As the basis, it is clear that $V_n = V_n^{'}$ for *n* = 3, 4. If $V_n = V_n^{'}$ holds for $n < k$ ($k > 4$), then when $n = k$, for each binary address $a_{k-3}a_{k-4}...a_1a_0 \in V_k^{'}$, we discuss two cases.

1) $a_{k-3} = 0$. As $a_{k-4}...a_1a_0 \in V_{k-1}^{'}$, thus $a_{k-4}...a_1a_0 \in V_{k-1}$. Then $a_{k-3}a_{k-4}...a_1a_0 = 0a_{k-4}...a_1a_0 \in 0 \| V_{k-1} \subseteq V_k$.

2) $a_{k-3} = 1$. Then $a_{k-4} = 0$. As $a_{k-5}...a_1a_0 \in V_{k-2}^{'}$, thus $a_{k-5}...a_1a_0 \in V_{k-2}$. Then $a_{k-3}a_{k-4}...a_1a_0 = 10a_{k-5}...a_1a_0 \in 10 \| V_{k-2} \subseteq V_k$.

Combine 1), 2), we get $V_n \supseteq V_n^{'}$. So $V_n = V_n^{'}$ holds for $n = k$. *Theorem* 4.1 is proved. g

The definition in *Theorem* 4.1 is more suitable for discussing routing strategies in *Fibonacci Cube.*

Enhanced Fibonacci Cube and *Extended Fibonacci Cube* can be defined in a similar way:

(*Definition* 7.4) *Enhanced Fibonacci Cube of order n* ($n \geq 3$) [14]

Let $EFC_n = < V_n, E_n >$ denote the *Enhanced Fibonacci Cube of order n,* then

$V_n = 00 \,||\, V_{n-2} \bigcup 10 \,||\, V_{n-2} \bigcup 0100 \,||\, V_{n-4} \,|| \bigcup 0101 \,||\, V_{n-4}$. Two nodes in EFC_n are connected by an edge in E_n if and only if their labels differ in exactly one bit position. As initial conditions for recursion, $V_3 = \{1,0\}$, $V_4 = \{01,\ 00,\ 10\}$

$V_5 = \{001,\ 101,\ 100,\ 000,\ 010\}$ and

$V_6 = \{0001,\ 0101,\ 0100,\ 0000,\ 0010,\ 1010,\ 1000,\ 1001\}$.

(*Definition* 7.5) *Extended Fibonacci Cube series of order n* [15]

A series of *Extended Fibonacci Cubes* is defined as $\{ XFC_k,\ k \geq 1 \}$, where

$XFC_k(n) = \{V_k(n), E_k(n)\}$. $V_k(n) = 0 \,||\, V_k(n-1) \bigcup 10 \,||\, V_k(n-2)$ for $n \geq k+4$. Two nodes in $XFC_k(n)$ are connected by an edge in $E_k(n)$ if and only if their labels differ in exactly one bit position. As initial conditions for recursion, $V_k(k+2) = \{a_{k-1} \cdots a_1 a_0 \mid a_i \in \{0,\ 1\}$ for $i \in [0, k-1]\}$, $V_k(k+3) = \{a_k \cdots a_1 a_0 \mid a_i \in \{0,\ 1\}$ for $i \in [0, k]\}$.

4.2.2 Comments and Analysis

The following property is important for our routing algorithm. Let current node address be *u* and destination node address be *d,* then each dimension corresponding to 1 in $u \oplus d$ is

called preferred dimension, where $\oplus$ stands for bitwise XOR operation. Due to the definition of Fibonacci-class Cubes, when a packet is routed in the network, it is quite likely that links in one or more preferred dimensions are not available at current node. But the following *Theorem* 4.2 guarantees that in a fault-free setting, there is always at least one preferred dimension available at its present node. Unlike binary hypercube, this is not a trivial result.

(*Theorem* 4.2)

In a fault-free *Fibonacci Cube, Enhanced Fibonacci Cube* or *Extended Fibonacci Cube*, there is always a preferred dimension available at the packet's present node before the destination is reached.

Proof :

Suppose we are discussing an *n*-dimension Fibonacci-class Cube. This means that we are discussing *FC, XFC and EFC* of order $n+2$. Let the binary address of current node be $a_{n-1}\cdots a_1 a_0$ and the destination be $d_{n-1}\cdots d_1 d_0$. Let the rightmost (least significant) bit correspond to dimension 0 while the leftmost bit correspond to dimension $n-1$.

Case I: *Fibonacci Cube* FC_{n+2}. Obviously, if the destination has not been reached, there is always a preferred dimension $i \in [0, n-1]$. If $a_i = 1$ and $d_i = 0$, then there is always a preferred link available at dimension *i* because changing one '1' in a valid address into 0 always produces a new valid address. So we only need to consider $a_i = 0$ and $d_i = 1$. When $n \le 3$, *Theorem* 4.2 can be easily proven by enumeration. So now suppose $n \ge 4$. Obviously, if $i \in [1, n-2]$, then $d_{i-1} = 0$, $d_{i+1} = 0$. If $a_{i-1} = 1$, then $i-1$ is an available preferred dimension. If $a_{i+1} = 1$, then $i+1$ is an available preferred dimension. If

$a_{i-1} = a_{i+1} = 0$, then dimension *i* is an available preferred dimension because inverting a_i to 1 will not produce two consecutive 0's in the new nodes address. This satisfies the precondition of *Theorem* 4.1, so that the new address is ensured to be a valid node address. If $i = 0$, then $d_1 = 0$. If $a_1 = 1$, dimension 1 is an available preferred dimension. If $a_1 = 0$, then dimension 0 is an available preferred dimension for the same reason as in $i \in [1, n-2]$. If $i = n-1$, then $d_{n-2} = 0$. If $a_{n-2} = 1$, then dimension *n*-2 is an available preferred dimension. If $a_{n-2} = 0$, then dimension $n-1$ is an available preferred dimension for the same reason as for $i \in [1, n-2]$. In whatever case, *Theorem* 4.2 holds.

Case II: *Extended Fibonacci Cube* $XFC_k(n+2)$

Suppose there is a preferred dimension *i*. If $i < k$, then it always produces a valid address if we invert a_i. If $i \geq k$, the discussion is the same as case I.

Case III: *Enhanced Fibonacci Cube* EFC_{n+2}.

The discussion is similar to case I. We only need to pay attention to the leftmost preferred dimension. Please refer to Appendix I for detailed proof. g

Theorem 4.2 implies that whenever a spare dimension is used, either a faulty component is encountered or all neighbors on preferred dimensions have been visited before. For the latter case, all such preferred dimensions must have been used as spare dimensions before. So both cases can be boiled down to the encounter of faulty components.

Theorem 4.2 implies the possibility that FTFR can be applied to all type of networks which can always ensure the existence of at least one preferred dimension. Actually, we

applied FTFR to all Fibonacci-class Cubes and find that it works well in all cases, including binary hypercube.

4.3 A Generic Approach for Cycle-free Routing (GACR)

4.3.1 Overview

This approach aims at providing a way of avoiding cycles in routing by checking the traversal history. The most valuable strength is that the algorithm only takes $O(1)$ time to check whether a neighbor has been visited before, and only $O(1)$ time to update the coded history record. Other advantages include its wide applicability and easy hardware implementation. It applies to such routing algorithms that deal with a network in which links only connect node pairs whose *Hamming* distance is 1 (called *Hamming* link). All networks constructed by node or link dilution meet the requirement. An extended version of the algorithm can be applied to those networks which have $O(1)$ types of non-*Hamming* links at each node. Thus, such networks as *Folded Hypercube, Enhanced Hypercube* and *Josephus Cube* can also use this algorithm.

The weakest point of this approach lies in the size of message overhead $O(L_m \log n)$, where n is the dimension of the network and L_m is the maximum length of a path a packet can traverse. However, in most cases, it is still within an acceptable bound [19].

4.3.2 Basic GACR

The traversal history is effectively an ordered sequence of dimensions used when leaving each visited node. For example, in Figure 4.3, the route that originates from 000 can be recorded as: 1210121. An obvious equivalent requirement for cycle-freeness is that: if '(' and ')' are inserted into the sequence, then for any combination of the places of '(' and ')' (as long as '(' precedes ')'), there must be at least one number between the brackets which appears for an odd number of times. Put it another way, the equivalent condition for a route to contain cycle is: there exists a way of inserting '(' and ')' into the sequence such that each number in ( ) appears for an even number of time.

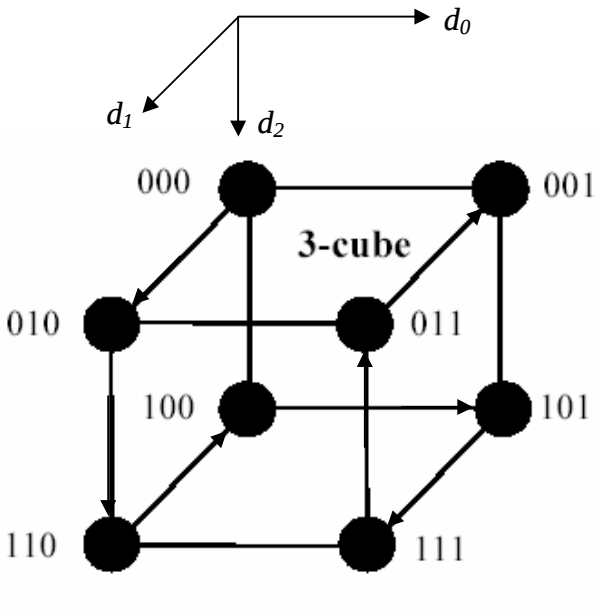


Figure 4.3 Example for routing history

For example, in 1(21012) , 0 appears only one time, which is an odd number. In (1210121), 1 and 2 appear for an even time but 0 still appears for an odd number of time. So neither forms a cycle. But for a sequence of 1234243, there must be a cycle: 1(234243). Suppose at node *p*, the history sequence is $a_1 a_2 \cdots a_n$, and it is guaranteed that no cycle exists hitherto, then to check whether using dimension a_{n+1} will cause any cycle, we only need to check whether in $(a_n a_{n+1})$, $(a_{n-2} a_{n-1} a_n a_{n+1})$, $(a_{n-4} a_{n-3} a_{n-2} a_{n-1} a_n a_{n+1})$ … each number will appear for an even time. Here we can omit dimension a_n because immediate backtrack will certainly cause cycle.

We first introduce the basic form of this algorithm that applies only to networks constructed by node/link dilution from binary hypercube. This algorithm is run at each intermediate node so as to ensure that no cycle is formed.

(*Algorithm* 4.1) *Basic GACR*

The data structure is a simple array: port[ ], with each element composed of $\lceil \log n \rceil$ bits. port[i] records the port used when exiting the node that the packet visited $i+1$ hops ago. So when a packet leaves a node, it only needs to append the dimension adopted to the head of the array port[ ]. As each node has only *n* ports and the meaning of dimension is common at all nodes, that is, dimension *c* at node *a* has the same meaning at node *b*, obviously only $\lceil \log n \rceil$ bits are necessary for representing these *n* possibilities. At the source node, the array port[ ] is null.

Suppose at node *x*, the length of the array is *L*. After running the following short code segment, each 0 in *mask* corresponds to a dimension, the using of which will cause an immediate cycle. Thus, the test time only takes one clock cycle.

```
unsigned Preprocess( unsigned port[], int L)
{
        unsigned dim, mask = 0, history = 1 << port[0];
        int k, flag = 1;
        for (k = 1; k < L;  k++)
        {
                dim  = 1 << port[k];
                history ^= dim;
                if (! flag )            // flag ensures that OnlyOne is called every other time
                {
                        if ( OnlyOne (history) )        // check if history has only one 1
                                mask |= history;
                        flag ++;
                }
                else
                        flag --;
```

```
        }
        return ~mask;
}
```

For instance, for the dimension sequence 875865632434121 from source to present, the mask is: 000010011. Because in 875865632434121a, there is a cycle formed when a = 2, 3, 5, 6, 7, or 8.

The operations in this algorithm are all basic logic operations. The *OnlyOne* function which tests whether *history* has and only has one 1 is also easy to implement such that only one clock cycle is required. Suppose $history = x_{n-1}x_{n-2}\cdots x_1x_0$ ($x_i \in \{0, 1\}$ for $i \in [0, n)$), then OnlyOne (history) =

$$\overline{x_{n-1}}x_{n-2}\cdots x_1x_0 + x_{n-1}\overline{x_{n-2}}\cdots x_1x_0 + \cdots + x_{n-1}x_{n-2}\cdots \overline{x_1}x_0 + x_{n-1}x_{n-2}\cdots x_1\overline{x_0},$$

The implementation of this only costs n AND gates and 1 OR gate, taking only one clock cycle. But in software simulation, it takes $O(n)$ time. Attention should be paid to this problem. The logic circuit of function OnlyOne is drawn in Fig. 4.4.

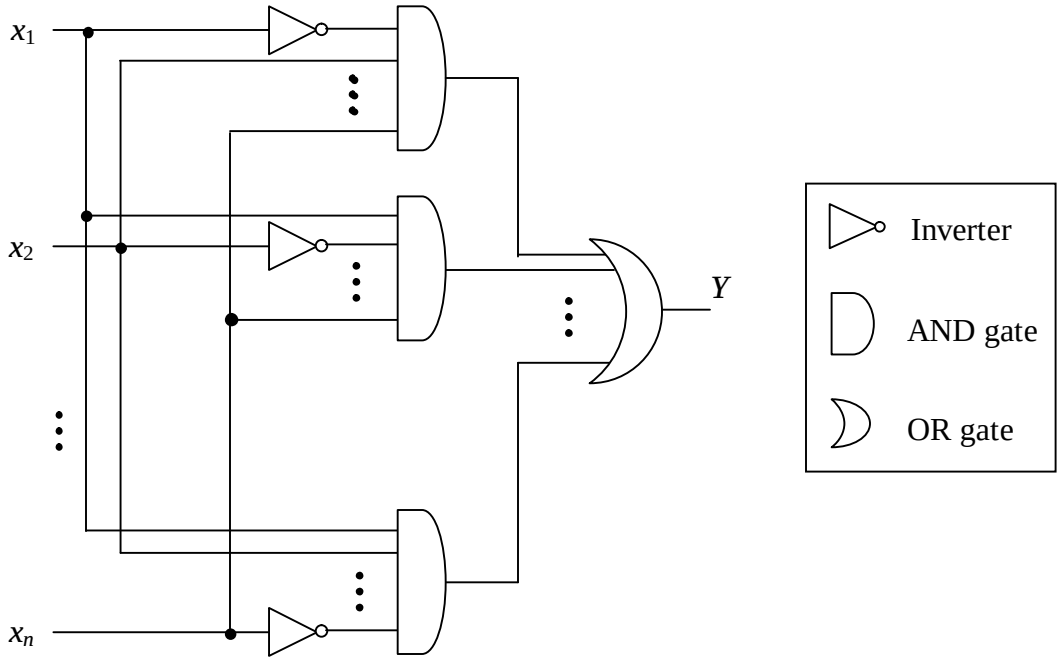


Figure 4.4 Logic circuit of function OnlyOne

Another strength of this algorithm is that the time for running the code above can be reduced to nearly zero because it is orthogonal to the routing algorithm. This makes parallelism and pipelining possible. At first sight, the time complexity is $O(L_{\max})$, where $L_{\max}$ is the length of the longest path the packet can traverse. In a network with heavy load, this preprocess of calculating *mask* can be done when the packet is still waiting in the buffer.

4.3.3 Extended GACR

If the network has $O(1)$ number of non-*Hamming* link types at each node and these links can be represented by a common and uniform way, then *Algorithm* 4.1 can be easily extended. For example, in *Josephus Cube* $JC(n)$ [64], we denote the complementary link as dimension n and the Josephus link as dimension $n+1$. Then the function of Preprocess can be modified into the following form:

(*Algorithm* 4.2) *Extended GACR*

```
void Preprocess( unsigned port[], int L, unsigned *mask1, unsigned *mask2,
                 unsigned *mask3)
{
    unsigned dim, history = 1 << port[0];
    *mask1= *mask2 = *mask3 = 0;
    for ( int k = 1; k < L;  k++)
    {
        if ( port[k] < n )
            dim  = 1 << port[k];
        else if ( port[k] == n )
```

```
                dim = ((1<<n) – 1);
        else  dim = (unsigned) 3;
        history ^= dim;
        if ( OnlyOne (history) )        // check if history has only one 1
                *mask1 |= history;      // for cycle caused by Hamming link
        else if  (AllOne(history) )     // check if history has straight 1's
                *mask2 = 1;             // for cycle caused by complementary link
        else if ( history == (unsigned) 3)      // check the rightmost two bits
                *mask3 = 1;             // for cycle caused by Josephus link
    }
    *mask1 = ~( *mask1);
}
```

mask2 = 1 represents that the use of complementary link will result in a cycle, while *mask3* = 1 stands for the fact that using *Josephus* link will bring about a cycle. The meaning of *mask1* remains the same as *mask* in the basic algorithm.

It might be noticed that the biggest shortcoming lies in the size of message overhead. For most routing algorithms, $L_m = O(n)$ thus $O(L_m \log n) = O(n \log n)$. However, this is still within the acceptable bounds in most applications. For example, the "visited stack" used by [19] incurs message overhead of $(n+1)\lceil \log_2 n \rceil$ bits for an n-dimension binary hypercube.

4.4 Fault-Tolerant Fibonacci Routing (FTFR)

4.4.1 Definition and notation

In a Fibonacci-class Cube of order $n + 2$ (n-dimensional), each node's address is an n-bit binary number where $n > 0$. Let the source node, $\boldsymbol{u}$, be identified by $(a_{n-1} \ldots a_1 a_0)$, where $a_i \in \{0, 1\}$ for all $0 \le i < n$, and the destination node, $\boldsymbol{v}$, by $(b_{n-1} \ldots b_1 b_0)$, where $b_j \in \{0, 1\}$ for all $0 \le j < n$. Then, the identity of the neighboring node of u along the d^{th} dimension, is $u^{(d)}$ for any $0 \le d < n$, where $u^{(k)}$ means inverting the k^{th} bit of the binary address of node u.

(*Definition* 4.5) *route vector*

When a packet reaches current node c, four r-bit route vectors are calculated as follows:

$$R_1 = \sim d \,\&\, c, \qquad R_2 = \sim c \,\&\, d, \qquad R_3 = c \,\&\, d, \qquad R_4 = \sim (c \mid d)$$

Here, '|', '&', '~' represent OR, AND and bitwise NOT operation, respectively.

Obviously, $R_1 \mid R_2 \mid R_3 \mid R_4 = 1^n$, $R_i \,\&\, R_j = 0$ for all $1 \le i, j \le 4 \;\; i \ne j$, where 1^n stands for a sequence of 1 with the length of n.

(*Definition* 4.6) *availability vector*

At each node x, the n-bit binary number *availability vector* (*AV(x)*) records a bit string, indicating by '1' what dimensions are available at x, and by '0' what dimensions are unavailable.

Here a dimension *d* is available means there is a nonfaulty link at *x* to $x^{(d)}$. For example, in Figure 4.5, node 1001 and link (0000, 0001) are faulty. The availability vector of each node is listed in Table 4.1:

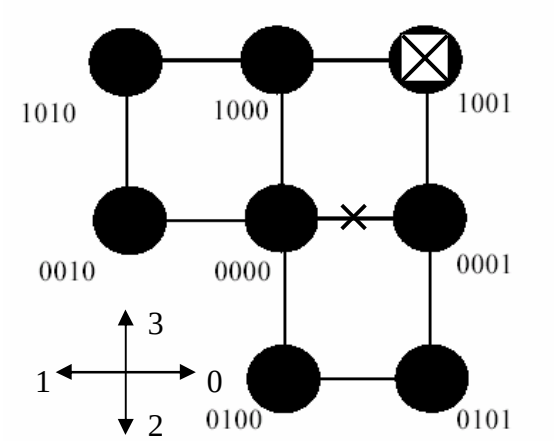


Figure 4.5 Example of *availability vector*

node	*AV*
0000	1110
0001	0100
0010	1010

node	*AV*
0100	0101
0101	0101
1000	1010

node	*AV*
1001	0000
1010	1010

Table 4.1 *availability vector* for Fig. 4.5

Availability vector is crucial for generalizing the applicability of the routing algorithm to other Fibonacci-class Cubes. It is effectively a distributed representation of the network topology, connectivity and fault distribution.

(*Definition* 4.7) *input link vector*

An *n*-bit *input link vector* at node *w* is defined as $I(w) = [l_{n-1} \ldots l_1 l_0]$, where $l_i = 0$ if the message arrives at *w* along the dimension *i* link, otherwise $l_i = 1$ for $0 \le i < n$. Setting the

corresponding bit to 0 for a used input link prevents the link from being used again immediately for message transmission, causing the message to "oscillate" back and forth. An input link vector has all *n* bits set to '1's for a new message generated at the node and after transmission of a received message.

(*Definition* 4.8) *mask vector*

To prevent cycles in the message path and to restrict the freedom of selecting output port, it is also necessary to keep track of link dimensions traversed. As part of the message overhead, a ***mask vector*** may be defined as $DT = [t_{n-1} \cdots t_1 t_0]$. At source node, we clear $DT = [1 \ldots 11]$. After that, whenever a spare dimension is to be used, it must be guaranteed that the corresponding bit in *DT* is 1. But the use of preferred dimension is never restricted. Different from many existing algorithms, each originally preferred dimension (preferred dimension at the source) can be used more than once. When it is used for the first time, *DT* doesn't record it. But at the second time when it is to be used as a spare dimension, its corresponding bit in *DT* is masked, so that it can't be used as a spare dimension again. It will then be used as a preferred dimension. Any 0-bit in *DT* cannot be set back to 1. As for originally spare dimensions, they can be used for at most two times, which is ensured by masking the corresponding bit in *DT* the first time it is used.

(*Definition* 7.9) neighbor condition vector array (NC_k)

Each node periodically exchanges its own availability vector with all neighbors. So it costs at most $O(n^2)$ space to store the neighbor condition. The availability vector of the neighbor on dimension *k* is denoted as NC_k.

4.4.2 Detailed description of FTFR

Empirically, the number of faults FTFR can tolerate is the network's node availability. There is an intricate mechanism in choosing candidate dimension when more than one preferred dimension are available, or when no preferred but several spare dimensions are available. First of all, the *GACR* is used to generate a mask *M*. Only those dimensions whose corresponding bit in (*M* AND *I* (*w*) AND *AV*) is 1 are further investigated. These dimensions are called **available**. To illustrate the algorithm, the following Figure 4.6 is useful. In Figure 4.6, 's' stands for spare dimension or neighbors on it, while 'p' stands for preferred dimension.

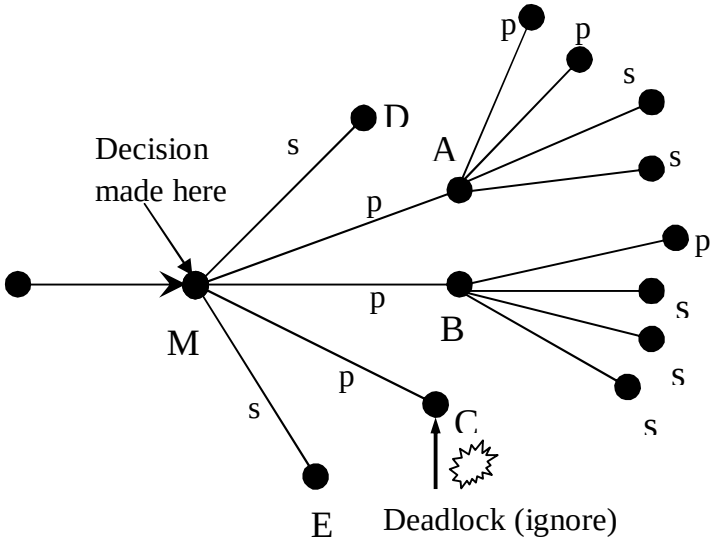


Figure 4.6 Illustrative example of *FTFR*

We divide our discussion into two cases.

(Case I)

We first check the 1's in R_1, R_2 (preferred dimensions). If there are several available preferred neighbors (like A and B), we compare which one has the largest number of non-faulty preferred dimensions. If tie, then compare their number of non-faulty spare dimension. If still tie, choose the lowest dimension. Actually, the value to compare is

given by $n \cdot$(No. of prefer) + (No. of spare). Here, For A, $n \cdot$ (No. of prefer) + (No. of spare) = $2n+2$, while for B, the value is $n+3$. So A is chosen.

(Case II)

If at current node M, there are no preferred dimensions available, spare dimensions have to be used, like D and E. Firstly, the eligibility is checked by *DT*. Then just like in case I, we compare $n \cdot$(No. of prefer) + (No. of spare). After one spare dimension is finally chosen, its corresponding bit in *DT* is masked to 0, so that it will not be used as spare dimension again.

In Case II, if all spare dimensions are masked by *DT*, the algorithm has to abort.

The $m = n \cdot$(No. of prefer) + (No. of spare) is a heuristic metric. After extensive experimentation, it is found that small modifications can be made to *m* so as to improve the performance of FTFR. Suppose the dimension under consideration is *i* and inverting the i^{th} bit of destination *d* produces *d'*= *d XOR* $0^{n-i}10^{i-1}$. If *d'* is a valid node address in that Fibonacci-class Cube, attaching some priority to dimension *i* will be helpful in reducing the number of hops. Hence, we add the value of node availability of the network to *m* for that dimension in such case. In Enhanced Fibonacci Cube, this is an indispensable measure for the algorithm to generate a path to destination when the number of faults in the network is no more than its node availability.

The following are two core routing functions. They are very easy to understand.

// this function is run at M, which looks ahead at A, B, C, D and E

// available = $AV(M)$ AND $I(M)$ AND (mask generated by GACR**)**

// source and destination are both in Fibonacci code

unsigned EnhFibCube::GetNext(unsigned int source, unsigned int destination,
unsigned int available, unsigned int *DT)

```
{
        int max1, max2;
        unsigned x2, temp1, temp2;

        if (source == destination)
                return    DEST_REACH;

        // first get preferred 1->0 dimensions
        x2 = (~destination & source);
        x2 &= available;
        max1 = -1;

        if(x2)    // if there exists some available 1->0 preferred dimensions,
                  // choose the one that has the largest
                  //  n*(No. of // prefer) + (No. of spare),
                  // the value is recorded in max1 (called by reference).
                  temp1 = OneBest(source, destination, x2, *DT, &max1);


        // check preferred 0->1 bits
        x2 = (~source  & destination);
        x2 &= available;
        max2 = -1;

        if(x2)    // such a dimension exists
                  temp2 = OneBest(source, destination, x2, *DT, &max2);

        if(max1 > max2)
                  return temp1;
        else if(max1 < max2)
                  return temp2;
        else if( max1 != -1 )
                  return temp1;

        // check spare 1->1, now make 1->0
        x2 = (source & destination);
        x2 &= available;
        x2 &= *DT;
        max1 = -1;
        if(x2)
                  temp1 = OneBest(source, destination, x2, *DT, &max1);
```

```
        // check spare 0->0, now make 0->1
        x2 = ~(source | destination);
        x2 &= available;
        x2 &= *DT;
        max2 = -1;
        if(x2)
                temp2 = OneBest(source, destination, x2, *DT, &max2);

        if(max1 > max2)
        {
                *DT ^=  (1 << temp1);      // remember to mask spare dimension once used
                return temp1;
        }
        if(max1 < max2)
        {
                *DT ^= (1 << temp2);
                return temp2;
        }
        if( max1 != -1 )
        {
                *DT ^=  (1 << temp1);
                return temp1;
        }
        return ABORT;
}


// each running of this function corresponds to the neighbors of A, B, C, D, E…
// each 1 in x2 corresponds to the candidate dimensions waiting to be tested
// m records the largest n*(No. of prefer) + (No. of spare)
// the return value indicates the selected dimension.
// If all neighbors in x2 are leading to deadlocks or these neighbors have no nonfaulty
// links, m is set to -1 (unchanged as before calling OneBest) and return INFINITY.
unsigned EnhFibCube::OneBest(unsigned int source, unsigned int destination, unsigned
int x2, unsigned int DT, int *m)
{
        unsigned x1, mask, neighbor, prefer, spare, i;
        int max, temp, total;
```

```
	mask = 1;
	max = 0;

	for( i=0; i < Num_Bits ; i++)		// iterate for each dimension
	{
		if(x2&mask)
		{
			neighbor = source ^ mask;		// get the neighbor (A, B, C, D…)
			temp = Fib2Dec(neighbor);		// get the array index of neighbor
			prefer = neighbor ^ destination;  // relative address.
			if(!prefer)					// the neighbor is destination
			{
				*m = 0x7fffffff;		// set m to INFINITY
				return i;		// return corresponding dimension
			}

			total = CalOnes (prefer & Node[temp].avaiVector & ~mask) *Num_Bits;
				// how many preferred dimensions are available at the neighbor

			spare = (~prefer  & DT & Node[temp].avaiVector& ~mask);

			total += CalOnes(spare);
				// how many spare dimensions are available at the neighbor

			if (CheckValid(destination ^ mask, Num_Bits))
				total = total + Node_Availability;

			if(total > max)				// record the max value
			{
				max = total;
				x1 = i;			// record the corresponding dimension
			}
		}
		mask <<= 1;
	}

	if(max == 0)					// return no qualified dimension is found
		return INFINITY;
	*m = max;					// record the max value
	return x1;					// record the corresponding dimension
}
```

4.5 An illustrative Example:

In an 9-dimension Regular Fibonacci Cube F_{11} :

It can tolerate at most $\left\lfloor \frac{9+2}{3} \right\rfloor - 1 = 2$ faulty components

Faulty Node: 000001000 and 000000001

Faulty Link: none

Now we want to go from **101010100** to **000001001**

The path selected is:

Step	876543210	Dimension Used	
	101010100		
(1) **è**	100010100	6	
(2) **è**	000010100	8	
(3) **è**	000010101	0	
(4) **è**	000000101	4	
(5) **è**	000000100	0	
(6) **è**	000000000	2	meet 000001000, 000000001.
(7) **è**	100000000	8	
(8) **è**	100000001	0	
(9) **è**	100001001	3	
(10) **è**	000001001	8	

At step (1), a preferred dimension 6 is used. There are 4 1->0 preferred dimensions available then, namely 2, 4, 6, 8. The metric $n\cdot$ (No. of prefer) + (No. of spare) is 4*9+1, 3*9+2, 3*9+2, 4*9+0, 4*9+0, respectively. After updated for dimensional availability at destination, the final score is 37, 29, 39, 39, respectively. Thus dimension 6 or 8 can be chosen. Here we choose the smaller one. Before step 7, we can always find a 1->0 preferred dimension. At 000000000, neither of the two preferred dimensions (3 and 0) is available because each will lead to a faulty node. So spare dimension has to be used then. The input dimension is 2 and using dimension 4 will lead to deadlock. Therefore, there are only 5 possible dimensions, namely 1, 5, 6, 7, 8. The score they get are (including the possible addition of node availability) are: 14, 25, 25, 25, 27, respectively. So dimension 8 is chosen. Note, now dimension 8 is used as spare dimension and its corresponding bit in DT will be masked. It will never be used as spare dimension again. Afterwards, three preferred dimensions are used successively.

Note here, each faulty component is not encountered twice. The final route is short. Actually, in the 9-dimension Fibonacci Cube with 2 faulty nodes, the longest possible route found by FTFR is 10.

Chapter 5: Exchanged Hypercube

5.1 Introduction

One important means of improving computation speed is by breaking the problem into subcomputation and execute concurrently with multi-processors. In this setting, the communication between processors is crucial. A number of interconnection networks have been designed to deal with the problem. One of the most researched as interconnection network is the binary hypercube [8][9].

The binary hypercube, however, scales too rapidly as its dimension n increases. The more serious problem is the number of edges: $n2^{n-1}$, which grows more drastically than the number of nodes: 2^n. Some variants have been proposed to remove as large a fraction of edges as possible, while, at the same time, preserve the desirable topological properties of the binary hypercube. Examples are *Gaussian Hypercube* [1] and *Reduced Hypercube* [10]. Nevertheless, when edges are diluted, some usefulness of a richer connectivity disappears. Routing between nodes becomes a serious problem, particularly when faulty components exist in the network.

The *Exchanged Hypercube* proposed in this chapter is based on link removal from binary hypercube, possessing only $\frac{1}{n}$ of the number of links in the latter topology with the same number of nodes, where n is the dimension of the network. It is defined with two parameters, which provide more flexibility of network structure. What is more, it maintains virtually all of the desirable properties of the binary hypercube, such as

Hamiltonian property (which ensures the optimal embedment of ring), uniform node degree, low diameter, and various possibilities of decomposition.

An interesting point is that an *Exchanged Hypercube* is isomorphic to a *Gaussian Cube*. It near-optimally emulates binary hypercube. Besides, it can embed meshes with reasonable efficiency (dilation 2, expansion 2, loading 1 and congestion 2). Being *Hamiltonian*, the *Exchanged Hypercube* can optimally embed linear arrays and rings.

The *Extended Binomial Tree*, which is proved to be the spanning tree of the *Exchanged Hypercube*, preserves many desirable properties of the original *Binomial Tree*, with only some minor variations in the initial conditions. This provides a necessary framework for solving many applications such as broadcasting, prefix sum computing and load balancing in *Exchanged Hypercube*.

Finally, a fault-tolerant routing strategy is proposed. For link-diluted hypercubic variants, the common nightmare is the low node availability (the maximum number of faulty neighbours of a node that can be tolerated without disconnecting the node from the network [18]). With refined analysis of the location of faulty components, our algorithm can tolerate more faults than the trivial bound of node availability. Besides, it is livelock free and generates deadlock free routes. It also ensures that a message path length never exceeds $2F$ longer than the optimal path found in a fault-free setting, provided the distribution of faulty components in the network satisfies the precondition of *Theorem* 5.1.

The rest of the chapter is organized as follows. In Section 5.2, we define the *Exchanged Hypercube*, discuss its structural properties including *Hamiltonian* property and present results of its diameter, node degree, node and link complexities. In Section 5.3, the

embeddings of Gaussian Cube, ring, mesh, binary hypercube are studied. In Section 5.4, we define the *Extended Binomial Tree*, together with its labeled form: *Exchanged Tree*. The good properties of these trees and their relationship with *Exchanged Hypercube* are discussed. In Section 5.5, we describe a fault-tolerant routing strategy.

5.2 The Exchanged Hypercube

5.2.1 Definition and Construction

(*Definition* 5.1) *Exchanged Hypercube*

The *Exchanged Hypercube* is defined as $EH(s,t) = (V,E)$ $(s \geq 1, t \geq 1)$, where

$V = \{a_{s-1} \cdots a_0 b_{t-1} \cdots b_0 c \mid a_i, b_j, c \in \{0,1\} \quad for \ \ i \in [0,s), j \in [0,t)\}$

$E = \{(v_1, v_2) \in V \times V \mid \quad$ where $\quad v_1 \oplus v_2 = 1$

or $\quad v_1[s+t:t+1] = v_2[s+t:t+1], \ H(v_1[t:1], v_2[t:1]) = 1, \ v_1[0] = v_2[0] = 1$

or $\quad v_1[t:1] = v_2[t:1], \ H(v_1[s+t:t+1], v_2[s+t:t+1]) = 1, \ v_1[0] = v_2[0] = 0 \ \}$

Here, $v[x:y]$ represents the bit pattern of *v* between dimension *y* and *x* inclusive (we borrow the syntax of *Handel-C* [11]). Let $H(x,y)$ represent the *Hamming* distance between *x* and *y*, where $(x,y) \in V \times V$.

$EH(1,2)$ is shown in Fig. 5.1:

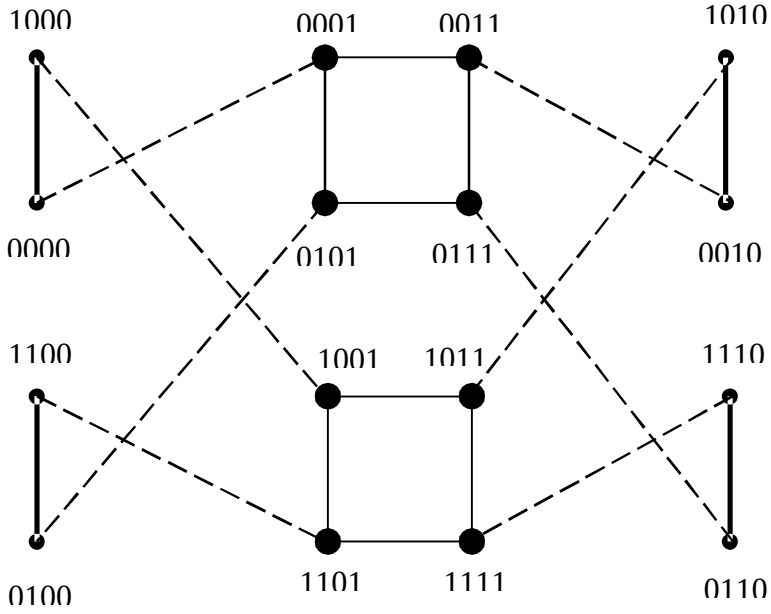


Figure 5.1 $EH(1,2)$

The dashed links correspond to $v_1 \oplus v_2 = 1$. The solid links correspond to $v_1[s+t:t+1] = v_2[s+t:t+1],\ H(v_1[t:1], v_2[t:1]) = 1,\ v_1[0] = v_2[0] = 1$ and the bold links to $v_1[t:1] = v_2[t:1],\ H(v_1[s+t:t+1], v_2[s+t:t+1]) = 1,\ v_1[0] = v_2[0] = 0$.

5.2.2 Structural Properties

(*Property* 5.1)

$EH(s,t)$ is isomorphic to $EH(t,s)$. This means *Exchanged Cube* is symmetric. $EH(s,t)$ can be decomposed into two copies of $EH(s-1,t)$ or $EH(s,t-1)$.

Let ∂T represent the smallest change in the number of network components (nodes or links) needed to increase the existing number of components T in a network while retaining its topological characteristics. $IE = \frac{\partial T}{T}$ measures the *incremental expandability* of the network. We use IE_{node} and IE_{link} to differentiate between node and link incremental expandabilities.

(*Property* 5.2)

$EH(s,t)$ has $s\,2^{s-1} + t\,2^{t-1} + 2^{s+t}$ links and 2^{s+t+1} nodes. Node incremental expandability is 1 and link incremental expandability is also approaching 1.

Proof. $$IE_{node} = \frac{\partial T_{node}}{T_{node}} = \frac{2^{s+t+1+1} - 2^{s+t+1}}{2^{s+t+1}} = 1,$$

$$IE_{link} = \frac{\partial T_{link}}{T_{link}} = \frac{(s+1)2^{s-1+1} + t2^{t-1} + 2^{s+t+1} - (s2^{s-1} + t2^{t-1} + 2^{s+t})}{s2^{s-1} + t2^{t-1} + 2^{s+t}} = \frac{(s+2)2^{s-1} + 2^{s+t}}{s2^{s-1} + t2^{t-1} + 2^{s+t}}$$

$$= \frac{(s+2)2^{-t-1} + 1}{s2^{-t-1} + t2^{-s-1} + 1} \to 1, \text{ as } s \to +\infty \text{ and/or } t \to +\infty .$$

(*Property* 5.3)

The number of links in $EH(s,t)$ is $\frac{1}{n+1}$ to $\frac{1}{2}$ of that of (*s*+*t*+1)-dimension *n*-cube B_{s+t+1}.

B_{s+t+1} has $(s+t+1)2^{s+t}$ links. The ratio of the number of links between $EH(s,t)$ and B_{s+t+1} can be evaluated in the following way:

Without loss of generality, suppose $s \geq t$, let $n = s+t$, $m = s-t$. Define

$$r = \frac{\text{number of links for } EH(s,t)}{\text{number of links for } B_{s+t+1}}$$

$$= \frac{s2^{s-1} + t2^{t-1} + 2^{s+t}}{(s+t+1)2^{s+t}}$$

$$= \frac{\frac{(n-m)}{2}2^{\frac{n-m}{2}} + \frac{(n+m)}{2}2^{\frac{n+m}{2}} + 2^n}{(n+1)2^n}$$

$$= \frac{\frac{(n-m)}{2}2^{\frac{-n-m}{2}} + \frac{(n+m)}{2}2^{\frac{m-n}{2}} + 1}{n+1}$$

To calculate the range of *r*, we have

$$\frac{\partial r}{\partial m} = \frac{1}{n+1}\left(-\frac{1}{2}2^{\frac{-n-m}{2}} + \frac{n-m}{2}2^{\frac{-n-m}{2}}\frac{-\ln 2}{2} + \frac{1}{2}2^{\frac{m-n}{2}} + \frac{n+m}{2}2^{\frac{m-n}{2}}\frac{\ln 2}{2}\right)$$

$$= \frac{1}{4(n+1)}2^{\frac{-n-m}{2}}(m2^{m+1}\ln 2 + (2^m - 1)(2 + (n-m)\ln 2))$$

As $2^m \geq 1$ for $m \geq 0$ and $n > m$, $\frac{\partial r}{\partial m} > 0$ for $m > 0$ and $\frac{\partial r}{\partial m}\big|_{m=0} = 0$.

It is easy to see that with a fixed *n*, *r* increases as *m* increases. So $r_{\min} = r\,|_{m=0} = \frac{n2^{-\frac{n}{2}}+1}{n+1}$,

which approaches $\frac{1}{n+1}$ when *n* is large enough. On the other hand, $r_{\max} = r|_{m=n-2}$ $= \frac{2^{-n+1} + \frac{n-1}{2}}{n+1}$, which approaches $\frac{1}{2} \cdot \frac{n-1}{n+1} \to \frac{1}{2}$ as *n* approaches infinity. In conclusion, $r \in (\frac{1}{n+1}, \frac{1}{2})$. A useful rule is that the smaller the difference between *s* and *t* is, the better is the proportion of links reduced.

(*Property* 5.4)

For 0-ending nodes, the node degree is $s+1$ while the node degree of 1-ending nodes is $t+1$.

Proof: This is obvious from the definition of *Exchanged Hypercube*.

(*Property* 5.5)

Routing in $EH(s,t)$ is straightforward. If source and destination differ in the leftmost *s* bits, then it must reach a 0-ending node from which the difference can be offset by routing in the subgraph of 0-ending nodes. If source and destination differ in the middle *t* bits, then it must reach a 1-ending node from which the difference can be offset by routing in the subgraph of 1-ending nodes. Which one is done first depends on the rightmost bit of source and destination. For example, in $EH(2,\ 2)$, if we want to go from 00000 to 10100, then we must use spare dimension 0 twice: $00000 \to 10000 \to 10001 \to 10101 \to 10100$. If we want to go from 00001 to 10101, then go: $00001 \to 00101 \to 00100 \to 10100 \to 10101$.

(*Property* 5.6)

The distance between each node pair is in [*H*, *H*+2], where *H* is their *Hamming* distance. According to *Property* 5.5, the detailed conclusion is listed in table 5.1. Suppose source is $s = a_{s-1} \cdots a_0 b_{t-1} \cdots b_0 c$ and destination is $d = a'_{s-1} \cdots a'_0\, b'_{t-1} \cdots b'_0\, c'$.

No.	$a_{s-1}\cdots a_0 = a'_{s-1}\cdots a'_0$	$b_{s-1}\cdots b_0 = b'_{s-1}\cdots b'_0$	c	c'	distance
1	Yes	Yes	any	any	H
2	Yes	No	0	0	$H + 2$
3	Yes	No	0	1	H
4	Yes	No	1	0	H
5	Yes	No	1	1	H
6	No	Yes	0	0	H
7	No	Yes	0	1	H
8	No	Yes	1	0	H
9	No	Yes	1	1	H+2
10	No	No	0	0	H+2
11	No	No	0	1	H
12	No	No	1	0	H
13	No	No	1	1	H+2

Table 5.1 Node distance in Exchanged Cube

For example, the 9th case means if $a_{s-1}\cdots a_0 \neq a'_{s-1}\cdots a'_0$, $b_{s-1}\cdots b_0 = b'_{s-1}\cdots b'_0$, $c = 1$ and $c' = 1$, then the distance between *s* and *d* is *H*+2, where *H* is the *Hamming* distance between *s* and *d*. The +2 is because it has to use dimension 0 (originally spare) twice: 1->0 and 0->1, for changing the first *s* bits. From Table 5.1, since for all rows in which distance equals *H*+2, *c* equals *c'* so $H \leq s + t$, the distance is no more than $s + t + 2$. For other rows, distance is $H \leq s + t + 1$. Thus, the diameter of $EH(s,t)$ is $s + t + 2$.

(*Property* 5.7)

$EH(s,t)$ is *Hamiltonian*, with a closed cycle encompassing all nodes only once.

We prove the property of *Hamiltonian* by induction on *s* and *t*. As $EH(s,t)$ is isomorphic to $EH(t,s)$, we only need to take induction on *s*. As a basis, we show that $EH(1,2)$ and $EH(2,\ 2)$ are *Hamiltonian* in Fig. 5.2 and 5.3 respectively.

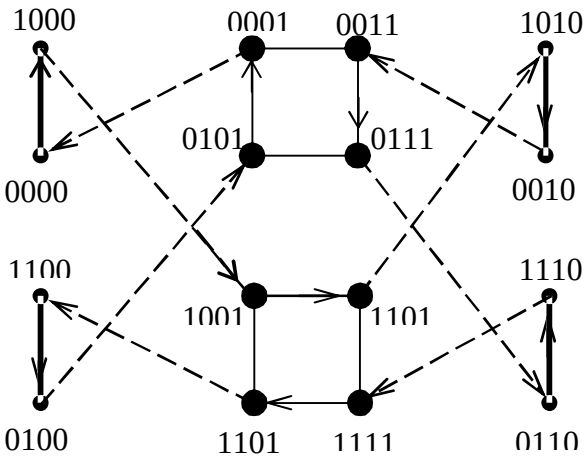


Figure 5.2

Hamiltonian cycle in $EH(1,\ 2)$

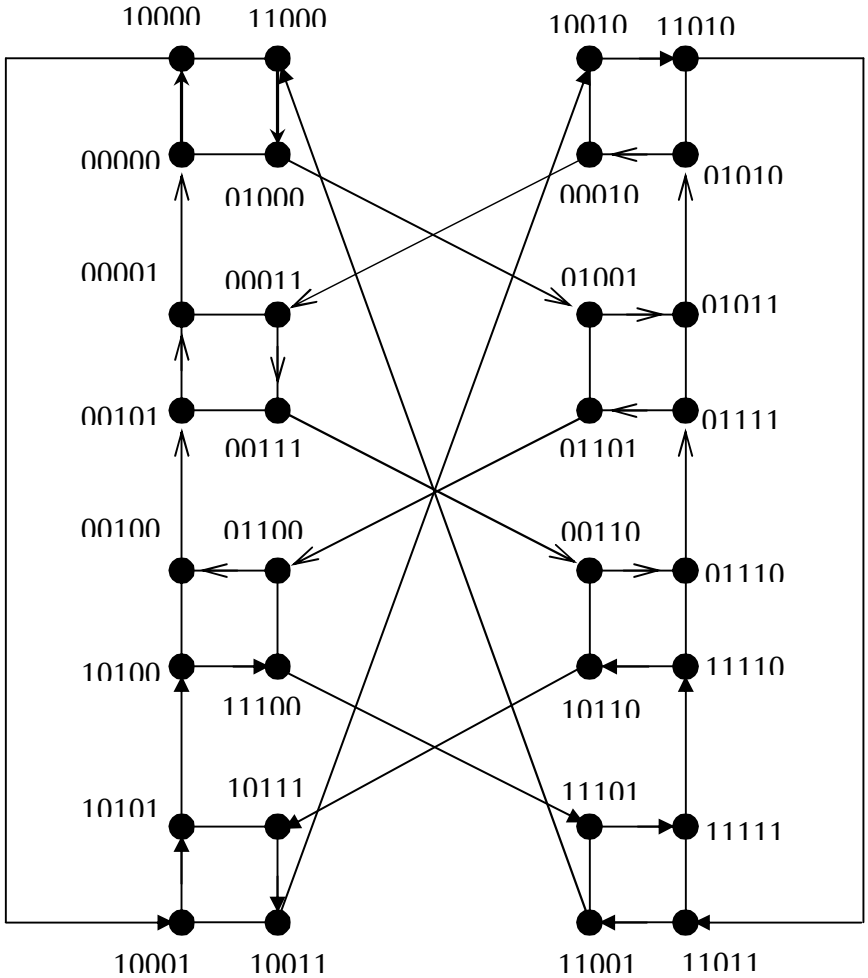


Figure 5.3

Hamiltonian cycle in $EH(2,\ 2)$

Assume that for $s \le k$ $(k \ge 1)$, $EH(s,t)$ is *Hamiltonian*. Then when $s = k+1$, we decompose $EH(k+1,t)$ into two subgraphs: $G_1(k,t)$ and $G_2(k,t)$.

$G_1(k,t) =< V_1(k,t), E_1(k,t) >$ where

$V_1(k,t) = \{0a_{k-1}a_{k-2}\cdots a_0 b_{t-1}b_{t-2}\cdots b_0 c \mid a_i, b_j, c \in \{0,\ 1\} \text{ for } i \in [0,k-1],\ j \in [0,t-1]\}$

$E_1(k,t) = \{(v_1, v_2) \in EH(k+1,t) \mid v_1, v_2 \in V_1(k,t)\}$

$G_2(k,t) =< V_2(k,t), E_2(k,t) >$ where

$V_2(k,t) = \{1a_{k-1}a_{k-2}\cdots a_0 b_{t-1}b_{t-2}\cdots b_0 c \mid a_i, b_j, c \in \{0,\ 1\} \text{ for } i \in [0,k-1],\ j \in [0,t-1]\}$

$E_2(k,t) = \{(v_1, v_2) \in EH(k+1,t) \mid v_1, v_2 \in V_2(k,t)\}$

Obviously, $G_1(k,t)$ and $G_2(k,t)$ are both isomorphic to $EH(k,t)$. Based on the induction assumption, there must be a *Hamiltonian* route R_1 in $G_1(k,t)$ from $u_1 = 010^{k+t}$ to $v_1 = 0^{k+t+2}$, where 0^l represents a sequence of 0s with length l $(l \ge 0)$. Similarly, there must also be a *Hamiltonian* route R_2 in $G_2(k,t)$ from $u_2 = 10^{k+t+1}$ to $v_2 = 110^{k+t}$.

As $e_1 = (v_1, u_2)$ and $e_2 = (v_2, u_1)$ are both edges in $EH(k+1,t)$, we now find a *Hamiltonian* cycle: $R_1 \| e_1 \| R_2 \| e_2$, where || denotes a concatenation operation. g

Actually, in previous Figure 5.3, the *Hamiltonian* cycle found in $EH(2,2)$ is constructed by the method in the proof. The path from 01000 to 00000 connected by $\longrightarrow$ is effectively R_1 and the path from 10000 to 11000 connected by $\longrightarrow$ is R_2. The two links represented by $\longrightarrow$ correspond to e_1 and e_2. R_1 and R_2 are mapped from the *Hamiltonian* cycle in $EH(1,2)$ demonstrated above.

5.3 Embedding other networks

(*Property* 5.8)

$B_s, B_t \subset EH(s,t) \subset B_{s+t+1}$. $EH(s,t)$ can also be decomposed into 2^s B_t and 2^t B_s simultaneously. The 0-ending nodes (denoted as $V_s(EH(s,t))$) together with the links connecting in between (denoted as $E_s(EH(s,t))$) comprise 2^t *s*-dimension binary hypercubes (denoted as $B_s(EH(s,t))$ collectively), while 1-ending nodes ($V_t(EH(s,t))$) together with the links connecting in between ($E_t(EH(s,t))$) comprise 2^s *t*-dimension binary hypercubes ($B_t(EH(s,t))$). And links in $E_0(EH(s,t)) = \{(v_1, v_2) \mid v_1 \oplus v_2 = 1\}$ span between these two classes of binary hypercubes.

(*Property* 5.9)

$EH(s,s)$ is isomorphic to $GC(2s+1,\ 2)$. $EH(s,s-1)$ is isomorphic to $GC(2s,\ 2)$. Here $GC(n,M)$ stands for a *Gaussian Cube*. For $GC(n,2^a)$, it can embed simultaneously $\{ EH(s_k,t_k) \mid k \in [0,2^a) \}$, where

$$s_k = \left\lfloor \frac{n-k-1}{2^a} \right\rfloor + 1 - d(k,a),\ t_k = n - a - s_k,\ d(k,a) = k < a\ ?1:0.$$

This property will be proved in the following *Chapter* 6, which is about *Gaussian Cube*.

Applications emulation performance is a measure of how efficiently an application expressed as a *guest network* may be represented or *mapped* onto a *host network*. The embedding results demonstrate two important factors: the computational equivalence (or non-equivalence) between networks of different topology and the efficiency of the simulation of algorithms designed for the guest network on the host network [56].

(*Definition* 5.2)

Let the guest and host networks be denoted as $G_g = (V_g, E_g)$ and $G_h = (V_h, E_h)$, respectively. An *injective* embedding of G_g onto G_h is a one-to-one mapping that assigns every node and edge of G_g to a node and path, respectively, of G_h. Given an embedding, the *dilation* is the maximum distance in G_h between two adjacent nodes in G_g. The *expansion* is the smallest number of nodes in G_h that is required to map all the nodes in G_g. *Loading* is the maximum number of nodes in G_g mapped to the same node in G_h while *congestion* is the maximum number of edges in G_g mapped to the same edge in G_h.

For optimal embedding, dilation = expansion = loading = congestion = 1.

(*Property* 5.10)

$EH(s,t)$ can optimally embed a ring network of the same size.

Proof: This property is ensured by *Property* 5.7 that $EH(s,t)$ is *Hamiltonian*.

(*Property* 5.11)

$EH(s,t)$ can embed a mesh of size $2^{s-1} \times 2^{t+1}$ or $2^{s+1} \times 2^{t-1}$ with dilation 3, expansion 2, loading and congestion 1, or with dilation 2, expansion 2, loading 1 and congestion 2.

Before presenting the strategy for embedment, we first define a subgraph $GM(2^{s+2}, 2^{t-1})$ of $EH(s,t)$ by removing part of its links. $GM(2^{s+2}, 2^{t-1})$ is like a mesh, though two intermediate nodes may be inserted between two neighboring nodes in the same column of the mesh. The Figure 5.4 below demonstrates $GM(16, 2)$ and how a $2^1 \times 2^3$ mesh is

embedded into it. The nodes with double cycle are images of the guest network: mesh of 2×8.

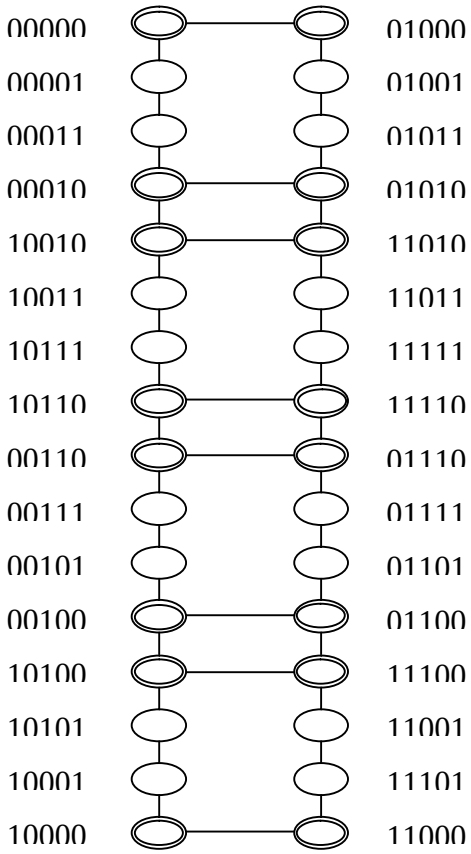


Figure 5.4 $GM(16,2)$ and how $2^1\times 2^3$ mesh is embedded into $EH(2,2)$

The procedure of constructing $GM(2^{s+2},2^{t-1})$ is as follows. Since *n*-dimension binary hypercube B_n is *Hamiltonian*, there is a sequence of node address in B_{s-1} and B_t such that the *Hamming* distance between neighboring addresses is 1. Denote the sequence as $\{a_0,a_1,\cdots,a_{2^{s-1}-1}\}$ and $\{b_0,b_1,\cdots,b_{2^t-1}\}$. Then the first row of $GM(2^{s+2},2^{t-1})$ is: $0a_0b_00,\ 0a_1b_00,\cdots,\ 0a_{2^{s-1}-1}b_00$, where $0a_ib_00$ means concatenating 0, a_i, b_0 and 0. They are all connected to $c_i=0a_ib_01$ respectively. But c_i and c_{i+1} ($i\in[0,2^{s-1}-2]$) are not neighbors, though they are all neighbored by $d_i=0a_ib_11$, which is in turn neighbored by

$e_i = 0a_ib_10$. Now, e_i and e_{i+1} are connected for $i \in [0,2^{s-1}-2]$. They form the second horizontally connected row of $GM(2^{s+2},2^{t-1})$. Moreover, e_i has a link to $f_i = 1a_ib_10$, which are also sequentially connected and form the third row of the mesh. This process continues on until the $2^{t+1}{}_{th}$ row is formed.

It is obvious that a $2^s \times 2^t$ mesh can be embedded into $GM(2^{s+1},2^t)$ and $GM(2^s,2^{t+1})$. For example, the embeddings of 4×8 mesh into $GM(8,8)$ with dilation 2, expansion 2, loading 1 and congestion 2, and with dilation 3, expansion 2, loading 1 and congestion 1 are shown in Fig. 5.5 and Fig. 5.6 respectively. The nodes represented by ✱ serve as images of the guest mesh, while the ● stands for those nodes in the host network that are not images of any node in the guest mesh.

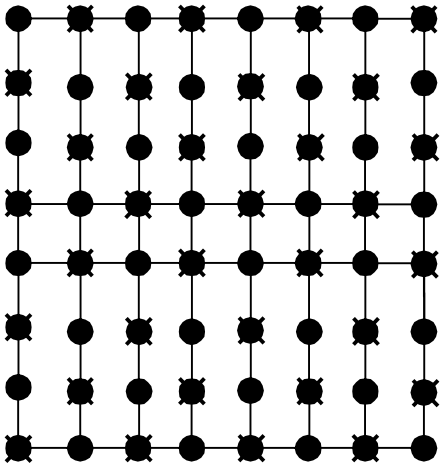


Figure 5.5 Embedment with dilation 2, expansion 2, loading 1 and congestion 2

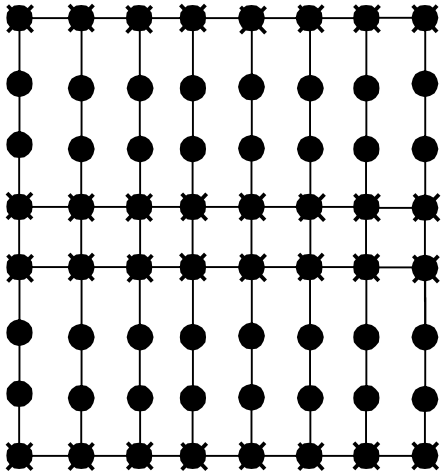


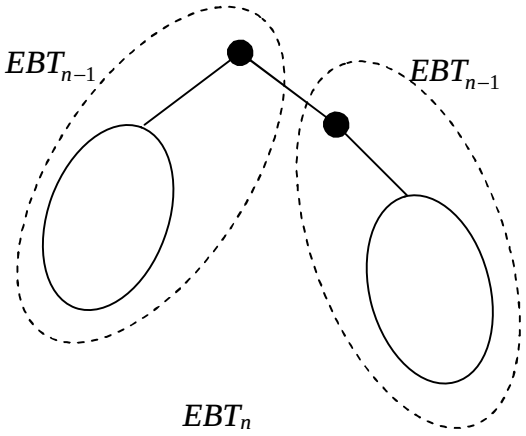
Figure 5.6 Embedment with dilation 3, expansion 2, loading 1 and congestion 1

5.4 Extended Binomial Tree

In binary hypercubes, many applications such as broadcasting, prefix sum computing and load balancing can be solved with the aid of *Binomial Trees* (special spanning trees of hypercube). For the *Exchanged Cube*, we introduce the *Extended Binomial Tree*. It is very similar to the *Binomial Tree*, with only a small change in the initial condition. It is proved later that the labeled form of *Extended Binomial Tree*, *Exchanged Tree*, preserves many desirable properties of *Binomial Tree*.

(*Definition* 5.3) *Extended Binomial Tree*

Extended Binomial Tree is defined by induction. For $n \geq 3$, an *Extended Binomial Tree of dimension n* (EBT_n) is formed by two copies of EBT_{n-1}, where the root of one EBT_{n-1} (randomly chosen) becomes the root of EBT_n and root of the other EBT_{n-1} becomes the child of the root of the former EBT_{n-1}. EBT_2 and EBT_3 are defined in Figure 5.7:



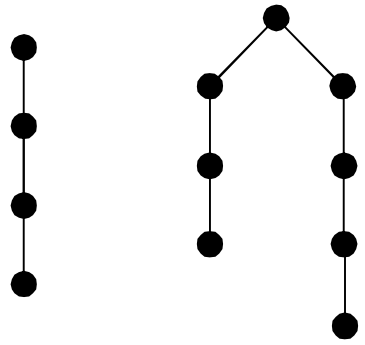


Figure 5.7 EBT_2 and EBT_3

Extended Binomial Tree maintains several good properties of *Binomial Tree* as follows:

(*Property* 5.12)

There are 2^n nodes in EBT_n for $n \geq 2$. This can be simply proved by induction on *n*.

(*Property* 5.13)

The height of EBT_n is $n+1$ for $n \geq 2$.

(*Property* 5.14)

For $n \geq 2$, the root of EBT_n has degree $n-1$, which is the largest among all nodes.

(*Property* 5.15)

In EBT_n ($n \geq 2$), there are exactly $a_n^i = C_{n-1}^i + C_{n-1}^{i-2}$ nodes at depth *i* for *i* = 0, 1, 2, …., $n+1$. Here, $C_n^m = \frac{n!}{m!(n-m)!}$, where $n! = n(n-1)\cdots 1$, for $m \in [1,n]$, $m, n \in N$. $C_n^0 = 1$ for all $n \in N \bigcup \{0\}$. For all other cases, $C_n^m = 0$. The depth of root is 0.

Proof: (By induction on *n*)

For $n = 2,\ 3$, this proposition is true based on the Figure 5.7 above. It is easy to see that due to the construction of EBT_n, for all $n \geq 2$, $a_n^0 = 1$, $a_n^{n+1} = 1$, $a_n^i = a_{n-1}^{i-1} + a_{n-1}^i$ for $i \in [1,n]$. Suppose the proposition is true for $n \leq k$ $(k \geq 2)$. Then when $n = k+1$, for $i \in [1,\ k+1]$:

$$
\begin{aligned}
a_{k+1}^i &= a_k^{i-1} + a_k^i = a_k^{i-1} + (a_{k-1}^i + a_{k-1}^{i-1}) = a_k^{i-1} + a_{k-1}^{i-1} + (a_{k-2}^i + a_{k-2}^{i-1}) = \cdots \\
&= a_{i-1}^{i-1} + a_{i-1}^i + a_i^{i-1} + a_{i+1}^{i-1} + \cdots + a_k^{i-1} \\
&= (C_{i-2}^{i-1} + C_{i-2}^{i-3}) + (C_{i-2}^i + C_{i-2}^{i-2}) + (C_{i-1}^{i-1} + C_{i-1}^{i-3}) + \cdots + (C_{k-1}^{i-1} + C_{k-1}^{i-3}) \\
&= (C_{i-1}^{i-1} + C_i^{i-1} + \cdots + C_{k-1}^{i-1}) + (C_{i-3}^{i-3} + C_{i-2}^{i-3} + \cdots + C_{k-1}^{i-3}) \quad (C_{i-3}^{i-3} = C_{i-2}^{i-2}) \\
&= C_k^i + C_k^{i-2}.
\end{aligned}
$$

The last step used the conclusion that:

$$
\begin{aligned}
& C_n^n + C_{n+1}^n + C_{n+2}^n + \cdots + C_{n+k}^n \\
= \; & C_{n+1}^{n+1} + C_{n+1}^n + C_{n+2}^n + \cdots + C_{n+k}^n \\
= \; & C_{n+2}^{n+1} + C_{n+2}^n + C_{n+3}^n + \cdots + C_{n+k}^n \\
= \; & C_{n+3}^{n+1} + C_{n+3}^n + C_{n+4}^n + \cdots + C_{n+k}^n \\
= \; & \cdots\cdots \\
= \; & C_{n+k}^{n+1} + C_{n+k}^n \\
= \; & C_{n+k+1}^{n+1} \qquad (n \geq 1, \; k \geq 0) \qquad \text{g}
\end{aligned}
$$

This property shows that the name *Extended Binomial Tree* is justified for the tree constructed here.

(*Property* 5.16)

For $n \geq 3$, EBT_n embeds the *Binomial Tree* of order $n-1$, with *dilation* 1, *congestion* 1, *load* 1 and *expansion* 2.

In the following, we introduce *Exchanged Tree* $ET(s,t)$, *which* is actually a labeled *Extended Binomial Tree.*

(*Definition* 5.4): *Exchanged Tree*

Exchanged Tree $ET(s,t)$ is constructed by the following sequence*:*

$ET(1,\ 1) \to ET(1,\ 2) \to ET(1,\ 3) \to \cdots$

$\to ET(1,\ t) \to ET(2,\ t) \to \cdots ET(s,t)$.

$ET(1,\ 1)$ is demonstrated in Figure 5.8:

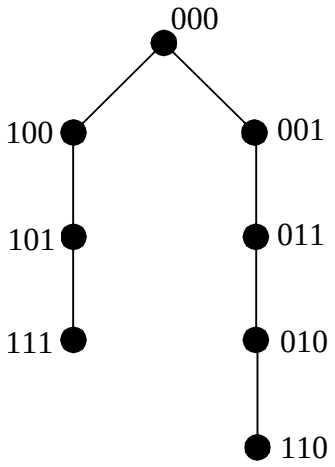


Figure 5.8 $ET(1,\ 1)$

Given $ET(1,\ t)$, $ET(1,\ t+1)$ is defined as:

Let G_1 and G_2 be two $ET(1,\ t)$s. We re-label G_1 by inserting one 0 between the left first and second bits of original node labels. Formally, it is a mapping f_1^1:

$a_0 b_{t-1} b_{t-2} \cdots b_0 c \rightarrow a_0 0 b_{t-1} b_{t-2} \cdots b_0 c$.

Then re-label G_2 by inserting one 1 between the left first and second bit. Formally, it is a mapping f_1^2: $a_0 b_{t-1} b_{t-2} \cdots b_0 c \rightarrow a_0 1 b_{t-1} b_{t-2} \cdots b_0 c$.

Finally, make the root of G_2 the rightmost son of G_1's root. $ET(1,\ 2)$ is illustrated in Figure 5.9.

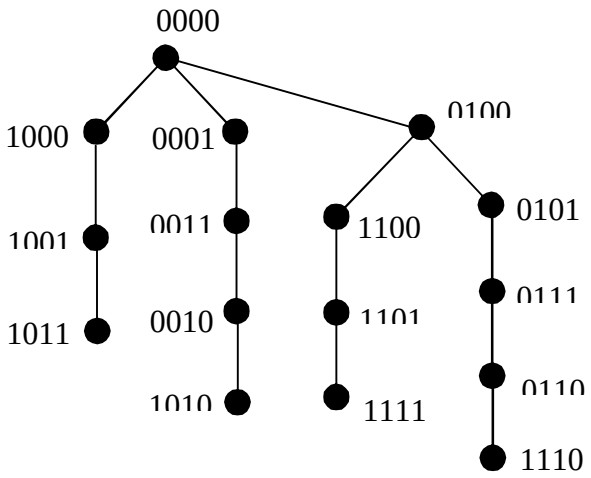


Figure 5.9 $ET(1,\ 2)$

Given $ET(s,\ t)$, $ET(s+1,\ t)$ is defined by:

Let G_1 and G_2 are two $ET(s,\ t)$s. We re-label G_1 by adding one 0 to the leftmost bit.

Formally, it is a mapping f_2^1: $a_{s-1} a_{s-2} \cdots a_0 b_{t-1} b_{t-2} \cdots b_0 c \rightarrow 0 a_{s-1} a_{s-2} \cdots a_0 b_{t-1} b_{t-2} \cdots b_0 c$.

Then re-label G_2 by adding one 1 to the leftmost bit. Formally, it is a mapping f_2^2:

$a_{s-1}a_{s-2}\cdots a_0 b_{t-1}b_{t-2}\cdots b_0 c \to 1a_{s-1}a_{s-2}\cdots a_0 b_{t-1}b_{t-2}\cdots b_0 c$.

Finally, make the root of G_2 the rightmost child of G_1's root. $ET(2,\ 2)$ is demonstrated in Fig. 5.10.

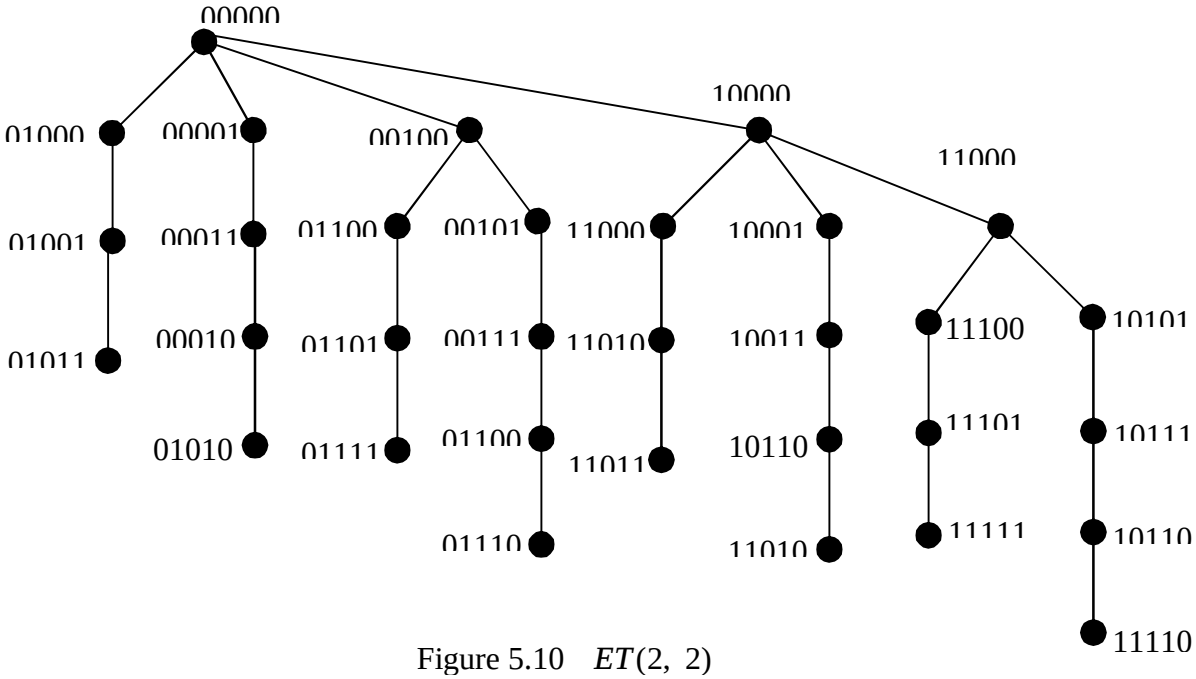


Figure 5.10 $ET(2,\ 2)$

Based on the procedure of constructing *Extended Binomial Tree* $ET(s,\ t)$, it is obvious that $ET(s,\ t)$ is an *Extended Binomial Tree* EBT_{s+t+1}. So it inherits all good properties of EBT_{s+t+1}. However, it also has some additional properties related to $EH(s,t)$.

(*Property* 5.17)

For all $s,t \geq 1$, the root of $ET(s,\ t)$ is 0^{s+t+1}. This is guaranteed by the procedure of construction.

(*Property* 5.18)

$ET(s,\ t)$ is a spanning tree of $EH(t,s)$.

Proof:

This property can be proved by induction. Firstly, it is obvious that all nodes in $EH(t,s)$ are covered by $ET(s,\ t)$. Then due to *Property* 5.17 and the fact that 0^{s+t+1} and 10^{s+t} are neighbors, it is guaranteed that each edge in $ET(s,\ t)$ is also in $EH(t,s)$.

(*Property* 5.19)

Suppose the pre-order of $ET(s,\ t)$ is $\{a_0, a_1, \cdots, a_{2^{s+t+1}-1}\}$.

Define $b_i = a_i$ AND $1^{s-1}01^{t-1}00$. Then, $\{b_0, b_1, \cdots, b_{2^{s+t+1}-1}\}$ is non-decreasing.

Proof:

Recall the sequence of construction:

$ET(1,\ 1) \to ET(1,\ 2) \to ET(1,\ 3) \to \cdots \to ET(1,\ t) \to ET(2,\ t) \to \cdots ET(s,t)$. When constructing $ET(1,\ t+1)$ from $ET(1,\ t)$, we place the new graph built by adding 1 the rightmost son of the root of its counterpart, which is built by adding 0. When constructing $ET(s+1,\ t)$ from $ET(s,\ t)$, the rule is followed too. These facts ensure this property of order. Masking three bits is due to the initial condition.

Property 5.19 provides a good way of routing in $ET(s,\ t)$. Suppose the source is *s* and the destination is *d*. We first find a path to $x = s$ *AND* $1^{s-1}01^{t-1}00$. This is simple because it is equivalent to routing in small-scaled $ET(1,\ 1)$, which can be accomplished by rote. Then from *x*, it is easy to find a path to $y = d$ *AND* $1^{s-1}01^{t-1}00$. Thanks to the ordering property, this is equivalent to routing in a *Binomial Tree*. Finally, a path is found from *y* to *d*.

5.5 Fault-tolerant routing in Exchanged Hypercube

We now present a fault-tolerant routing algorithm in $EH(s,t)$. It categorizes faulty components so as to produce a better result than tolerating merely as many faults as node availability. This approach is also applicable to the class of hypercube variants formed by link dilution.

As stated above, there are 2^t s-dimension binary hypercubes embedded in $EH(s,t)$. They are denoted as $B_s(EH(s,t))$ collectively. More specifically, for any $k \in [0,2^t)$, denote as $B_s(EH(s,t),k)$ the binary hypercube whose nodes comprise the following set: $V_s(EH(s,t),k) = \{a_{s-1}\cdots a_0 b_{t-1}\cdots b_0 0 \mid \overline{b_{t-1}\cdots b_0} = k,\ a_i, b_j \in \{0,1\}\ i \in [0,s),\ j \in [0,t)\}$. If $x \in V_s(EH(s,t),k)$ and $\overline{x[s+t:t+1]} = p$, we denote such nodes as $V_s(EH(s,t),k,p)$.

Likewise, there are 2^s t-dimension binary hypercubes embedded in $EH(s,t)$. They are denoted as $B_t(EH(s,t))$ collectively. $B_t(EH(s,t),l)$ is defined as the hypercube whose nodes are composed of: $V_t(EH(s,t),l) = \{a_{s-1}\cdots a_0 b_{t-1}\cdots b_0 1 \mid \overline{a_{s-1}\cdots a_0} = l,\ a_i, b_j \in \{0,1\}$ $i \in [0,s),\ j \in [0,t)\}$ $(l \in [0,2^s))$. If $x \in V_t(EH(s,t),l)$ and $\overline{x[t:1]} = q$, we denote such node as $V_t(EH(s,t),l,q)$. Obviously, $V_s(EH(s,t),k,p)[s+t:1] = V_t(EH(s,t),p,k)[s+t:1]$.

Suppose there are F_s faulty components in $B_s(EH(s,t))$, and F_t faulty components in $B_t(EH(s,t))$. Let $E_0(EH(s,t)) = \{(v_1,v_2) \in EH(s,t) \mid v_1\ XOR\ v_2 = 1\}$. Suppose there are F_0 faulty links in $E_0(EH(s,t)) \setminus \{(v_1,v_2) \in EH(s,t) \mid v_1$ or v_2 is faulty$\}$. We have:

(*Theorem* 5.1)

If $F_s + F_0 < s$ and $F_t + F_0 < t$, there is a deadlock-free and livelock-free routing algorithm that can deliver messages from a nonfaulty source *r* to a nonfaulty destination *d* in no more than $H(r,d) + 2(F_s + F_t) + 2$ hops.

This theorem is evident from the following algorithm:

(Algorithm 5.1) **F**ault-tolerant **R**outing in $EH(s,t)$ (*FREH*)

(Case I)

Suppose $r = B_s(EH(s,t), k_0, l_1)$ and $d = B_t(EH(s,t), l_0, k_1)$. Since $F_s + F_0 < s$, it is affordable to communicate within each $B_s(EH(s,t), k)$ in the initialization phase, so that each node in it knows and records the set of nodes in $B_s(EH(s,t), k)$ whose link in $E_0(EH(s,t))$ (i.e. in dimension 0) is faulty.

In one case, if *r* finds that $B_s(EH(s,t), k_0, l_0)$'s link in dimension 0 is non-faulty, it sends the packet within $B_s(EH(s,t), k_0)$ to $B_s(EH(s,t), k_0, l_0)$. This is guaranteed to succeed because $F_s(k_0) < s$ and there are a lot of existing deadlock-free and livelock-free routing algorithms (including my FTFR) that work well in *s*-dimension hypercube in the face of no more than $s-1$ faulty components. After that, $B_s(EH(s,t), k_0, l_0)$ sends the packet to $B_t(EH(s,t), l_0, k_0)$ via the link in dimension 0. Finally, the packet is sent in $B_t(EH(s,t), l_0)$ to $B_t(EH(s,t), l_0, k_1)$, which is guaranteed by $F_t + F_0 < t$.

In the other case, if by looking up its local table, *r* finds that the 0-dimension link of $B_s(EH(s,t), k_0, l_0)$ is faulty, then there must be a nonfaulty neighbor of *r* whose 0-dimension link is also nonfaulty. This is guaranteed by $F_s + F_0 < s$. Denote it as

$B_s(EH(s,t),k_0,l_2)$. So the packet is sent to $B_s(EH(s,t),k_0,l_2)$, which in turn, sends the packet to $B_t(EH(s,t),l_2,k_0)$. Now there must be a nonfaulty neighbor of $B_t(EH(s,t),l_2,k_0)$ in $B_t(EH(s,t),l_2)$ whose 0-dimension link is also nonfaulty. If there is such a neighbor in preferred dimension, then use it. Otherwise, use the spare dimension and mask it so that it will not be used again. After going back to $B_s(EH(s,t))$, the process above repeats and finally the packet reaches *d*.

Obviously, deadlock-freeness is still guaranteed. Since faulty components might cause the use of a spare dimension, which brings about for and pro between $B_t(EH(s,t))$ and $B_s(EH(s,t))$, the number of hops is bounded by $H(r,d)+2(F_s+F_t)$.

(Case II)

If $r=B_t(EH(s,t),l_0,k_1)$ and $d=B_s(EH(s,t),k_0,l_1)$, due to the symmetricalness of *Exchanged Hypercube*, the algorithm is the same as case I.

(Case III)

Suppose $r=B_s(EH(s,t),k_0,l_0)$ and $d=B_s(EH(s,t),k_1,l_1)$. If $k_1=k_0$, then it is routing in *s*-dimension binary hypercube. Otherwise, the packet is sent to $B_t(EH(s,t),k_0)$ via the 0-dimension link of *r* or one of its neighbors in $B_s(EH(s,t),k_0)$. Then the problem is the same as in case I. But now, the number of hops is bounded by $H(r,d)+2(F_s+F_t)+2$.

(Case IV)

Suppose $s=B_t(EH(s,t),l_0,k_0)$ and $d=B_t(EH(s,t),l_1,k_1)$.

This case is handled in the same way as in case III. g

Apart from the initialization cost $O(\max(s,t)) < O(n)$, the algorithm is run at time cost $O(1)$ and message overhead $O(n)$. The most important thing is that both node faults and link faults (including those spanning in dimension 0) can be tolerated.

Chapter 6 A Fault-Tolerant Routing Strategy for Gaussian Cube Using Gaussian Tree

6.1 Introduction

Gaussian Cubes (*GC*s) is a family of interconnection networks parameterized by a modulus M and a dimension n [1][2]. Their desirable scalability makes possible generalized analysis of interconnection cost, routing performance, and reliability. Besides, such communication primitives as unicasting, multicasting, broadcasting/ gathering [7] can also be done rather efficiently in all *GC*s [1]. However, although research achievements abound in routing in binary hypercubes, there are no existing fault-tolerant routing strategies for *GC*s or for node/link dilution cubes. One of the difficulties lies in the low network node availability (maximum number of faulty neighbors of a node that can be tolerated without disconnecting the node from the network). Thus, if the topology is fixed, new methods have to be employed to tackle this intrinsic problem.

In this chapter, we present a new routing algorithm based on a new topology called *Gaussian Tree* (*GT*). In $GC(n,2^a)$, *GT* is dependent only on a and divides all the nodes in $GC(n,2^a)$ into 2^a classes according to their least significant a bits. So the original problem is converted into first routing in *GT* (i.e. between different classes) and then routing in one such class. The former is facilitated by the definite and predictable routing in trees while the latter is actually routing in ordinary binary hypercube. Faults encountered in different stages of this divide-and-conquer strategy lead to a new categorization of faulty components, which enables to analyze the routing strategy in the presence of far more faults than the network node availability. The encouraging result is

demonstrated in the chapter. Methodologically speaking, this approach also opens window to a brand-new way of analyzing network reliability, which is especially valuable for incomplete networks.

The characteristics of our routing strategy for $GC(n,\ 2^a)$ encompasses:

1) Incurs message overhead of only $O(n)$.

2) The computation complexity for intermediate nodes is at most $O(a(n-a)\log a)$.

3) Guarantees a message path length not exceeding $2F$ longer than the optimal path found in a fault-free setting, provided the distribution of faulty components in the network satisfies the precondition of *Theorem* 6.3 and *Theorem* 6.4.

4) Each node requires at most $\left\lceil \frac{n-1}{2^a} \right\rceil + 1$ rounds of fault status exchange with its neighbors.

5) Each node maintains and updates at most F n-bit node addresses, where F is the number of faults related to nodes whose least significant a bits are same as the current node.

6) Generates deadlock-free and livelock-free routes.

7) The number of faulty components tolerable is presented in *Fig*. 6.6 and *Theorem* 6.4.

The chapter is organized as follows. Preliminaries are given in Section 6.2 to provide an equivalent definition of *GC* that facilitates the following discussion. Section 6.3 defines *GT*. The routing algorithm for the fault-free *GC* is described in Section 6.4 separately to make the subsequent section clearer. In Section 6.5, the fault-tolerant routing strategy that deals with all categories of faults is studied.

6.2 Preliminaries

6.2.1 Original Definition

(*Definition* 6.1) The binary *Gaussian Cube* is denoted by $GC(n, M)$ [1][2], where n (network dimension) ≥ 0 and M (modulus) ≥ 1. It has 2^n nodes that are identified with unique n-bit labels. A link connects two nodes p and q if the following conditions are both true:

1) The labels of p and q differ in the c^{th} bit for some c, $0 \leq c \leq (n-1)$.
2) p and q are in the congruence class $[c]_{M'}$, where $M' = \min\{2^c, M\}$.

The congruence class of c modulo M, $[c]_M$, is the set $\{kM + c \mid k \in Z\}$, where Z represents the set of integer.

6.2.2 Transformation:

According to *Definition* 6.1, if node $p = a_{n-1}a_{n-2}\cdots a_c \cdots a_1 a_0$ ($a_i \in \{0, 1\}$ for $i \in [0, n-1]$) has a link to $q = a_{n-1}a_{n-2}\cdots \overline{a_c} \cdots a_1 a_0$, then there must exist k_1 and k_2, such that

$$\overline{a_{n-1}a_{n-2}\cdots a_c \cdots a_1 a_0} = k_1 M' + c \qquad (1)$$

$$\overline{a_{n-1}a_{n-2}\cdots \overline{a_c} \cdots a_1 a_0} = k_2 M' + c \qquad (2)$$

(1) – (2) and take absolute value on both sides, we get:

$$2^c = |k_1 - k_2| M' \qquad (3)$$

Therefore, M' must be the power of 2. Since $M' = \min\{2^c, M\}$, M must also be the power of 2 if $M < 2^c$. We examine two cases.

1. M is not power of 2. If $M \geq 2^{n-1}$, since $c \leq n-1$ and M' = min $\{2^c, M\} = 2^c$, it makes no difference to the original network if we set $M = 2^{n-1}$. So we assume $M < 2^{n-1}$. In this case, there will be no link spanning in dimension c, where c is larger than $\lfloor \log M \rfloor$. Effectively, the network is separated into $2^{n-1-\lfloor \log M \rfloor}$ disconnected subnetworks, with each combination of the first $n-1-\lfloor \log M \rfloor$ bits representing one such subnetwork. Formally, $GC(n, M) = \bigcup_{i=0}^{2^{n-1-\lfloor \log M \rfloor}-1} G_i$. Each subnetwork G_i is composed of $<V_i, E_i>$, where

$$V_i = \{a_{n-\lfloor \log M \rfloor -2} \cdots a_i \cdots a_0 b_{\lfloor \log M \rfloor} \cdots b_j \cdots b_0 \mid b_j \in \{0,1\},\ 0 \leq j \leq \lfloor \log M \rfloor,\ \overline{a_{n-\lfloor \log M \rfloor -2} \cdots a_i \cdots a_0} = i\}$$

$E_i = \{(v_1, v_2) \in E \mid v_1 \in V_i, v_2 \in V_i\}$, where E is the set of edges in the original network. Obviously, for $\forall i, j \in [0, 2^{n-1-\lfloor \log M \rfloor})$ and $i \neq j$, $V_i \cap V_j = \Phi$, $E_i \cap E_j = \Phi$. So routing can be done within the subnetwork G_i if the source and destination both belong to G_i, or fails otherwise. Furthermore, as G_i is isomorphic to $GC(\lfloor \log M \rfloor + 1,\ 2^{\lfloor \log M \rfloor})$, this situation is covered in the following case, where M is power of 2.

2. M is power of 2. Denote $\alpha = \log_2 M$, and $\alpha \in Z$. We have the following theorem, which can be viewed an equivalent definition of *Gaussian Cube*.

(*Theorem* 6.1)

In $GC(n, 2^a)$, node $p = a_{n-1} a_{n-2} \cdots a_c \cdots a_1 a_0$ ($a_i \in \{0, 1\}$ for $i \in [0, n-1]$) has a link in dimension c ($c \in [1, n-1]$) if and only if:

$$\begin{cases} \overline{a_{a-1} a_{a-2} \cdots a_0} = c \% 2^a & \text{if } c \in (a, n) \\ \overline{a_{c-1} a_{c-2} \cdots a_0} = c & \text{if } c \in [1, a] \end{cases}$$

where x%y represents the modulus of x divided by y, like in *C/C++*. And each node has a link in dimension 0.

Proof: We prove *Theorem* 6.1 by considering three cases.

(Case I) $c \in (a, n)$.

(Necessary)

According to Equation (1), $\overline{a_{n-1}a_{n-2}\cdots a_a \cdots a_1 a_0} = k_1 M + c$.

Thus, $\overline{a_{n-1}a_{n-2}\cdots a_{a+1}a_a} \cdot 2^a + \overline{a_{a-1}a_{a-2}\cdots a_0} = k_1 \cdot 2^a + c$.

Take the modulus of 2^a on both sides and due to the fact that $\overline{a_{a-1}a_{a-2}\cdots a_0} < 2^a$, we obtain $\overline{a_{a-1}a_{a-2}\cdots a_0} = c \% 2^a$.

(Sufficient)

If $\overline{a_{a-1}a_{a-2}\cdots a_0} = c \% 2^a$, then $\overline{a_{n-1}a_{n-2}\cdots a_a a_{a-1}\cdots a_1 a_0} - c$ can be wholly divided by 2^a. Define $k_1 = \dfrac{\overline{a_{n-1}a_{n-2}\cdots a_c \cdots a_1 a_0} - c}{2^a} \in Z$ and

$$k_2 = \begin{cases} k_1 + 2^{c-a} & \text{if } a_c = 0 \\ k_1 - 2^{c-a} & \text{if } a_c = 1 \end{cases}$$

Then, $\overline{a_{n-1}a_{n-2}\cdots a_c \cdots a_1 a_0} = k_1 \cdot 2^a + c = k_1 \cdot \min(2^c, 2^a) + c = k_1 \cdot M' + c$

and $\overline{a_{n-1}a_{n-2}\cdots \overline{a_c} \cdots a_1 a_0} = k_2 \cdot 2^a + c = k_2 \cdot \min(2^c, 2^a) + c = k_2 \cdot M' + c$

In other words, according to the original definition, $a_{n-1}a_{n-2}\cdots a_c \cdots a_1 a_0$ has a link in dimension *c*.

(Case II) $c \in [1, a]$.

(Necessary)

According to Equation (1), $\overline{a_{n-1}a_{n-2}\cdots a_c \cdots a_1 a_0} = k_1 \cdot 2^c + c$.

Thus, $\overline{a_{n-1}a_{n-2}\cdots a_{c+1}a_c} \cdot 2^c + \overline{a_{c-1}a_{c-2}\cdots a_0} = k_1 \cdot 2^c + c$.

By taking the modulus of 2^c on both sides and due to the fact that $\overline{a_{c-1}a_{c-2}\cdots a_0} < 2^c$ and $c < 2^c$ for $c \geq 1$, we obtain $\overline{a_{c-1}a_{c-2}\cdots a_0} = c$.

(Sufficient)

If $\overline{a_{c-1}a_{c-2}\cdots a_0} = c$, then $\overline{a_{n-1}a_{n-2}\cdots a_c \cdots a_1 a_0} - c$ can be wholly divided by 2^c.

Define $k_1 = \dfrac{\overline{a_{n-1}a_{n-2}\cdots a_c \cdots a_1 a_0} - c}{2^c} \in Z$ and

$$k_2 = \begin{cases} k_1 + 1 & \text{if } a_c = 0 \\ k_1 - 1 & \text{if } a_c = 1 \end{cases}$$

Then, $\overline{a_{n-1}a_{n-2}\cdots a_c \cdots a_1 a_0} = k_1 \cdot 2^c + c = k_1 \cdot \min(2^c, 2^a) + c = k_1 \cdot M' + c$

and $\overline{a_{n-1}a_{n-2}\cdots \overline{a_c} \cdots a_1 a_0} = k_2 \cdot 2^c + c = k_2 \cdot \min(2^c, 2^a) + c = k_2 \cdot M' + c$

In other words, according to the original definition, $a_{n-1}a_{n-2}\cdots a_c \cdots a_1 a_0$ has a

(Case III) $c = 0$.

For any $M \geq 1$, M' = min $\{2^c, M\}$ = 1. For any integer *p* and *q*, they must be in the congruence class $[c]_{M'} = [0]_1$. So each node has a link in dimension 0.

Since the case of *M* not being the power of 2 can be solved once we have a routing strategy for *M* being power of 2, in this paper, we only discuss the latter situation, i.e. assuming $\alpha = \log_2 M \in Z$.

6.3 *Gaussian Tree*

According to *Theorem* 6.1, we can see that whether a packet can be forwarded through dimension *c* at node *p*, is entirely irrelevant to $a_{n-1}a_{n-2}\cdots a_a$, regardless of whether c > α or not. So the last α bits in nodes' address is of more importance. We define a *Gaussian Graph* based on these α bits.

(*Definition* 6.2): *Gaussian Graph*

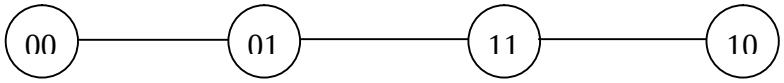
We call the undirected graph G_n ($n \geq 2$) *Gaussian Graph* if it is composed of <V_n, E_n>, where: $V_n = \{ a_{n-1}a_{n-2}\cdots a_1a_0 \mid a_i \in \{0,1\}, \textit{ for } i \in [0,\ n-1] \}$

$$E_n = \{(a_{n-1}a_{n-2}\cdots a_c\cdots a_1a_0,\ a_{n-1}a_{n-2}\cdots \overline{a_c}\cdots a_1a_0) \mid$$

c=0 or $c \in$ [1, n-1] and $\overline{a_{c-1}a_{c-2}\cdots a_1a_0} = c\}$.

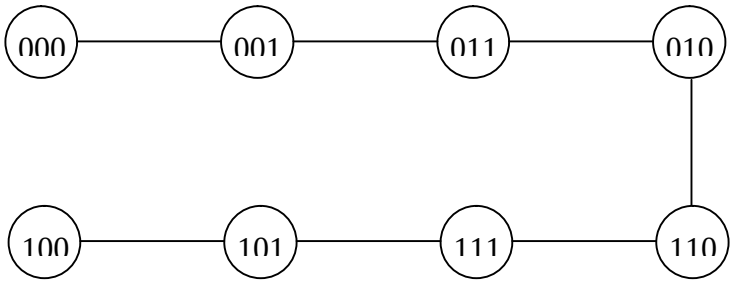
The Figure 6.1 below demonstrates the topology of G_2, G_3, and G_4. They can be generated easily by adding edges, according to the definition of E_n, to the original graph which is composed only of nodes.

G_2 :



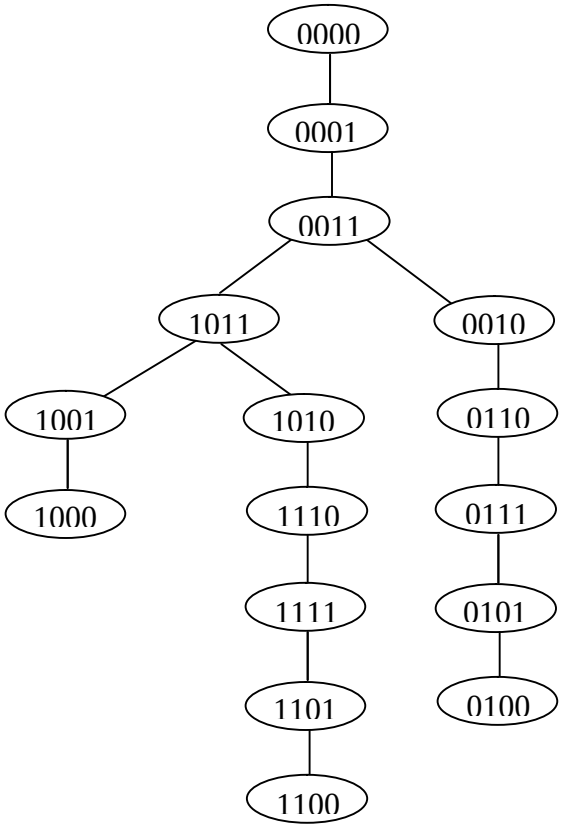
(a)

G_3 :



(b)

G_4 :



(c)

Figure 6.1 (a) G_2, (b) G_3, and (c) G_4

Lemma 1: (Equivalent definition of Tree)

Suppose graph *G* has *n* vertices ($v_1, v_2, \cdots, v_n$) and *e* edges. *G* is a tree if and only if *G* is connected and $e = n - 1$.

Proof: A tree is defined as a connected graph which contains no cycle.

(Sufficient) We prove the proposition by induction on *n*. Clearly, this proposition holds for $n = 1,\ 2$. Assume that it is true for all $n \le k$ ($k \ge 2$). When $n = k + 1$, since *G* is connected, so there is no isolated vertex (vertices whose degree is 0). If there is no

end-vertex (vertices whose degree is 1) in *G*, then $e = \frac{1}{2}\sum_{i=1}^{k+1}\deg(v_i) \geq \frac{1}{2}\sum_{i=1}^{k+1}2 = k+1$,

which contradicts with the fact that $e = n-1 = k$. So there must be an end-vertex *v*. Remove *v* and its only edge from *G*, we get a subgraph *G'*, which has *k* vertices and $k-1$ edges. Since *v* is an end-vertex, *G'* is still connected. Based on the induction assumption, *G'* is a tree. Obviously, constructed by adding an end-vertex to *G'* together with its only edge, the graph *G* is still a tree.

(Necessary)

This is an apparent property of tree. So the proof is omitted here. g

(*Theorem* 6.2) G_n is a tree.

Proof. We prove *Theorem* 6.2 in three steps.

1. G_n is connected.

We prove this proposition by induction on *n*. Clearly, this proposition holds for $n \leq 4$ based on the figures above. Assume that it is true for all $n \leq k$. Suppose when $n = k+1$, G_{k+1} is not connected. Then there must be two vertices $u = u_k u_{k-1} \cdots u_0$ and $v = v_k v_{k-1} \cdots v_0$ ($u_i, v_i \in \{0, 1\}$ for $i \in [0,k]$) between which there is no path. Let *c* be the dimension of the leftmost 1 in $u \oplus v$ ($c \in [0,k]$) and $c = \overline{a_{c-1}a_{c-2}\cdots a_0}$ ($a_i \in \{0, 1\}$ for $i \in [0,c-1]$). Clearly, edge $l = (u', v') = (u_k u_{k-1} \cdots u_c a_{c-1}a_{c-2}\cdots a_0, u_k u_{k-1} \cdots \overline{u_c} a_{c-1}a_{c-2}\cdots a_0) \in G_{k+1}$. We define a subgraph of G_n as $G' = <V', E'>$, where

$$V' = \{u_k u_{k-1} \cdots u_c x_{c-1}x_{c-2}\cdots x_0 \mid x_i \in \{0, 1\} \text{ for } i \in [0,c-1]\}$$

$$E' = \{(v_1, v_2) \mid v_1, v_2 \in V' \text{ and } (v_1, v_2) \text{ is an edge in } G_n\}.$$

Since the connectivity in dimensions less than c is not influenced by the bit value of dimensions no less than c, G' is isomorphic to G_c. As $c \le k$, based on the induction assumption, there is a path in G_{k+1} which connects u and u'. Likewise, there is also a path in G_{k+1} which connects v and v'. As (u', v') is an edge in G_{k+1}, by concatenation, a path is found in G_{k+1} that connects u and v, which contradicts the assumption that there is no path between them. Therefore, G_{k+1} is a connected graph.

2. There are 2^n nodes in G_n. (Obvious)

3. There are exactly $2^n - 1$ edges in G_n.

We denote the number of links spanning in dimension i as $\mathrm{E}_n(i)$ ($i \in [0,\ n-1]$).

According to *Theorem* 6.1, each node has a link in dimension 0, so $\mathrm{E}_n(0) = 2^{n-1}$.

A node has a link on dimension 1 if and only if its rightmost bit is 1. Such links only connect nodes in the form of $(a_{n-1}a_{n-2}\cdots x\ 1, a_{n-1}a_{n-2}\cdots \bar{x}\ 1)$. So $\mathrm{E}_n(1) = 2^{n-2}$.

A link spanning in dimension 2 can only connect node pairs in the form of:

$$(a_{n-1}a_{n-2}\cdots x10,\ a_{n-1}a_{n-2}\cdots \bar{x}10)$$

So $\mathrm{E}_n(2) = 2^{n-3}$.

Likewise, it is easy to prove that $\mathrm{E}_n(\mathrm{i}) = 2^{n-i-1}$.

Thus $|\mathrm{E}_n| = \sum_{i=0}^{n-1} E_n(i) = \sum_{i=0}^{n-1} 2^{n-i-1} = 2^n - 1.$

Combining 1-3 and applying *Lemma* 1, we can conclude that G_n is a tree. g

From now on, we denote G_n as T_n to emphasize this property. We denote the node *k* in T_n as $T_n(k)$. The existence of such a tree is crucial for our algorithm because, for each source and destination pair in a tree, there is a set of nodes, which the packet must come across in its journey, and which can be calculated at the source. This makes routing much more definite and predictable.

6.4 Routing Strategy for Fault-free Gaussian Cube

6.4.1 Introduction

We first develop an algorithm which ensures optimal routing in a fault-free *Gaussian Cube* $GC(n,2^a)$. The algorithm has the following properties:

1) It generates the shortest path for any (source, destination) pair.
2) The computation complexity is $O(a(n-a)\log a)$ at only several nodes on the path, the exact number of which is bounded by $\left\lceil \frac{n-1}{2^a} \right\rceil$.
3) The message overhead is $O(na)$. We have good methods to compress the overhead.

6.4.2 Routing in *Gaussian Tree*

(*Definition* 6.3) *k-Ending Class*

In Gaussian Cube $GC(n,2^a)$, for $\forall k \in [0,2^a - 1]$, we call the following set $EC(n,a,k)$ *k*-ending class:

$$EC(n,a,k) = \{a_{n-1}a_{n-2}\cdots a_a a_{a-1}\cdots a_1 a_0 \mid \ a_i \in \{0,1\},\ i \in [0,n),\ a_{a-1}\cdots a_0 = k\ \}$$

For simplicity, we abbreviate it as $EC(k)$ when the *Gaussian Cube* is given. One obvious conclusion, according to *Theorem* 6.1, is: if a link (v_1, v_2) spans in dimension $c \geq a$, then $v_1, v_2 \in EC(c\%2^a)$. $EC(k)$ corresponds to $T_a(k)$ in *Gaussian Tree* T_a. Note these concepts are all independent of *n*. Let the dimensions no less than α in which each node of $EC(k)$ has a link comprise set *Dim*(*n*, *α*, *k*), then *Dim*(*n*, *α*, *k*) = $[a, n-1] \bigcap [k]_{2^a}$.

To begin with, we briefly introduce the basic ideas underneath this algorithm. Suppose the source is *s* and the destination is *d*. Denote *R* = *s* Å *d*. If there is a 1 in *R* and its dimension *c* is no less than α, then the path from *s* to *d* must cover at least one node *x*, such that $x \in EC(c\%2^a)$. Viewed in T_a, that means the path must begin from $T_a(s\%2^a)$, end at $T_a(d\%2^a)$ and must pass all nodes in $S = \{T_a(k\%2^a) \mid k \geq a,\ R \& 2^k \neq 0\}$. Since the problem has now been mapped to a tree, with the starting and ending nodes as well as the intermediate nodes given, it is simpler to find an optimal route.

Prior to the discussion of our routing algorithm in fault-free *Gaussian Cube*, two fundamental algorithms are introduced. To begin with, the following one aims at finding a route from $T_a(s)$ to $T_a(d)$ in T_a, when a, *s*, and *d* are given.

(*Algorithm* 6.1) *Path Construction Algorithm (PC)*

Let $s = s_{a-1}s_{a-2}\cdots s_1 s_0$ and $d = d_{a-1}d_{a-2}\cdots d_1 d_0$. We first find the leftmost '1' in $R = s$ Å *d.* Suppose it corresponds to dimension *c*. If $c = 0$, then *s* and *d* are neighbors and (*s, d*) can be appended to the path. If $c \neq 0$, as it must reach a node in T_a whose last *c* bits $a_{c-1}a_{c-2}\cdots a_0 = c$, we first go to the node $s_{a-1}\cdots s_c a_{c-1}\cdots a_0$. We can add link *l* = ($s_{a-1}\cdots s_c a_{c-1}\cdots a_0$, $s_{a-1}\cdots \overline{s_c} a_{c-1}\cdots a_0$) to the final path. Then the algorithm runs recursively on *PC* ($s_{a-1}s_{a-2}\cdots s_1 s_0$, $s_{a-1}\cdots s_c a_{c-1}\cdots a_0$) and *PC* ($s_{a-1}\cdots \overline{s_c} a_{c-1}\cdots a_0$, $d_{a-1}d_{a-2}\cdots d_1 d_0$). The recursion must be able to terminate because the leftmost '1' moves at least one bit rightward after one recursion, until it reaches dimension 0 when the source and destination will be neighbors. Finally, *l* concatenates the two paths found. Since it is obvious that the path will not go to one node more than once and we are routing in a tree, the resultant route must be optimal. Besides, such a recursion will go no deeper than α. The implementation of this algorithm has been put in Appendix II. g

In our real implementation, we do not use recursive function. Instead, We use an array and a pointer to simulate a stack. Given the nature of double side recursive function, we cannot generate a route sequentially from source to destination. Therefore additional attention should be paid to the labeling of each link, so that we can find the correct order of links on that path by a simple sort on the labels with time complexity $O(a \log a)$.

As the algorithm finds the path link by link, the complexity (both spatial and computational) is dependent on the diameter of T_a (maximum distance between node pairs). A program is written to calculate diameter of the tree, denoted as $D(T_a)$. The principle idea of the program is that $d(u,v)$, the distance between node *u* and *v* in T_a, equals $d(u,p)+d(p,v)$,

where p is the deepest common ancestor of u and v. Please refer to Appendix III for the source code. The result shows that $D(T_a)$ is $O(a)$. See Figure 6.2 below.

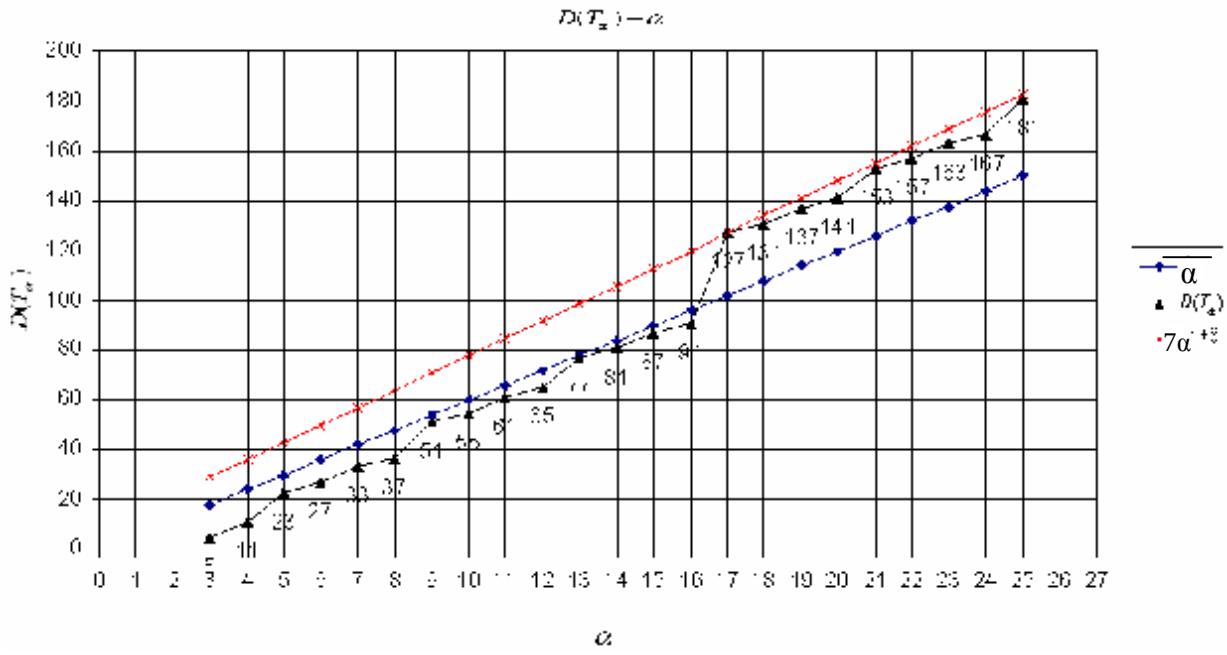


Figure 6.2 Diameter of T_a versus a

Granted, in real practice, we will almost never use α larger than 10 for reasonable node availability. Here we calculated α up to 25 only with an eye to showing that $D(T_a)$ is $O(a)$. So the time complexity for running the Path Construction Algorithm is $O(D(T_a) + D(T_a)\log D(T_a)) = O(a \log a)$.

Secondly, we introduce an algorithm for arranging multi-destination routing from a tree root. Several nodes belonging to the tree need to be visited and then the packet must go back to the root. It is easy to find that as long as the following principle is met, the path generated must be optimal: if the packet is currently at node *p*, it can never backtrack to the parent unless no destination still exists in the subtree of *p*.

(*Algorithm* 6.2) *Closed-Traverse Algorithm in tree (CT)*

Suppose we are at the root $r = r_{a-1}r_{a-2}\cdots r_1 r_0$ where $r_i \in \{0,1\}$ for all $i \in [0, a-1]$. We are to visit $D = \{d_1, d_2, \cdots, d_n\}$ whose members are all nodes in the tree and finally go back to *r*. The prototype of the algorithm is $CT(r, D)$. We first pick up randomly one $d \in D$ and use *Algorithm* 6.1 to find a route *L* from *r* to *d*. Then for each $d_i \in D$, if d_i is covered by *L*, we only need to record that fact. But if it is not covered, we will use the technique in *Algorithm* 6.1 (*PC*) to find a node in *L* at which the packet must branch away from *L*. For example, in the tree shown in Fig. 6.3, the bold line represents *L*, and to reach d_i, the route must branch at b_i.

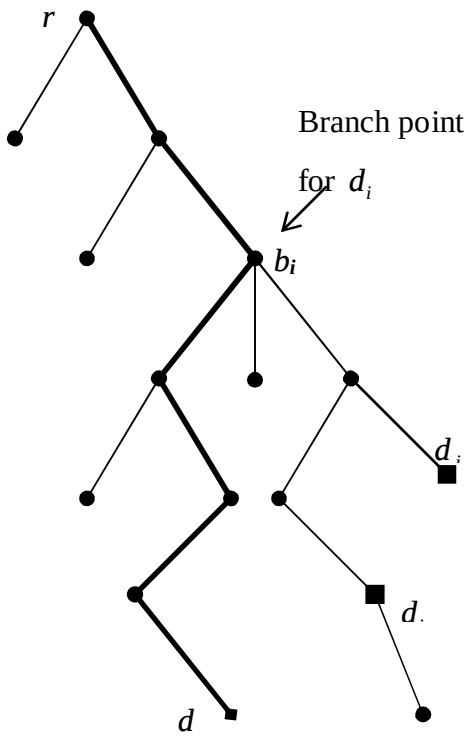


Figure 6.3 Example for *CT* algorithm

However, to calculate b_i, we do not need to find the complete path from *r* to d_i. Similar to Algorithm 1, we first find the leftmost '1' in $R = r$ Å d_i. Suppose it corresponds to dimension *c*. If *c* = 0, then *r* and d_i are neighbors and $b_i = r$. If $c \neq 0$, as it must reach a node in T_a whose rightmost *c* bits $\overline{a_{c-1}a_{c-1}\cdots a_0} = c$, we now check link (v_1, v_2) = $(r_{a-1}\cdots r_c a_{c-1}\cdots a_0,\ r_{a-1}\cdots \overline{r_c} a_{c-1}\cdots a_0)$. If v_1 belongs to *L* while v_2 does not, then $b_i = v_1$. If both v_1 and v_2 belong to *L*, the algorithm only needs to search the branch point between v_2 and d_i. If neither v_1 nor v_2 belongs to *L*, then the branch point must lie between *r* and v_1. So the process can proceed in a recursive way and terminates within α

steps after finding the branch point. Since a node in L might serve as branch point for more than one destination in D, we use a table to record it. We denote the mapping as $B(\cdot)$. For example, in the Figure 6.3, b_i is the branch point for d_i and d_j, so $B(b_i) = \{d_i, d_j\}$.

After all members in D are processed and table $B(\cdot)$ is obtained, we can begin to go from r to d by following L. Once we arrive at a node p where $B(p) \neq \Phi$, we only need to run this algorithm again by calling $CT(p, B(p))$. After this call returns, we proceed along L, until d is reached. Then we only need to go back to r in a reverse direction of L. Since this is a distributed algorithm, CT is not recursive as it appears here. g

It can be easily confirmed that the rule stated above is obeyed in CT, so the route is optimal. The conclusion about the complexity of this algorithm is: suppose we are routing in T_a and $|D| = m<n$ for the original D, the space cost is at most $O(n^2)$ to run CT at each necessary node and time complexity is $O(na)$. The overhead of packet is $O(na)$.

The message overhead ($O(na)$) is a little bit large. We have effective ways to compress it by increasing computation. The major overhead cost lies in recording each branch points p and $B(p)$. But if we calculate $B(p)$ again at each node p at the computation cost of $O(a(n-a))$, the size of overhead can be reduced to $O(n)$. Therefore, the resultant overall gain depends on which part is bottleneck, processor's speed or transmission rate.

We have also noticed that the degree of each node in T_a is tightly bounded. This provides a compact way to record *L* in *CT*, thus reducing the overhead size. It is unnecessary to record the sequence of node address. Instead, we only need to know through which port to go ahead. Moreover, if the degree of current node is 1, then it must backtrack. If the degree is 2, then it must go with the dimension not used in entering the node. So for both cases we don't need to record where to go next. It is calculated that the degree of about 81% nodes in T_a is less than 3. See Figure 6.4.

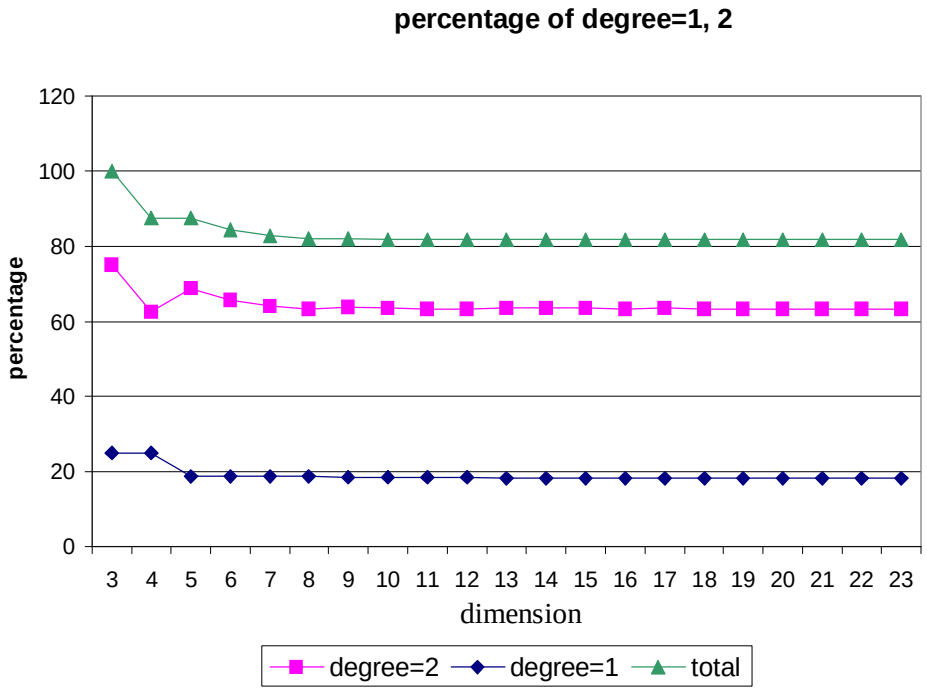


Figure 6.4 Percentage of nodes with degree 1, 2

Even if a node's degree is larger than 2, we still need only to record which link to use. For $\alpha < 11$, the maximum degree is 3, and for α in a practical range, 2 bits is enough for indicating which link to use.

6.4.3 Routing in Fault-free Gaussian Cube

Finally, we present the complete routing algorithm for fault-free *Gaussian Cube,* by combining *Algorithm* 6.1 (PC) and *Algorithm* 6.2 (CT).

(*Algorithm* 6.3) Fault Free Gaussian Cube Routing (FFGCR)

The input of FFGCR is: n and α for $GC(n,2^a)$, source $s = s_{n-1}s_{n-2}\cdots s_1s_0$ and destination $d = d_{n-1}d_{n-2}\cdots d_1d_0$.

Firstly, we map the problem from $GC(n,2^a)$ to T_a. Let $p = s \oplus d$. We denote:

$P = \{ i \in [a, n-1] \mid p \,\&\, 2^i \neq 0 \}$ $\qquad$ $D = \{ T_a(x\%2^a) \mid x \in P \}$

By viewing in T_a, we are routing from $s' = T_a(s \% 2^a)$ to $d' = T_a(d \% 2^a)$ and we must cover all nodes in the set *D*.

At *s'*, we use *Algorithm* 6.1 to find a path $L \subset T_a$ to *d'*. Then we use the technique in *Algorithm* 6.2 to find the branch point for all members in *D*. Go along *L*. This means traversing by using the least significant α dimensions in the original $GC(n, 2^a)$. Once we reach a branch point *b*, use *Algorithm* 6.2 to traverse all nodes whose branch point to *L* is *b*. Whenever the packet reaches a node *x* whose corresponding node in T_a is a member of *D,* it will go through all preferred dimensions $c \in [a, n-1] \cap [x]_{2^a}$. g

Obviously, FFGCR finds the shortest route from *s* to *d*. Denote $H = D(T_a)$. The time complexity is $O(H + H\log H + (n-a)a\log H)$, where each item stands for:

H:	Picking up links that comprise the path L from *s'* to *d'* (not necessarily in the order of from source to destination)
$H \log H$	Sorting the links to reorganize the path from *s'* to *d'*
$(n-a)a \log H$:	Time for finding branch point. There are at most $n-a$ preferred dimensions in $[a, n-1]$. Search in *Algorithm* 6.2 takes at most α rounds of recursion. Each round involves a look-up in H sorted nodes in L.

Table 6.1 Components of computation complexity

Since $H = O(a)$, the total complexity is $O(a(n-a)\log a)$. Such an amount of computation is carried out at the source and all branch points.

The space required for each node to run the algorithm is $O((n-a)+H) = O(n)$.

The message overhead is: $O(na)$. It can be reduced if the method proposed in section 6.4.2 (after the introduction of *Algorithm* 6.2) is adopted.

Up to now, the routing problem in fault-free *Gaussian Cube* has been completely solved. It will be shown later that *Algorithm* 6.3 serves as a basis for our fault-tolerant routing strategy in *Gaussian Cube* and it contributes to the theoretical completeness of routing in *Gaussian Cube*.

6.5 Fault-tolerant Routing in Gaussian Cube

6.5.1 Introduction

When we go ahead to fault-tolerant routing strategy design, we have to take some practical considerations. The most important one is that node degree in $GC(n,2^a)$ is mostly about $\frac{n-a}{2^a}+2$. The minimum node degree is $\left\lfloor \frac{n}{2^a} \right\rfloor +1$, occurring at nodes whose address is multiple of 2^{α}. So a natural bound is that $GC(n,2^a)$ cannot tolerate over $\left\lfloor \frac{n}{2^a} \right\rfloor$ faulty components. It is clear that once α reaches 3 or more, the network is very likely to be disconnected and suffer from intrinsically poor fault tolerance ability. There are two approaches to tackle this problem. A natural idea is to restrict α to be small. When α = 0, $GC(n,\ 2^a)$ is effectively a binary hypercube. If we restrict α to within [0, 2], the problem will be very uninteresting and T_a will degrade to a linear array. Therefore, some novel notions and metrics must be used in this new setting. In this chapter, a new approach to classify errors is introduced and the influence of errors is analyzed. We first discuss the basic form of the fault-tolerant routing strategy, which disposes of faulty links only. The extended form, which completely solves the fault tolerant routing problem, will be presented in the last section with close relationship to *Exchanged Cube.*

6.5.2 Basic Fault-tolerant Routing Strategy

Firstly, a categorization of faulty components will be useful.

(*Definition* 6.4) A-category (link) fault

If a link error occurs at a dimension $c \geq a$, it is called A-category (link) fault.

(*Definition* 6.5) B-category fault

If all link failure(s) incurred by an error are in dimension(s) less than α, then the error is called B-category fault.

Note: unlike A-category fault which can occur only in the form of link error, B-category faults can be both link error and node error, as long as that node has no link spanning in a dimension $c \geq a$. A link error is either A-category or B-category.

(*Definition* 6.6) C-category (node) fault

If a node error implies break down of links in dimensions both smaller and no smaller than α, it is called C-category (node) fault.

A node error is either B-category or C-category because each node has one link spanning in dimension 0. In short, all faulty components must belong to one and only one of the three categories.

In *Gaussian Cube* $GC(n,2^a)$, for $\forall k \in [0,2^a - 1]$, we have defined *k*-ending class:

$$EC(k) = EC(n,a,k) = \{a_{n-1}a_{n-2}\cdots a_a a_{a-1}\cdots a_1 a_0 \mid \ a_i \in \{0,1\}, \quad a_{a-1}\cdots a_0 = k \ \}$$

The following definition decomposes *k*-Ending class into further refined classes.

(*Definition* 6.7) *k*-Ending-*t*-Equivalent Class

In *k*-ending class $EC(n,a,k)$, for $\forall\ t \in [0,2^{n-a-|Dim(k)|}-1]$, we call the following set

$EEC(n,a,k,t)$ *k*-Ending-*t*-Equivalent Class

$$EEC(n,a,k,t) = \{ a_{n-1}\cdots a_a a_{a-1}\cdots a_0 \in EC(n,a,k) \mid \text{bits other than } [0,a-1] \cup Dim(k) \text{ comprise value } t\}$$

We define *k*-Ending-*t*-Equivalent *Graph* $GEEC(n,a,k,t)$ as $< V(n,a,k,t), E(n,a,k,t) >$, where

$$V(n,a,k,t) = EEC(n,a,k,t)$$

$$E(n,a,k,t) = \{(v_1, v_2) \mid v_1, v_2 \in EEC(n,a,k,t),\ (v_1, v_2) \in E\}$$

(E is the edge set of $GC(n,2^a)$)

The following theorem is obvious, but it gives an insight into the advantage of categorizing faulty components.

(*Theorem* 6.3)

If only A-category faults exist in $GC(n, 2^a)$, and in each $GEEC(n,a,k,t)$ ($k \in [0,2^a-1]$, $t \in [0,2^{n-a-|Dim(k)|}-1]$), the number of faulty component is less than $|Dim(k)| = \left\lfloor \frac{n-1-k}{2^a} \right\rfloor + 1 - d(k,a)$ ($d(k,a) = k < a\ ?1:0$), there is a fault-tolerant and cycle-free routing strategy for any source and destination pair.

Proof.

Obviously, $GEEC(n,a,k,t)$ is a binary hypercube embedded in $GC(n,2^a)$. There are many existing routing algorithms, including *FTFR* I proposed, that ensure a packet to be

sent from any non-faulty source to any non-faulty destination in a deadlock-free fashion, as long as the number of faulty links is less than the dimension of the hypercube and no node fault exists.

In *FFGCR* for $GC(n,2^a)$, let source be *s* and destination be *d*. Let $p = s \oplus d$. We denote:

$P = \{ i \in [a, n-1] \mid p \,\&\, 2^i \neq 0 \}$ $\quad D' = \{ x \ \% \ 2^a \mid x \in P \}$ $\quad I = \{ EC(x) \mid x \in D' \}$

As there are only A-category faults, traversing through links spanning in the least significant α dimensions is always successful. So it is guaranteed that for any member $EC(k)$ in *I*, a packet can reach at least one node in $EC(k)$. Suppose a packet reaches $EC(k) \in I$ by arriving at node x and $x \in EEC(n,a,k,t)$. The k $(if\ k \geq a), k+2^a$, $k+2\cdot 2^a, k+3\cdot 2^a, \cdots, k+\max(0,\left\lfloor \frac{n-k-1}{2^a} \right\rfloor)\cdot 2^a$ bits of x and d are $x_0 x_1 \cdots x_{|Dim(k)|-1}$ and $d_0 d_1 \cdots d_{|Dim(k)|-1}$ respectively. Then we can focus on routing in binary hypercube $GEEC(n,a,k,t)$ from $x_0 x_1 \cdots x_{|Dim(k)|-1}$ to $d_0 d_1 \cdots d_{|Dim(k)|-1}$, which is guaranteed by existing algorithms and the precondition of the theorem. All the bits in $[k]_{2^a}$ are set to be same as *d*, we can use links spanning in the last α dimensions to go to another member in *I*, and finally we get to the destination *d*. g

Suppose |*D'*| = m, and in the *k*-Ending-*t*-Equivalent class which is encountered at the i^{th} time, there are F_i A-category faults. Then the resultant route is at most $2 \cdot \sum_{i=1}^{m} F_i$ longer than the optimal route found in a fault free setting.

Now we can conclude that in $GC(n,2^a)$, if there are only A-category faults, then the maximum number of fault tolerable is:

$$T(GC(n,2^a)) = \sum_{k=0}^{2^a-1} 2^{n-a-t_k} \max(t_k - 1,\ 0) \qquad \text{Eq. (4)}$$

$$\text{where } t_k = \left\lfloor \frac{n-k-1}{2^a} \right\rfloor + 1 - d(k,a)$$

The following figures demonstrate the trend of $T(GC(n,2^a))$ with respect to n, when α = 1,

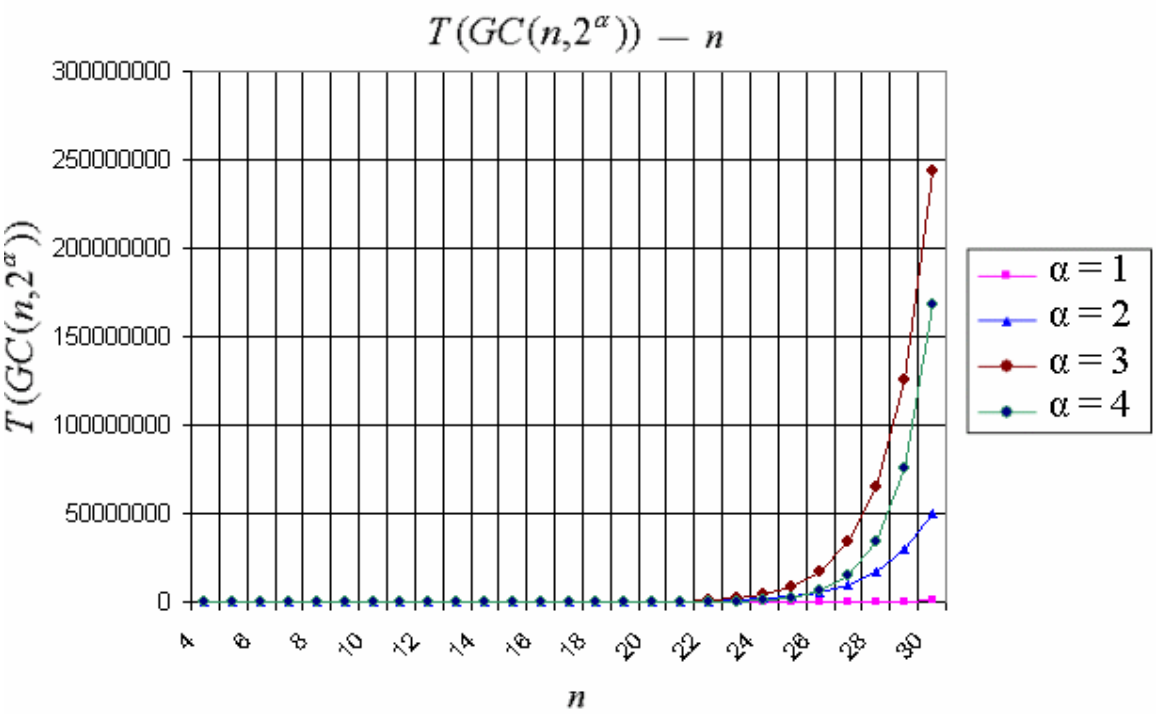


Figure 6.5 $T(GC(n,2^a)) \sim n$

To make the figure clearer, we use $\log_2(T(GC(n,2^a))$ for comparison.

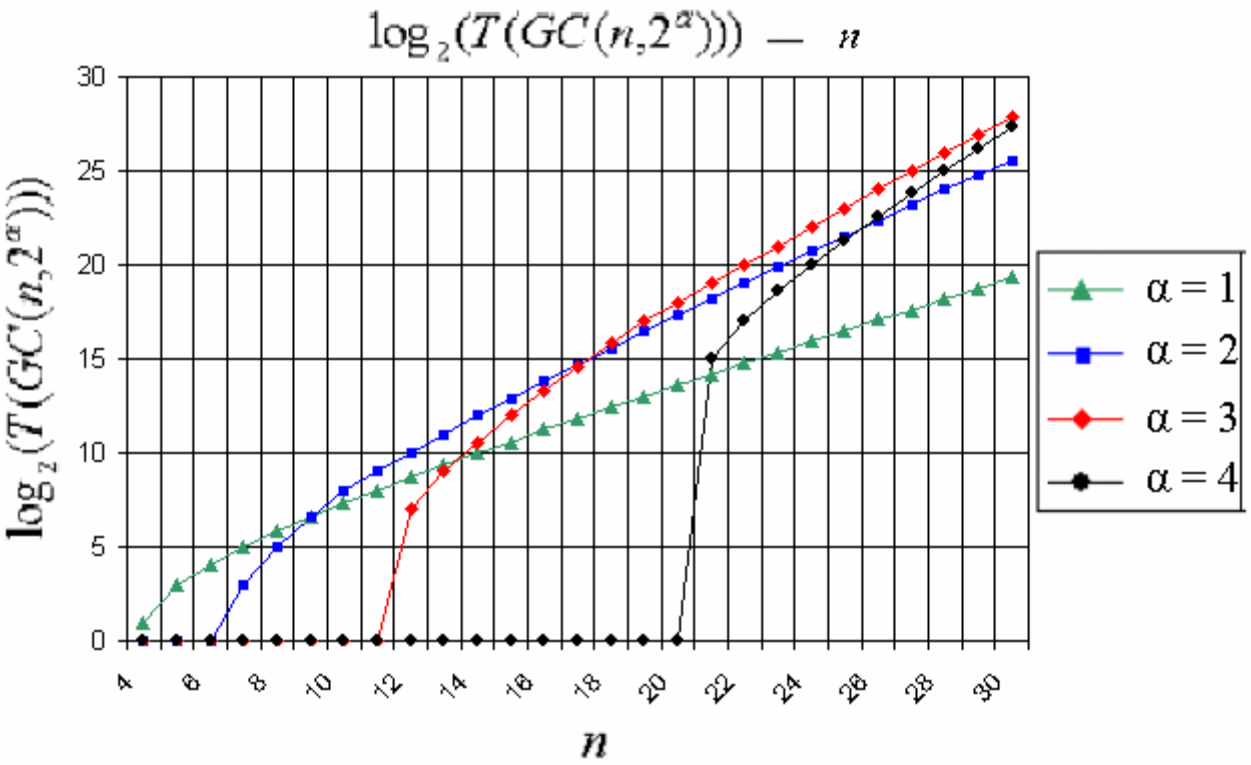


Figure 6.6 $\log_2(T(GC(n,2^a)))$ ~ n

An interesting observation is that when α increases, $T(GC(n,2^a))$ decreases for small n and increases for large n. We can see that when α = 3, the jump point of the line corresponding to $a = 3$ is after $n = 8 = 2^{\alpha}$, and when α = 4, the jump point is delayed by 4 from $n = 16 = 2^{\alpha}$. This is because only after the dimension of a network becomes large enough, can it tolerate faults. In Equation (4), $T(GC(n,2^a)) = \sum_{k=0}^{2^a-1} 2^{n-a-t_k} \max(t_k - 1,\ 0)$ and $t_k = \left\lfloor \frac{n-k-1}{2^a} \right\rfloor + 1 - d(k,a)$, only when $n \geq 2^a$ can some $t_k \neq 0$ and thus $T(GC(n,2^a)) \neq 0$. The delay is caused by $d(k,a)$, because the dimension of embedded k-Ending-t-Equivalent *Graph* must go larger than 1. As for large n, when α increases, t_k decreases, so that 2^{n-a-t_k} grows exponentially which makes $T(GC(n,2^a))$ larger. In other

words, it is the exponentially increasing number of embedded *k*-Ending-*t*-Equivalent *Graph*s that makes $T(GC(n,2^a))$ also grows exponentially.

Another interesting property of this algorithm lies in the influence of each A-category fault. If there exists an $GEEC(n,a,k,t)$ in which the number of A-category fault is over $\left\lfloor \frac{n-1-k}{2^a} \right\rfloor - d(k,a)$, routing will still be guaranteed to be successful if source and destination do not differ in any dimension $c \in [a, n-1] \bigcap [k]_{2^a}$.

Since A-category fault excludes the possibility of node faults, we need an algorithm to deal with B-category and C-category faults as well. The following section 6.5.3 deals with this problem. The discussion of that algorithm is closely related to *Exchanged Cube.*

6.5.3 Extended Fault-tolerant Routing Strategy

Suppose in $GC(n,2^a)$, $T_a(p)$ and $T_a(q)$ are neighbors in T_a. For each $k \in [0, 2^{n-a-|Dim(p)|-|Dim(q)|}-1]$, we define graph $G(n,a,p,q,k) = < V(n,a,p,q,k)$, $E(n,a,p,q,k) >$, where $V(n,a,p,q,k)$ is the set of nodes in $GC(n,2^a)$ whose bits in dimensions other than $Dim(p) \bigcup Dim(q) \bigcup [0, a-1]$ comprise value *k* and whose rightmost α bits represent *p* or *q*. $E(n,a,p,q,k)$ is the subset of links in $GC(n,2^a)$ which connect nodes in $V(n,a,p,q,k)$. If the last α bits are viewed as one dimension that can take value only in {*p*, *q*}, then $G(n,a,p,q,k)$ is effectively isomorphic to *Exchanged Cube*

$EH(|Dim(p)|,|Dim(q)|)$. (Note: we do not use $EH(|Dim(q)|,|Dim(p)|)$ though both can do.) Denote the number of faulty component in $B_t(G(n,a,p,q,k))$ as $e_t(n,a,p,q,k)$, and that in $B_s(G(n,a,p,q,k))$ as $e_s(n,a,p,q,k)$. The number of link faults in $E_0(G(n,a,p,q,k))$ is denoted as $e_0(n,a,p,q,k)$.

(*Theorem* 6.4)

In $GC(n,2^a)$, for all $T_a(p)$ and $T_a(q)$ which are neighbors in T_a, as long as $e_s(n,a,p,q,k)+e_0(n,a,p,q,k)<|Dim(p)|$ and $e_t(n,a,p,q,k)+e_0(n,a,p,q,k)<|Dim(q)|$ for all $k\in[0,2^{n-a-|Dim(p)|-|Dim(q)|}-1]$, there is a fault-tolerant and cycle-free routing strategy for any nonfaulty source and destination pair.

Proof. (Outline)

The algorithm used in *Theorem* 6.3 fails only when a link in dimension $[0,a-1]$ is broken. With our discussion about the fault tolerant routing in *Exchanged Cube,* such a problem is solved once the fault number is restricted by the precondition of *Theorem* 6.4. g

Unfortunately, different from *Theorem* 6.3, if there exists a $G(n,a,p,q,k)$ in which the number of faulty component violates the restriction in the precondition of *Theorem* 6.4, routing might fail even if source and destination do not differ in any dimension $c\in Dim(p)\cup Dim(q)$. That is because the B-category and C-category faults influence the routing in *Gaussian Tree* T_a, where many dimensions other than the preferred dimensions will be used more than once.

Up to now, we have completely solved the problem of fault tolerant routing in *Gaussian Cube*. We used a new method to categorize faulty components so our approach is more meaningful than dealing with the trivial bound of network node availability. For hypercubes constructed by link dilution, this approach to analyzing routing algorithms' ability of tolerating faults is novel and useful because it is expected that this kind of topology will lose in traditional metric: node availability. The tree structure is very helpful to make the problem more deterministic and controllable.

Chapter 7: Simulator

In this part, a software simulator is constructed to imitate the behavior of the real network, and thus test the performance of FTFR. The current simulator model is mainly based on the work of Wong Chuen Vong [20]. In this project, we point out some rectifications and improvements to the model, both technical and theoretical. Special attention was paid as to how to efficiently simulate an incomplete network.

7.1 Overview of the Simulator

In this simulator, packets can traverse the network and reach the destination, with routing decisions made at each intermediate node. There are three important components in the simulator: **j** topology of the network; **k** implementation of the routing strategy; **l** timing methods to measure the useful metrics and statistical analysis of the result.

There are nine basic assumptions in this simulator:

- Fixed packet-sized messages are used.
- Source and destination nodes must be nonfaulty.
- Destination node must not be source node.
- Packet reaching destination is absorbed
- Eager readership is employed where packet service rate is faster than packet arrival rate.
- A node is faulty when all of its incident links are faulty.
- A node knows status of its links to its neighboring nodes and faulty nodes in the network
- No packet is generated for faulty nodes.
- All faults are fail-stop.

The simulation model is composed of several functional modules, with their relationship shown Fig. 7.1:

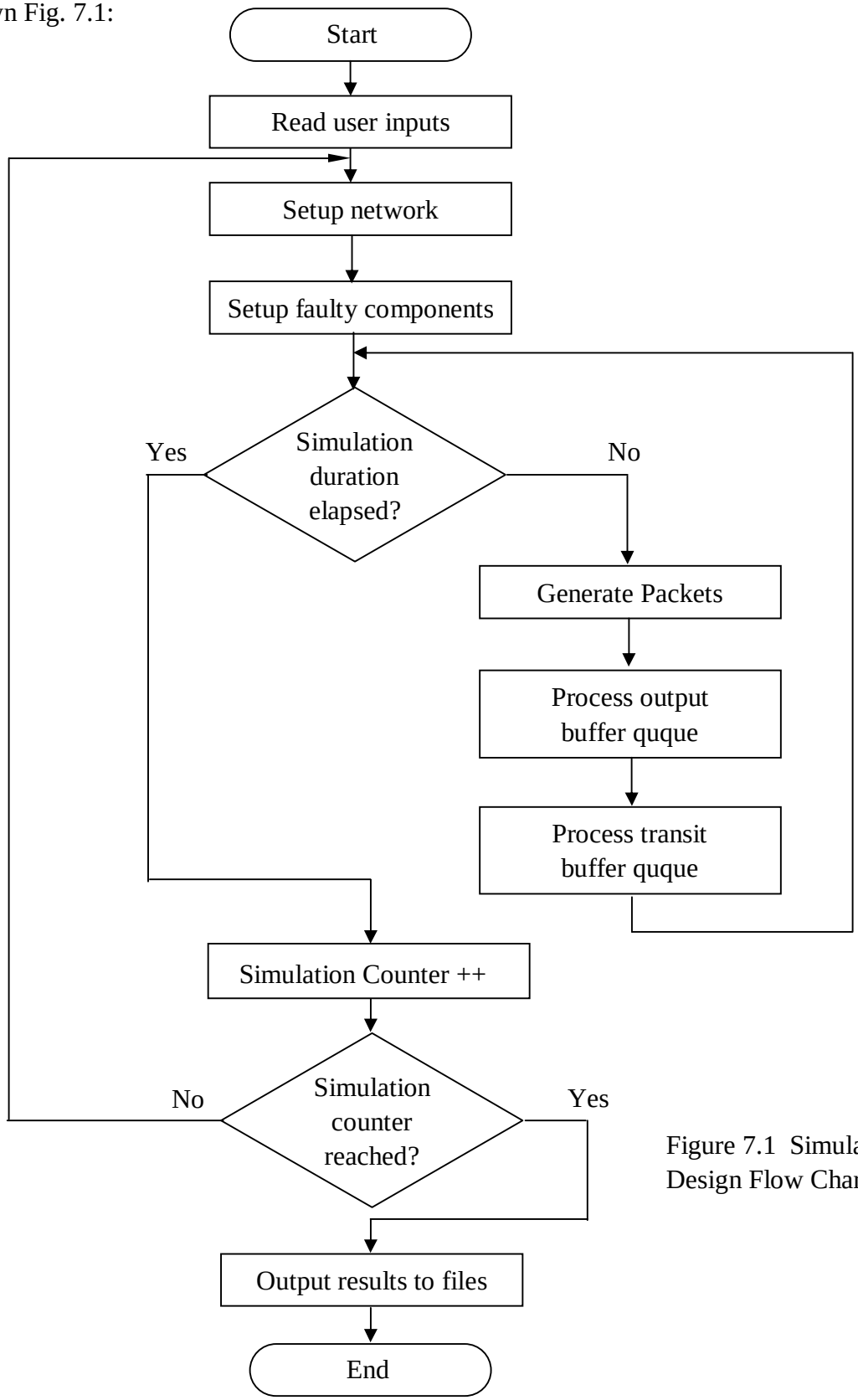


Figure 7.1 Simulation Design Flow Chart

7.2 Analysis of simulator components

This part describes in detail the components in Fig. 7.1. Some rectifications and improvements are mentioned in this section that are made to the previous design. Two of them are of great significance to the final result. There are also some original proposals for implementing incomplete networks. For simplicity, we take the regular Fibonacci Cube of order $n+2$, for instance ($n \geq 1$).

7.2.1 Setup Network

In addition to initializing network parameters such as node availability and total number of nodes and links, the major task in this stage is initializing the node array, which is the physical representation of the whole network. The number of nodes can be calculated by the sequence presented in [12][13][14][15]. The number of links can also be easily obtained by induction introduced in [12][13][14][15]. The data structure of a node is as follows:

```
class  CNode
{
public:
    unsigned avaiVector;          // availability vector
    CQueue *NodeQueue;            // point to first packet in node queue (injection queue)
    CQueue *TransitQueue;         // point to first packet in transit queue (input queue)
    CQueue *OutputQueue;          // point to packet in Output Queue
    CPacket *CentralBuffer;       // point to packet in Central Buffer
}
```

Various numbers of buffers are allocated to each queue at each node. There is only one injection queue assigned to each node and with unlimited size (which is acceptable for simulation). Depending on the topologies employed and the dimensions of the network, each node will have node degree number of transit queues and output queues, 10 packet buffers per transit queue and 1 packet buffer per output queue. (Refer to Figure 7.2)

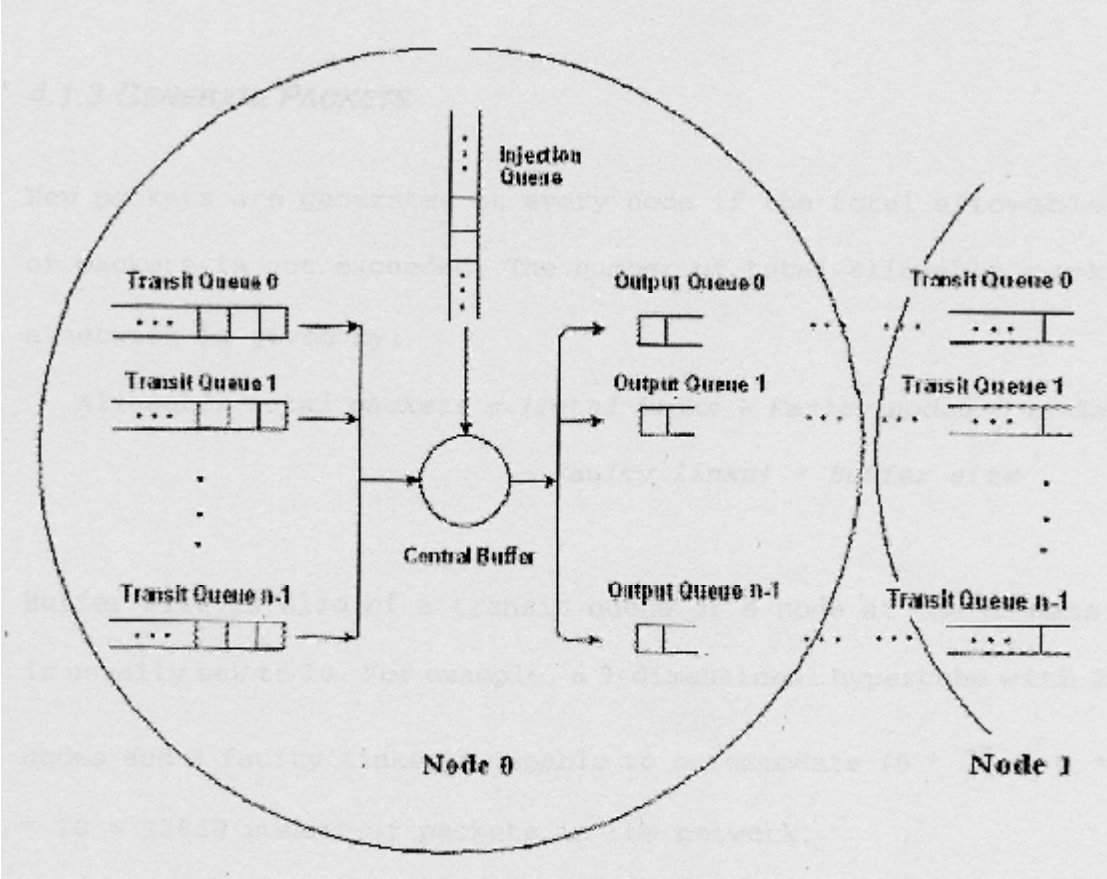


Figure 7.2 Node model

In our simulation model, there is no data structure for links or edges. Instead, each output queue and transit queue at the neighbor correspond to one link connection in between. A network can obtain message from either its injection queue or transit queue. New packets are injected into the injection queue and packets received from neighboring nodes are queued in the transit buffer. To make routing decision, packets must be transferred to central buffer one by one. Then it is routed to the next node's transit queue via a certain

dimension if destination is not reached, or the output queue if next node's transit queue for that dimension is full, or injection queue if even the local output queue is full.

An intricate problem for incomplete cube is that not all dimensions are available at each node, even in a fault-free setting. In FC_n, the node degree varies from $\left\lfloor \frac{n-2}{3} \right\rfloor$ to $n-2$ [12]. In EFC_n, the node degree varies from $\left\lceil \frac{n}{4} \right\rceil$ to $n-2$ [14]. In $XFC_k(n)$, the node degree varies from $\left\lceil \frac{n-(k-1)}{3} \right\rceil + k - 1$ to $n-2$ [15]. Thus, we have to calculate the availability vector beforehand. Unlike the former model, we don't construct the queues until the faulty components are selected. The benefit is we need not allocate memory space for these faulty links, though we still allocate memory for faulty nodes. As a result, the availability vector at a node contains all the information about the available dimensions.

7.2.2 Setup faulty components

Faulty components consist of either faulty nodes or faulty links or both. The determination of number of faulty nodes and links is:

$$FC = FN + FL$$

FC is the number of faulty components, *FN* is the number of faulty nodes and *FL* is the number of faulty links. A faulty node will render all its incident links faulty.
We can specify both *FN* and *FL*. We can also specify *FC* only, with *FN* and *FL* determined by random selection as long as *FC* = *FN* + *FL*. The selection of the location of faulty components is same as the previous model, with careful avoidance of duplicate selection and picking non-existent components.

7.2.3 Gather global network status

In FTFR, each node needs to know the availability vector of all its neighbors. However, this process of exchanging information is omitted here for two reasons.
Firstly, this is a simple duplication costing one hop time with no calculation.

Secondly, there is no point in allocating space locally at each node since the information is available in global data structure. Consequently, a large amount of space is saved.

7.2.4 Generate Packets

New packets are generated at every node if the total allowable number of packets is not exceeded. The total allowable packet number is defined as:
(Total Links – Faulty Link – Number of links incident to Faulty Nodes)×Buffer Size
Buffer size is the size of transit queue of a node at each dimension. In our test, it is set to 10.

The generation of packets follows the trend in the selected probability distribution function. Ten choices of distribution functions are provided in the program:

- Uniform distribution
- Bernoulli distribution
- Beta distribution
- Binomial distribution
- Exponential distribution
- Normal distribution
- Log normal distribution
- Poisson distribution
- Weibull distribution
- Erlang distribution

As global information is easily accessible in the simulation tool, it is easy to ensure the assumption that destination is not a faulty node.

7.2.5 Process output buffer queues

Packets waiting in the output buffer queues are sent to their respective neighbors via the corresponding links. The transmission is considered as one hop. If the transit buffer queue of the neighbor is full, the packet will remain in the current node's output buffer queue. All output buffer queues are processed in round robin fashion.
However, unlike binary hypercube, in incomplete cubes, the packet stored in the output queue OutputQueue[i] (which means it will use the i^{th} available dimension at current node), might not be expected to be sent to the neighbor's TransitQueue[i], because that dimension will possibly no longer stand as the i^{th} available dimension there. For example, for node 100 in a 3-dimension regular Fibonacci Cube, dimension 2 is the second available dimension since 110 is not a valid Fibonacci address. But at the corresponding neighbor, 000, dimension 2 will be the third available dimension. So we need a translation table at each node, the i^{th} item of which records such a change in the i^{th} available dimension. The value can be calculated by utilizing the availability vector of the current node and its neighbors. In real implementation, we save that huge space by re-calculating it at each iteration. The result shows that this $O(n)$ computation costs only a very small fraction of total simulation time. The main advantage is the saving of a significantly large amount of memory space, which enables us to test networks of higher dimension.

7.2.6 Process transit buffer queues

If packets are available in transit buffer queues, it is transferred to the central buffer where routing algorithm is applied and determined whether this packet has reached its destination or needs to be routed. If the packet is destined for the current node, it is absorbed (deleted or de-allocated). Otherwise, it is sent to the next node's transit buffer

queue (if there is available buffer space) or transferred to current node's output buffer queue (again, if space is available) or appended to the injection buffer queue. All transit buffer queues are processed in round robin fashion.

7.2.7 Process injection buffer queue

Similar to the processing of transit buffer queues, packets generated that are waiting in the input buffer queue are transferred to the central buffer and routing algorithm is applied there. Then the packet is sent to the next nodes' transit buffer queue (if there is available buffer space) or transferred to current node's output buffer queue (again, if space is available) or appended to the injection buffer queue.

7.3 Special problems and solutions

In this section, we focus on some special problems for simulating incomplete hypercubes. These include an efficient way of storing the incomplete network, and the intrinsic timing problem of using a single processor to simulate the parallel architecture. The precision problem is also recapitulated.

7.3.1 Efficient Storage

Fibonacci-class cubes are incomplete cubes, so if we use the binary value of a node's address as index of the node array, a lot of space will be wasted. Therefore, we need a function that efficiently maps between the order of a node and the node's address.
An interesting property of Fibonacci code is that each integer $N \in [0, f_{n-2} - 1]$ has a unique *order-n Fibonacci code*. This can be attributed to the *greedy* approach used in conversion. First, find the greatest Fibonacci number $f_k \leq N$, and assign a "1" to the bit

that corresponds to f_k. Then, proceed recursively for $N - f_k$. The unassigned bits are 0's. In a Fibonacci code, the least significant bit is f_2 rather than f_1.

The set of Fibonacci number $f_2, f_3, \cdots, f_n, \cdots$ is not linearly independent on {0,1}, that is, any f_i $(i \geq 4)$ can be expressed by the linear combination of other Fibonacci numbers with coefficients taken in {0,1}, given $i \in [0, f_n - 1]$. Thus, there are more than one $(b_{n-1,} \cdots, b_3, b_2)_F$ such that b_j is either 0 or 1 for $2 \leq j \leq (n-1)$ and $i = \sum_{j=2}^{n-1} b_j \cdot f_j$. In Fibonacci Cube, it is the greedy approach that guarantees this inner-product-like mapping to be a bijection between [0, $f_{n-2} - 1$] and the node address in Fibonacci Cube FC_n.

This property makes it possible to use an array in the size of f_n to simulate the Fibonacci Cube of order *n*. In the simulator, function *Fib2Dec*() can convert a $(n-2)$-bit binary address $(b_{n-1,} \cdots, b_3, b_2)_F$ into a decimal number *i* by applying $i = \sum_{j=2}^{n-1} b_j \cdot f_j$. The inverse function is implemented by *Dec2Fib*().

Unfortunately, as the variants of Fibonacci Cube don't employ greedy approach, different nodes might represent the same integer. E.g. in EFC_8, 100000 and 010110 are both valid addresses. But $(1,0,0,0,0,0) \cdot (f_7, f_6, \cdots, f_2)^T = 13 = (0,1,0,1,1,0) \cdot (f_7, f_6, \cdots, f_2)^T$.

Hence, to simulate these cubes of order *n* without the loss of their foremost advantage: low expandability, we have to find a one-one bijection which can efficiently map between a valid node address and [0, *F*-1] where *F* is the total number of nodes in the

network. Otherwise, it is inevitable to use an array of length 2^{n-2}. Before presenting this interesting method, it is better to see an algorithm that maps an *n*-bit Fibonacci code 'original' to an integer $x \in [0, f_{n+2} - 1]$. The result is the same as what the greedy approach produces. To use it, call Fib2Dex (x, $n-2$).

```
// x is a (digit)-bit Fibonacci Code
unsigned Fib2Dec(unsigned x, unsigned digit)
{
        unsigned mask, top;

        if(digit > 1)
        {
                mask = (1 << (digit - 1));
                if(x & mask)            // test whether the highest two bits are '10'
                        return FibNum[digit + 1] + Fib2Dec( x, digit - 2);
                        // FibNum[digit+1] stores Fibonacci number F_digit+1
                else
                        return Fib2Dec( x, digit - 1);
        }
        else
                return x & 1;
}
```

Denote the mapping as $G(\cdot)$. The principle underneath it is:

If $a_i \in \{0,1\}$ for $i \in [0, n-1]$,

$$G(a_{n-1}a_{n-2}\cdots a_1a_0) = \begin{cases} G(a_{n-2}\cdots a_1a_0) & \text{if } a_{n-1} = 0 \\ f_{n+1} + G(a_{n-3}\cdots a_1a_0) & \text{if } a_{n-1}a_{n-2} = 10 \end{cases}$$

It can be easily proved that this algorithm is equivalent to the greedy approach. However, it opens a window to finding a one-one bijection for other Fibonacci-class cubes. The following demonstrates an algorithm that works for Enhanced Fibonacci Cube of order *n*. To use the algorithm, call Fib2Dec (x, $n-2$).

```
unsigned Fib2Dec(unsigned x, unsigned digit)
{
	unsigned mask, top;

	if(digit > 4)
	{
		top = x >> (digit - 2);			// test the leftmost 2 bits
		if(top == 0)				// if they are 00
		{
			mask = (1 << (digit - 2)) - 1;
			return Fib2Dec(mask & x, digit - 2);
			// extract the last digit – 2 bits for recursion

		}
		else if(top == 2)			// if they are 10
		{
			mask = (1 << (digit - 2)) - 1;
			return 2 * FibNum[digit-2] + FibNum[digit] +Fib2Dec(mask & x,
				digit - 2);	// extract the last digit – 2 bits for recursion
		}
		top = x >> (digit - 4);			// test the leftmost 4 bits
		if(top == 5)				// if they are 0101
		{
			mask = (1 << (digit - 4)) - 1;
			return FibNum[digit-2] + FibNum[digit] +Fib2Dec(mask & x,
									digit - 4);
		}
		else					// if they are 0100
		{
			mask = (1 << (digit - 4)) - 1;
			return FibNum[digit] + Fib2Dec(mask & x, digit - 4);
		}
	}

	// the following disposes of the initial conditions
	else if (digit == 4)
	{
		if(x<3)
			return x;
		else if(x>7)
			return x-3;
```

```
                else
                        return x-1;
        }
        else if(digit==3)
        {
                if(x<3)
                        return x;
                else
                        return x-1;
        }
        else
                return x;
}
```

A weak point of the algorithm above is that it is recursive, which is not suitable for hardware implementation. Thanks to the left-induction nature of Fibonacci Cube's definition, we can simply convert it into a non-recursive function. Please refer to Appendix IV for these algorithms.

It is easy to extend this method to Extended Fibonacci Cubes. It is also straightforward to design an algorithm that maps an integer back to a Fibonacci code. For details, please refer to Appendix IV.

Now, we have found an efficient bijection which will help us save a lot of memory space in simulation. As is shown later, we can safely simulate Fibonacci-Class cubes to dimensions over 20. This is worthwhile because the scale of *n*-dimensional Fibonacci-Class Cube is about the same as a binary hypercube with dimension n/1.46. Here,

$$1.46 \approx \frac{2}{\frac{(1+\sqrt{3})}{2}}.$$

The discussion above also gives a valuable hint that if we want to study Fibonacci-Class Cubes in a unified way, it is better to focus on node labels' bit pattern, instead of their corresponding decimal numbers.

7.3.2 Timing strategy

In actual communication network, routing is performed in a distributed fashion by all processors in parallel. As only one processor is available for simulation, special approaches must be adopted for conversion. Actually, two metrics are related to timing: packet latency and throughput time. The latter will be discussed in 7.3.3. As for the former, the elapsed time for a node to service a packet is recorded. For the serviced packet and other packets in the current node's queues except the injection buffer queue, the recorded elapsed time is added to their accumulated time. This recorded elapsed time is not added to the accumulated time for other packet in other nodes' queues. The time to generate a packet will not be included in the elapsed time of that packet. Hence, the total accumulated time for each packet is dependent on the time it is being serviced and the time it is waiting in queue of a node while that node is servicing another packet. By using accumulation of elapsed time for packets, it seems like all packets are processed in nodes concurrently.

To control the total simulation time, a timer is used to record the time passed since the beginning of simulation. Each node is processed in a round robin fashion and it processes output queue, transit queue and injection queue successively. At the end of the node's process, the timer is checked to see whether the total elapsed time has exceeded the specified simulation duration.

7.3.3 Timing precision issue

The library function provided by system can measure time by milliseconds. However, if we use that 'large' unit, the result will be all zero. To achieve the accuracy at microsecond level, a set of assembly directives were written, which can make timing accuracy up to the level of processor clock cycle number. The contribution of the project is to encapsulate the original approach into a separate class, providing P( ) and V( ) methods for measuring time. Its usage is like a stop watch, with P( ) starting it and V( ) stopping it. For example, after executing the sequence: Reset, P_1, V_1, P_2, V_2, . . . , P_n, V_n, the value returned by calling getDuration( ) is $\sum_{i=1}^{n} d(V_i, P_i)$, where $d(V_i, P_i)$ represents the time passed between V_i and P_i. Besides, the implementation is more efficient, with the use of ULONGLONG data type, which is far faster than computing by 'double' type. Please refer to Appendix V for the details of implementation.

7.3.4 Two Improvements

Firstly, the original throughput time is calculated in a very inaccurate way. There, the start and end time of processing each node are recorded. After all nodes have been iterated, the latest end time among all nodes is subtracted from the earliest start time among all nodes. This produces the processing time of all nodes processing packets in parallel which is then accumulated.

However, it uses a random number generator to produce the start time for each node:

```
StartNode_Time = (double) rand()/(double)(RAND_MAX * SCALE_FACTOR);
```

Here, SCALE_FACTOR is set to 1000.0. The scale of StartNode_Time is about $\frac{1}{1000}$ of the unit of throughput time. So after the accumulation of hundreds of thousands of rounds, it was detected to be the major contribution to the throughput time. In other words, in the expression $TP = \frac{DP}{PT}$, where *TP*, *DP*, and *PT* are the throughput of network, number of packets that have reached destination, and the total processing time taken by all nodes, respectively, the result is mainly composed of the accumulation of difference of randomly generated numbers. Some statistical variables were added to measure the time contributed by random number generator, and the result confirms this conclusion.

To patch up the problem, a new method is used in this project. It records the total time of processing all nodes. Let it be *T*. Then the throughput time is calculated as $\frac{T}{N}$, where *N* is the total number of nodes. Here, *T* is effectively the simulation time specified in the input file. Maybe in the final iteration, some nodes have been processed while some have not. However, as the total number of iteration is very large, such a minor difference can be neglected. The experimental result shows that as long as the simulation time is long enough and thus $\frac{T}{N}$ is large enough, the throughput fluctuates in a very small range, such that no result is discarded by the 95% confidence interval technique (see Section 7.4). The assumption underlying this model is that all nodes always run in parallel.

The second problem is that nearly 10 percent of the packets are lost halfway. Actually, when the neighbor's transit queue and local output queue are both full, the packet is not added to the injection queue due to a mistake in programming. The prototype of the function is: void Requeue(CPacket *Target,CPacket **Packet) and it is called by:

```
Requeue (Node[CurrentNode].NodeQueue->Packet, &(Node[CurrentNode].
        CentralBuffer));
```

Obviously, the pointer to central buffer is copied to the formal parameter of Requeue, instead of the real parameter Node[CurrentNode].NodeQueue->Packet. This causes the loss of packet and leakage of memory. A simple way to fix the problem is to call by reference the first parameter of Requeue, i.e. the prototype is changed into:

void Requeue(CPacket *&Target, CPacket **Packet).

Now, the debugger of Visual C++ reports no memory leakage and the batch mode proposed by Yan Yan [22] can be run safely.

7.4 Filter of simulation results

The confidence interval check is used in processing the simulation results. This technique is more necessary in incomplete cubes than in binary hypercube, Folded Cubes or Josephus Cube. The reason is that the incomplete cubes are not stable networks. Here stable network is defined as follows:

(*Definition* 7.1) Stable Network

For any node address p in network N, if all nodes $x \in N$ are re-labeled as $x \ XOR \ p$, the new network N_p is isomorphic to the original one, then we call network N as Stable Network.

Obviously, binary hypercube, Folded Cubes or Josephus Cube are all stable networks. As most routing algorithms are based on XOR operation, it can be easily proved that in stable networks, for any node address p, a faulty node located at x is equivalent to being located at $x \ XOR \ p$, while any faulty link (x, y) is equivalent to (x XOR p, y XOR p).

Thus, the location of faulty components is less important for stable networks than for Fibonacci-class Cubes. In other words, in the latter class of networks, simulation result might change noticeably due to the location of faulty components. Therefore, the confidence interval check is more useful to ensure the result is representative of the given situation setting.

An example in point is node 0^n in an *n*-dimensional Fibonacci Cube. The main idea of routing in FC is basically as follows: invert all 1's in preferred dimensions to 0 and then invert all 0's in preferred dimension to 1 [13]. As such, if node 0^n is faulty, the influence will be far more significant than if node $(10)^{\frac{n}{2}}$ is faulty.

Each time simulations runs, five sets of results are generated with each simulation run and each set of result takes about 60 seconds. Each set of result is generated by different simulated network that has random distribution of faulty nodes and/or faulty links if the total number of faulty components is specified.

A 95% confidence interval is based on the *n* results. Denote the *n* results as $x_1, x_2, \cdots, x_n$. Then define the mean of them as $m = \frac{1}{n}\sum_{i=1}^{n} x_i$, and standard deviation $s = \sqrt{\sum_{i=1}^{n}(x_i - m)^2}$.

The simple *z*-test is by defining $z(x) = \frac{x - m}{s / \sqrt{n}}$.

As for 95 % confidence interval, define a real number $z_{0.95}$ such that

$$\int_{-z_{0.95}}^{z_{0.95}} e^{-\frac{x^2}{2}} dx = 0.95$$

Then, the 95% confidence interval is defined as $\{x \in R \mid |z(x)| < z_{0.95}\}$, or equivalently,

$(m - z_{0.95}\frac{s}{\sqrt{n}}, m + z_{0.95}\frac{s}{\sqrt{n}})$. Here $z_{0.95} = 1.96$. The consequence is:

for $\forall x \in (m - z_{0.95}\frac{s}{\sqrt{n}}, m + z_{0.95}\frac{s}{\sqrt{n}})$, the probability $P(-1.96 < \frac{x-m}{s/\sqrt{n}} < 1.96) > 0.95$.

To analyze the result, we discard all results that are located outside the 95% confidence interval.

7.5 Comments from the perspective of Software Engineering

The new simulator is organized in a very different way from the original version. In one word, it is object oriented. That brings a lot of convenience for programming because the routing strategy is unified for all Fibonacci-Class Cubes. To demonstrate the benefit, it is good to see the definition of class: CExtFibCube, which is a class for Extended Fibonacci Cube.

```
class CExtFibCube : public EnhFibCube       // inherit from Enhanced Fibonacci Cube
{
public:

    CExtFibCube(int dim, int sub, int nodeFault, int linkFault, int distribution,
                    CString *Doc);

        virtual ~CExtFibCube();

protected:
        virtual bool CheckValid(unsigned x, int digits = Num_Bits);
        virtual unsigned Fib2Dec(unsigned x, unsigned digit=Num_Bits);
        virtual unsigned Dec2Fib(unsigned x, unsigned digit=Num_Bits);
```

```
        unsigned k;             // subscript of XFC_k(n)
};
```

Only four functions need to be overridden for this new class inherited from Enhanced Fibonacci Cube. They are CheckValid, Fib2Dec, Dec2Fib, and the construction function. All other functions that are not 'virtual' can be inherited and used with no change. See the definition of EnhFibCube below.

```
class EnhFibCube
{
public:

        EnhFibCube (int dim, int nodeFault, int linkFault, int distribution, CString *Doc);
        virtual ~EnhFibCube();
        void Run(CWnd *win,CDC *pDC);

protected:

        //  Shared functions
        void Clear();
        unsigned OneBest(unsigned source, unsigned destination, unsigned x2, unsigned
                        DT, int *m);
        unsigned GetNext(unsigned int source, unsigned int destination, unsigned int
                    available, unsigned int *DT);
        void BuildPacket(void);
        unsigned char CalDimOrder(unsigned current, unsigned char *orderDim,
                                unsigned char *inverseDim);
        void Initialise_Dimmap(unsigned current, unsigned char *mapDim, unsigned
                            char total, unsigned char *map);
        void Simulate(CDC *pDC);
        unsigned countPos(unsigned current);
        void Initialise_Node(void);
        void Initialise_StatParams(void);
        void BuildFault(void);
        unsigned GetNeighbor(unsigned available, int dimension);
        void Initialise_Network(void);

        //  Only three virtual functions that need to be overriden by sub-classes
```

```
        virtual bool CheckValid(unsigned x, int digits = Num_Bits);
        virtual unsigned Fib2Dec(unsigned x, unsigned digit=Num_Bits);
        virtual unsigned Dec2Fib(unsigned x, unsigned digit=Num_Bits);

protected:

        //  attritbutes
        CString *report;
        unsigned *Link1;
        unsigned *Link2;
        unsigned *Fault;
        unsigned * FibNum;
        unsigned Node_Availability;
};
```

The structure of the whole program is therefore more streamlined and modular. Actually, it can serve as a base class for many incomplete hypercubes. Besides, the code is now scattered in several files and classes thus it is more convenient to manage.

Another improvement of organization is extracting all globally accessed variables and functions such as random distribution functions into one file (Common.h). Then it can be included into the implementation file (.cpp files) of other classes if necessary.

Chapter 8: Analysis of Simulation Results

8.1 Introduction

Using the completed simulation tool, the performance of FTFR in term of network efficiency, can be measured by network efficiency. This chapter summarizes the simulation procedure, analyses and compares performance in terms of average network latency, mean throughput with respect to network dimension, network topology and faulty component number. The raw data collected are placed in Appendix VI. Comparison diagrams illustrated in this chapter comply with the raw data.

To make a fair comparison, several factors are fixed concerning the simulation procedure, environment and result selection:

- All simulations must be run on a same computer. In this experiment, the Intelligent System Laboratory PC 8, DELL CPU 2.0GHz and Physical Memory 512MB is used.
- During the simulation process, all other non-system applications must be shut down. The network line is also disconnected to ensure no hidden CPU uses of Internet applications.
- Each set of input parameters must ensure that the CPU is running at 100% usage. This is to ensure that no swap in and out for virtual memory occurs. Otherwise, the timing will be very inaccurate because communicating with hard disk is of several orders slower than accessing physical memory. The upper bound of dimension is determined by this requirement.
- Each set of input parameters is simulated for 5 times, with each time lasting 60 seconds for network communication. Note, the 60 second is not how long the

simulation program runs, because of the overhead for simulation tool in addition to the useful communication simulated for the network.

- Uniform probability distribution is adopted for packet injection probability function and applies to all cases.
- The average network latency and mean throughput for the 5 simulations are calculated at the end of the program, together with their respective standard derivation.
- Simulation starts from network with dimension $n = 5$, since we are not interested in small size networks.
- The 95% confidence interval or 5% significance level is used for filtering undesired or deviating results.

8.2 Technical considerations for accurate simulation

8.2.1 Traits of expected result

Since we are simulating a very large number of packets within one round, it is naturally expected that the result of 5 rounds for a given set of input parameter should not fluctuate too much, i.e. the standard deviation should not be too large. Secondly, with dimension increasing, the network latency is to increase due to the longer path while the network throughput is also supposed to increase thanks to the increasing parallelism available. Thirdly, with the number of faulty components increasing, the latency is to increase and throughput to decrease. These are expected results and we will verify them in the following sections.

8.2.2 Buffer size

The current buffer size is set to 10, i.e. the maximum length of the transit queue for each dimension at every node is 10. This number is closely related to the likelihood of deadlock occurrence. If it is set to be small, it is more likely to bring about deadlock while setting it too big will cost more memory space because more packets will be generated. The influence of buffer size will be further discussed later.

8.2.3 Hop time

The hop time can be specified in the input file. However, in all our simulations, it is set to 500ns for a fair comparison. This value is determined empirically based on C104. Similar trends are also observed by varying hop times. In the network routing problem, there is a trade off between the path length and time for making routing decisions. The more intricate the decision making process is, the more time it takes, but possibly the shorter the final path will be. Conversely, a decision made quickly tends to result in longer path. If we set the hop time longer, the final result will more reflect the difference in path length while setting it smaller will make the time for running the routing algorithm more dominant. In FTFR, the routing algorithm is fixed, so the choice of hop time will not influence the final result much. If we set hop time longer, the difference between the decision making time for using spare dimensions and using preferred dimensions will be less significant.

8.2.4 Simulation duration time

How long the simulation should run is an important problem. In our simulation, the maximum possible number of allowable packets is:
(Total Links – Faulty Links – Number of links incident to Faulty Nodes)×Buffer Size.

For small and medium sized networks, they get saturated with packets shortly after the

beginning of simulation. Before saturation, the latency must be shorter than the stable value and the throughput lower. However for large sized networks, the network gets saturated very slowly. It was observed that after 60 seconds, the metrics do not converge. Setting the simulation duration longer will alleviate the problem. However, since one simulation duration is applied to all cases, it is not worthwhile to double or even triple the simulation time just for a few extremely large dimensions. Thus, for such irregular cases, they are deleted from the final valid data set. This point will be discussed later.

8.3 Comparison of FTFR's performance on various network sizes

In this section, FTFR is applied to fault-free regular Fibonacci Cube (*FC*), Enhanced Fibonacci Cube (*EFC*) and Extended Fibonacci Cube (XFC_k) and binary hypercube. The throughput and latency of them are shown in Figure 8.1 and 8.2 respectively.

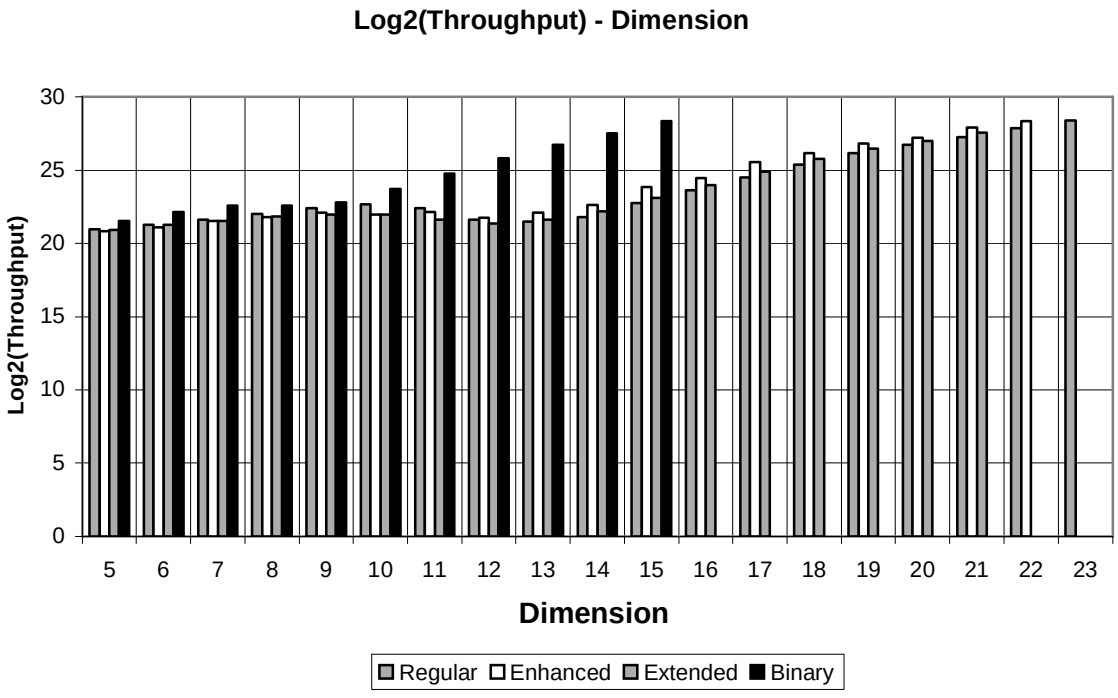


Figure 8.1 Throughput (logarithm) of Fault-free Fibonacci-class Cubes

In Figure 8.1, it is demonstrated that the throughput of all networks is increasing as the dimension is increased from 12 to over 20. This is due to the parallelism of the networks and the increase in the number of nodes (where n is the dimension) that can generate and route packets in the network, is faster as compared to the time complexity of $O(n\log n)$. By increasing the network size, the number of links is also increasing at a higher rate than the node number. This in turn increases the total allowable packets in the network. With parallelism, more packets will reach destination in a given duration. For the same reason mentioned in the previous discussion of latency, Enhanced Fibonacci Cube has the largest throughput among the three types of Fibonacci-class Cube. An interesting observation is that for dimensions between 11- 13, the throughput decreases for a two dimensions and increases again afterwards. One possible explanation is: the complexity of FTFR is $O(n\log n)$. For large n, the variation in $\log n$ is small compared to the case of small n. Thus the difference brought by $\log n$ will be small and the trend of throughput is the same as what an $O(n)$ routing algorithm produces. For small n, however, the contribution of $\log n$ is comparable with the increase rate of networks size, which leads to the seemingly irregularity. On the other hand, when dimension is small, the network scale is too small to display that characteristic. For Fibonacci Class Cube, the irregular range is 11-13, while for binary hypercube, such a range is 8-9. This again accords with $12:8.5\approx 2:\frac{1+\sqrt{3}}{2}$. Note the simulation for binary hypercubes with dimension over 15 is not carried out because there is no enough physical memory on the computer.

It is guaranteed that FTFR is cycle-free. But in the face of concurrency, does it guarantee deadlock-freeness? It is clear that if we decrease the parameter BUFFER_SIZE, the deadlock problem will become more evident if the routing algorithm is not deadlock free. When BUFFER_SIZE is set to 10, the irregular range is 11-13. When BUFFER_SIZE is

reduced, the range will move leftward (decrease). When BUFFER_SIZE is 1, such an irregular phenomenon will disappear. These reflect that FTFR is possibly NOT deadlock-free. As the BUFFER_SIZE is reduced, networks of even smaller dimensions will suffer from deadlock. Once deadlock occur, it will make a significant contribution to the packet latency. This in turn will make the irregular range caused by $O(n \log n)$ complexity less apparent.

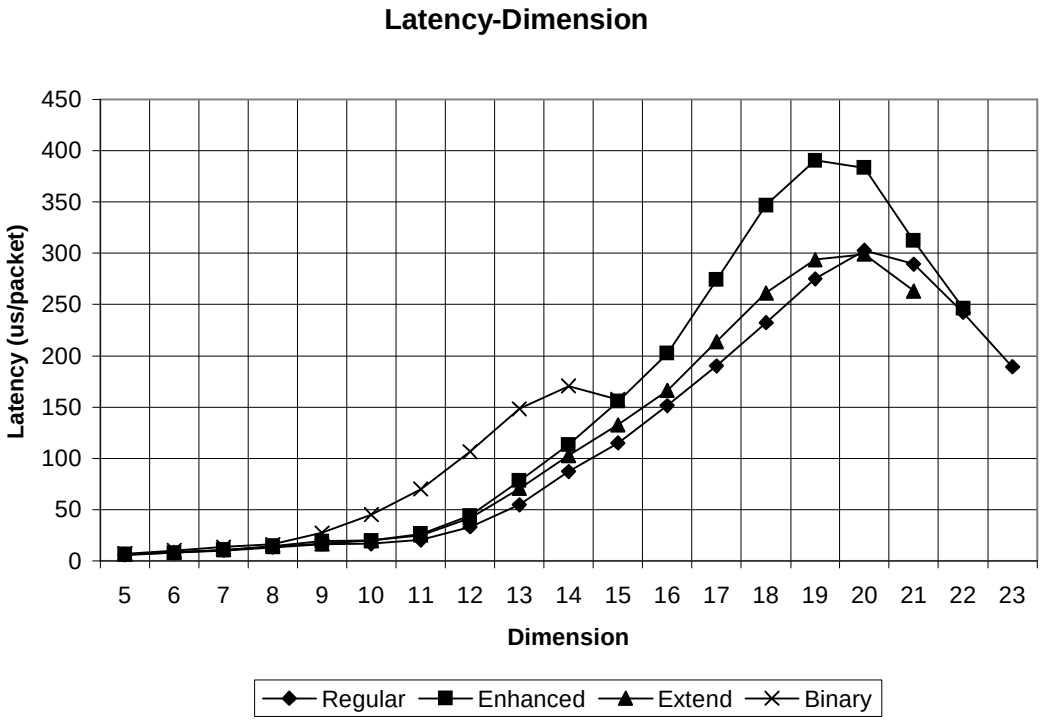


Figure 8.2 Latency of Fault-free Fibonacci-Class Cubes

In Figure 8.2, it can be observed that the average latency of regular/ Enhanced/ Extended Fibonacci Cubes increases as the networks dimension increases below 19. As the network size increases, the diameter of the hypercube also increases. A packet to be transmitted has to take a longer path to reach its destination, resulting in a higher average latency. The Enhanced Fibonacci Cube has the highest latency among three because when dimension is large enough, the number of nodes in Enhanced Fibonacci Cube is the largest among regular/Enhanced/Extended Fibonacci Cubes of the same dimension.

After the dimension reaches 19 or 20, the latency decreases. This is because the scale of the network becomes so large that the simulation time is insufficient to saturate the network saturated with packets. This is evident from the fact that for those dimensions, the number of packets reaching destination is lower than the total allowable packet number. So the packets in these networks spend less time waiting in output queue or injection queue, while that portion of time (incurred by concurrency) comprises a large part of latency for low dimensional networks that get saturated with packets in the simulation duration. A straightforward solution is to increase the simulation time. However, to make comparisons fair, the simulation time for other cases should also be increased proportionally. This will double or even triple the total time for simulation. As 19-20 dimension is already adequate for demonstrating the performance of FTFR, this effort is spared. Binary Hypercube, a special type of Extended Fibonacci Cube, demonstrates a similar trend, with latency beginning to decrease since 15. This also goes well with the fact that the number of nodes in Fibonacci-class Cube is $O((\frac{1+\sqrt{3}}{2})^n)$ and the node number of binary hypercube is $O(2^n)$. $\frac{1+\sqrt{3}}{2}:2\approx 15:20$. Note here that due to the insufficiency of physical memory, no simulation is carried out for binary hypercubes with dimension over 15.

8.4 Comparison of FTFR's performance on various numbers of faults

In this section, the performance of FTFR is measured by the varying the number of faulty components in network.

The result for $XFC_{13}(14)$ is as follows:

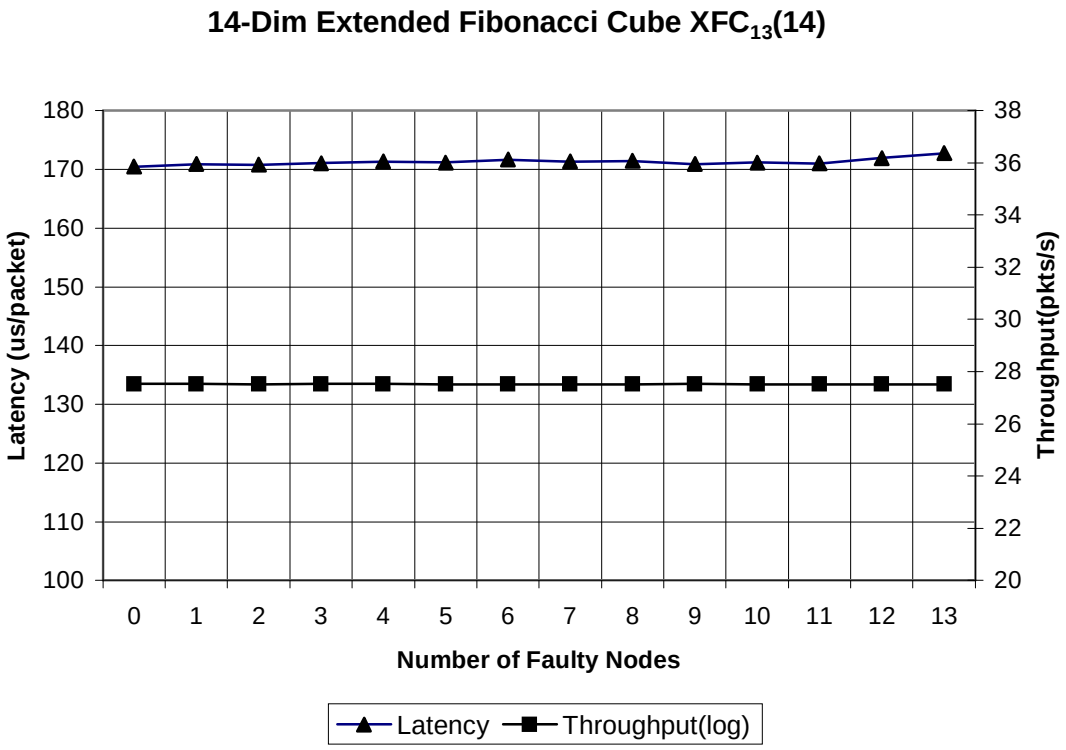


Figure 8.3 Latency and Throughput (**logarithm**) of 14-dim Extended Fibonacci Cube

It is clear that when the number of faults increases, the trend of average latency is to increase while the throughput is to decrease. This is because when more faults appear, the packet is more likely to use spare dimensions which makes the final route longer. In consequence, the latency increases and throughput decreases. However, there are some exceptional cases when the existence of faults reduces the number of alternative output

port available, and thus expediate the routing decision. The influence of different faults number is more evident when the network size is small. With fixed number of faults, there are fewer paths available for routing in smaller networks than in larger ones. Thus making some of the paths unavailable will bring about more significant influence on the former. While in large networks, with the total number of nodes in n-dimension network being $O((\frac{1+\sqrt{3}}{2})^n)$ and maximum faulty component number tolerable being $O(n)$, the influence of faulty components will bring about less and less significant influence on the overall statistical performance on the network. That explains why the throughput and latency fluctuate in Fig. 8.3. Nevertheless, the overall trend is still correct despite the glitches.

However, as the number of faults tolerable in Fibonacci-class Cubes of order n is approximately $\left\lfloor \frac{n}{3} \right\rfloor$ or $\left\lfloor \frac{n}{4} \right\rfloor$ [12][14][15], we have to use networks of large dimension to provide a large enough number of faults for comparison. That makes the underlying trend less likely to be evident in the experimental results. The following figures present the result for 20-dimension regular Fibonacci Cube, 19-Dim Enhanced Fibonacci Cube, 18-Dim Extended Fibonacci Cube.

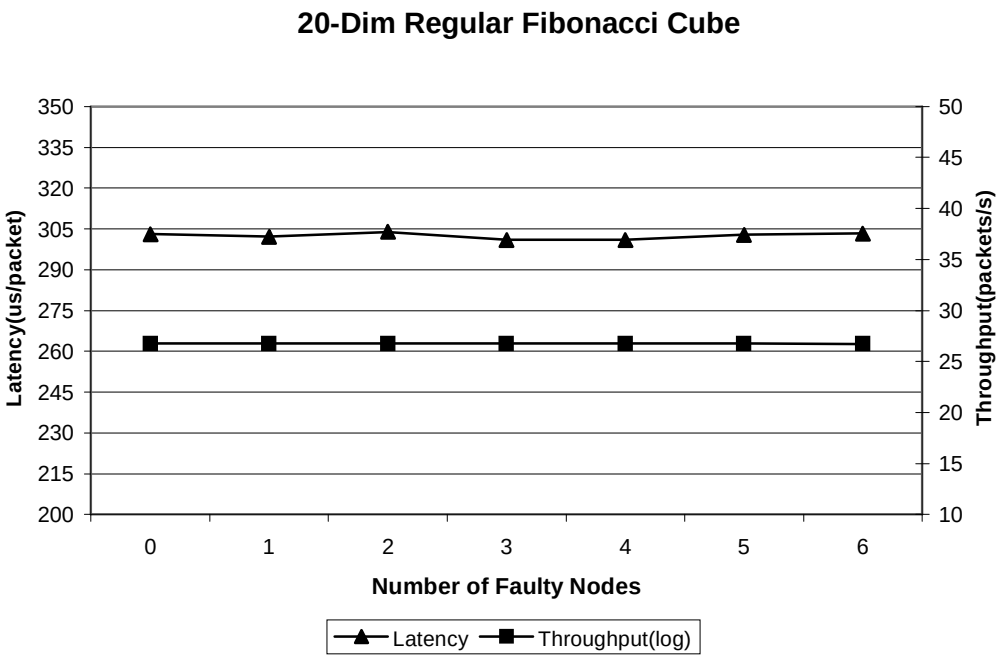


Figure 8.4 Latency and Throughput (**logarithm**) of faulty 20-Dim regular Fibonacci Cube

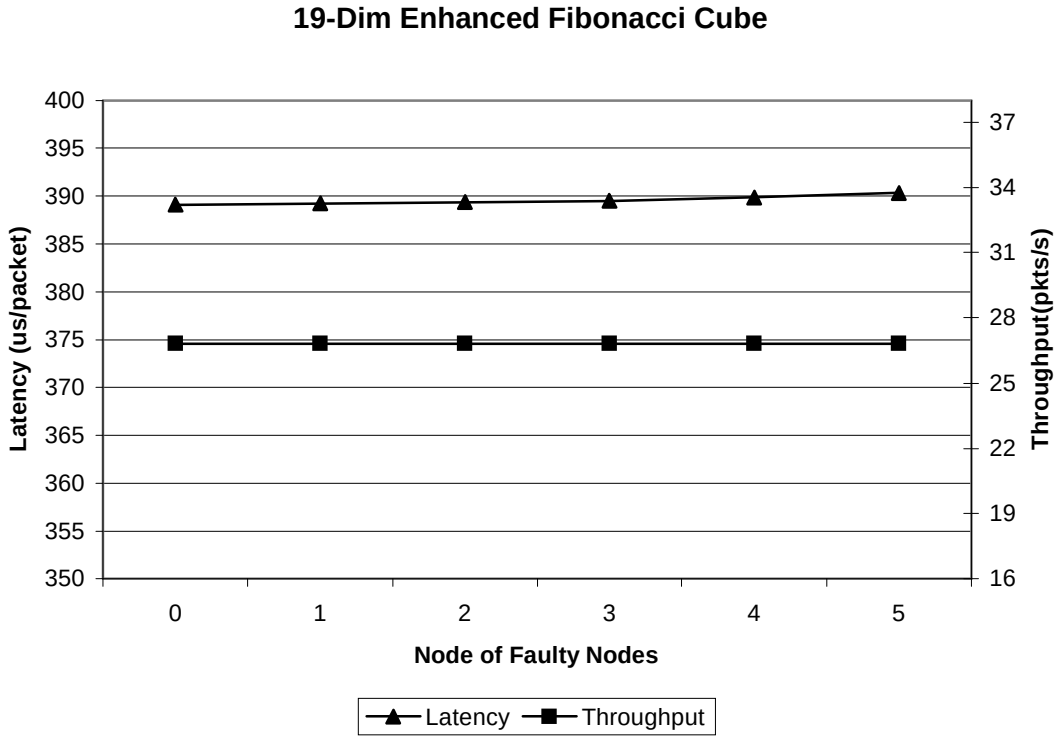


Figure 8.5 Latency and Throughput (**logarithm**) of faulty 19-Dim Enhanced Fibonacci Cube

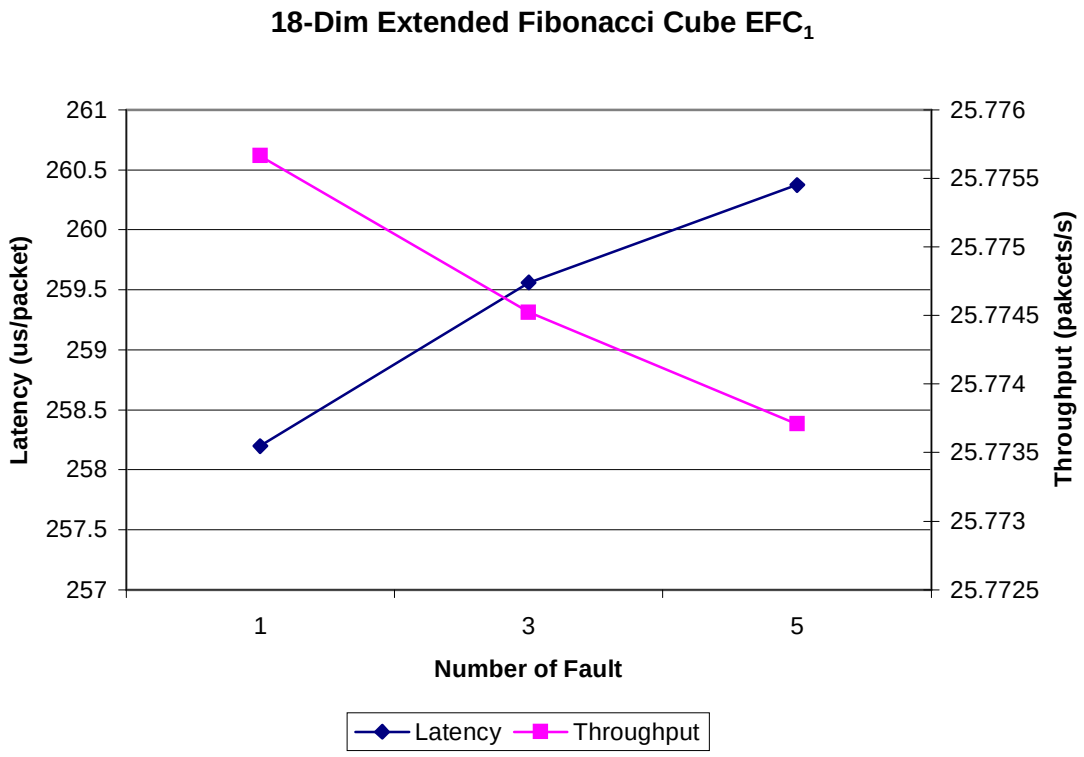


Figure 8.6 Latency and Throughput (**logarithm**) for faulty 18-Dim Extended Fibonacci Cube

The fluctuation of the result is actually needs to be examined carefully. For example, the latency in Figure 8.6 varies only in the range of below 1%. We know that with different simulation reading, the fault location is randomly distributed. Similarly, messages generated have different destinations based on the uniformly distributed packet destination. If we examine the standard deviation of the result, it is shown that such a small variation in Figure 8.6 is not too much outside the 95% confidence interval for any situation. Thus, it is more reasonable to focus on the trend of the statistical results, instead of the exact number.

8.5 Results of *Gaussian Cube*

The simulation results of *Gaussian Cubes* display very similar trend and properties as in Fibonacci-class Cubes. Thus in this section, we only present the Figures that are drawn based on the simulation result. The only thing that deserves attention is that the location of faults in the Gaussian Cube is very important. So different from the simulation scheme in FTFR in which we only specify the number of faults and randomly distribute the faults, now we specify the location of the faults. In this simulation test, we see how the faulty node located at 0^n influence the system performance.

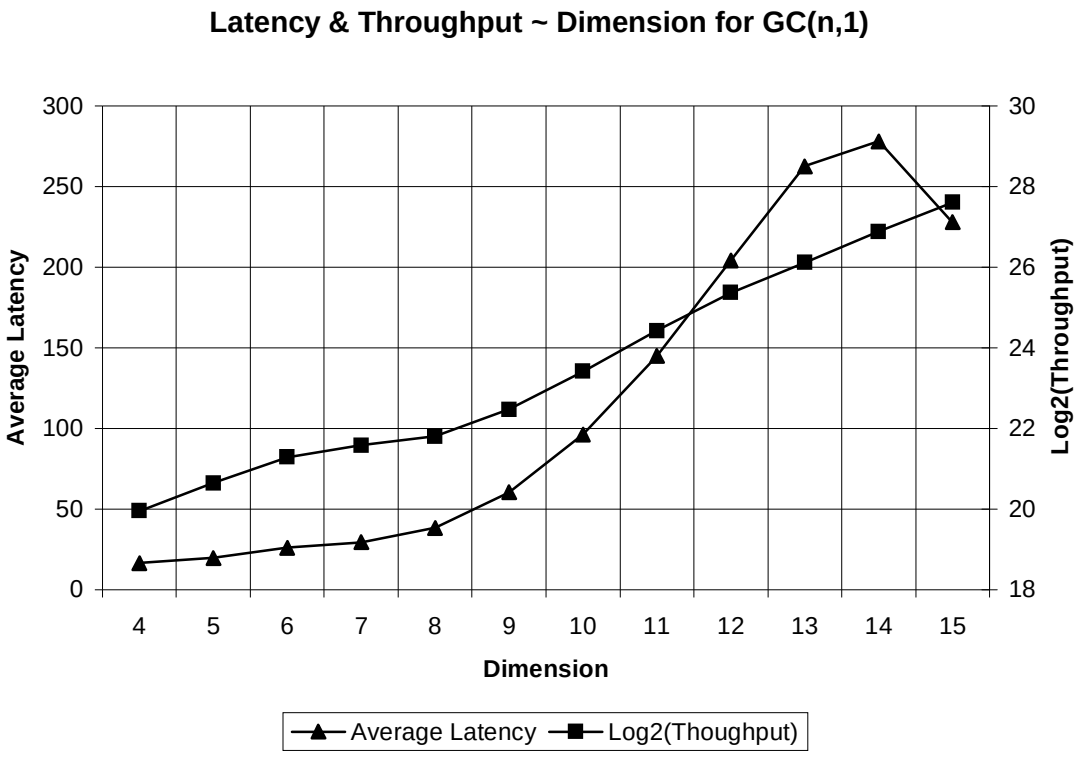
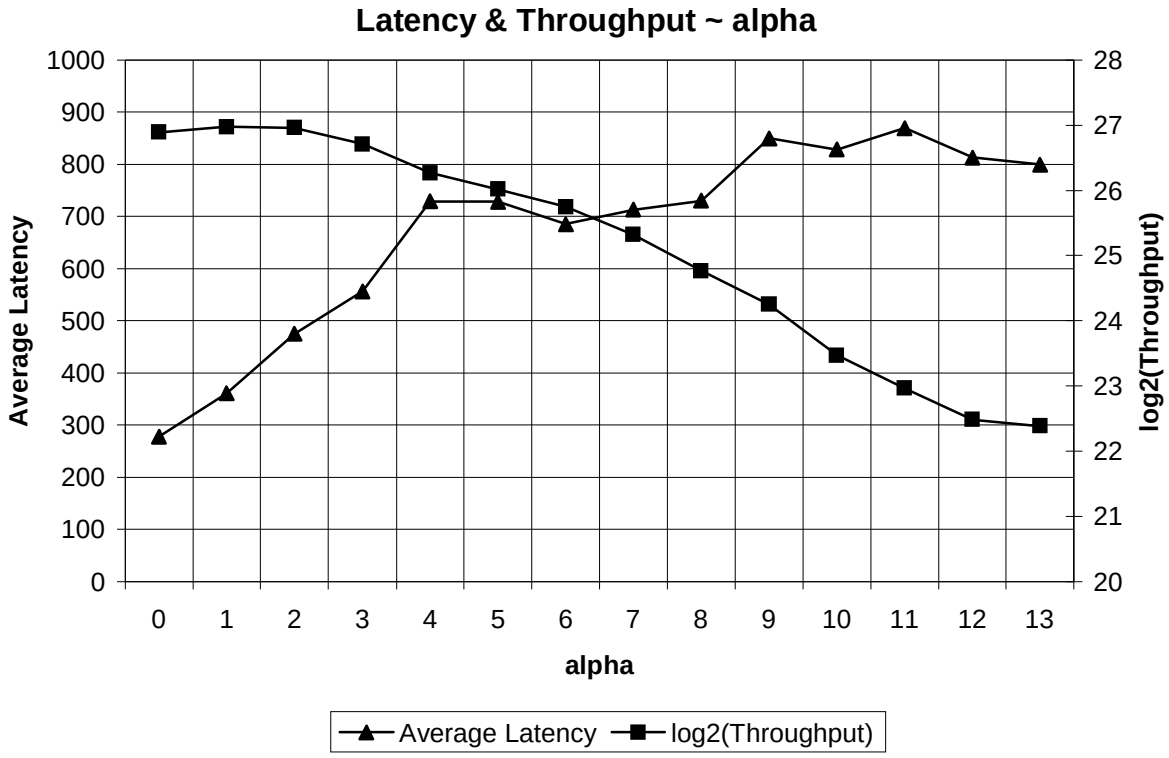


Figure 8.7 Average Latency and $\log_2$(Throughput) versus dimension for $GC(n,1)$

Since the algorithm's complexity includes a term logα and does not include log n, it is satisfying to see that the temporary decrease interval for average latency does not appear

in Figure 8.7. However, such an interval does appear again in Figure 8.8, where the *x*-



axis is α.

The following two figures (Figure 8.9 and 8.10) illustrate the influence of faulty node 0^n on the network average latency and throughput, respectively. The discussions (including the effect of glitch) in FTFR also apply to Gaussian Cube.

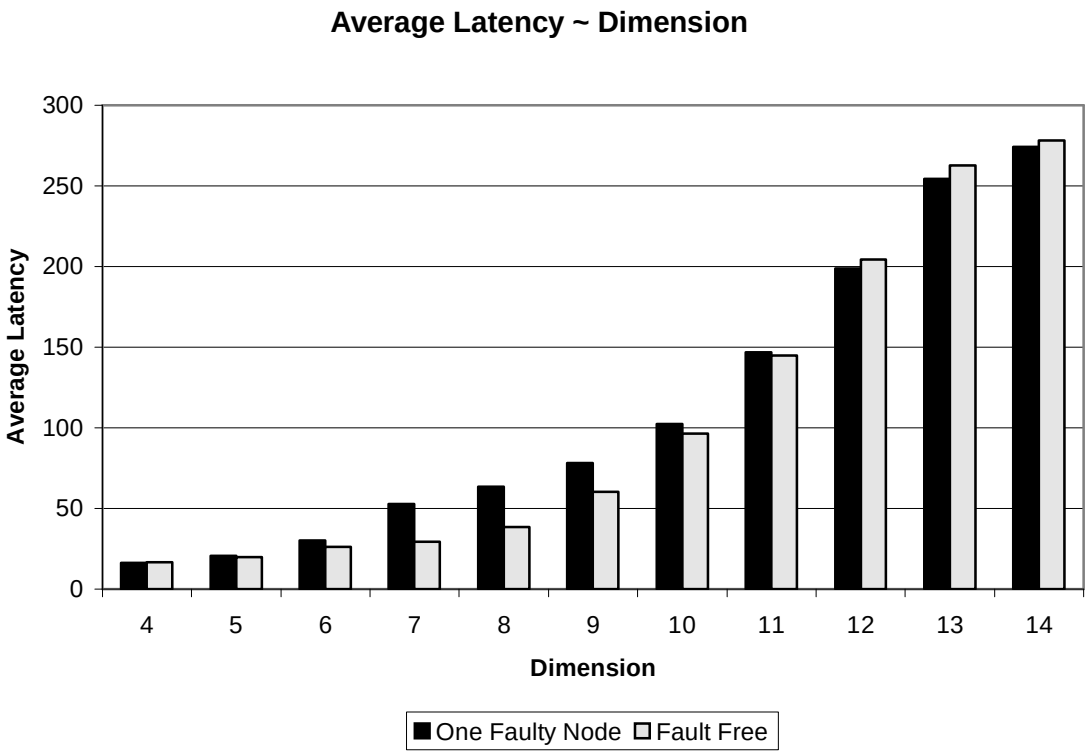


Figure 8.9 Influence of faulty node 0^n on network average latency

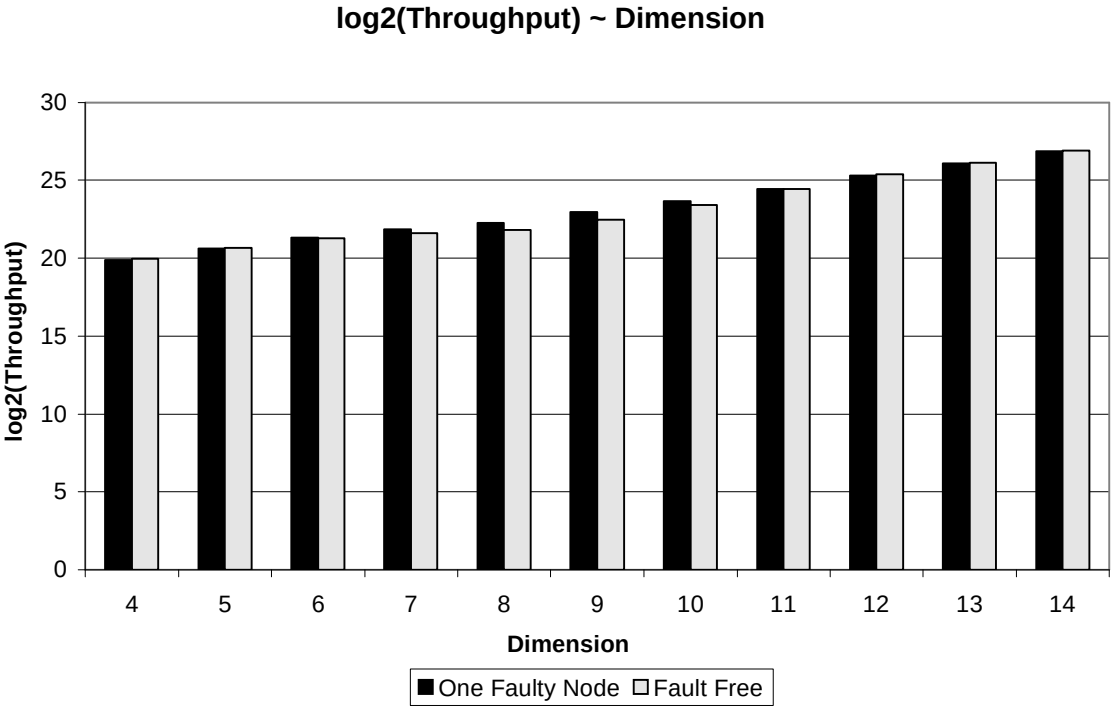


Figure 8.10 Influence of faulty node 0^n on network throughput

Chapter 9: FPGA Implementation of FTFR

9.1 Background

From the experience of software simulation, it is evident that the strength of software applications is the ability to be easily changed to suit customer demands. However, inevitably, hardware applications of the same are always much faster, but the tradeoff is its lack of programmability and reconfigurability. With the advent of high-density, high-performance and low-cost Field Programmable Gate Array (FPGA) that can be easily reconfigured, the situation had since changed. It promises to give vendors an added edge in supplying custom-made applications to suit the customers' varied requirements in shorter product development cycles and lower costs substantially, by using the latest software technology and design flows such as Celoxica DK1 [71]. The commercial potential is indeed enormous. Figure 9.1 demonstrates the design flow of DK1 *software-compiled* system design [65].

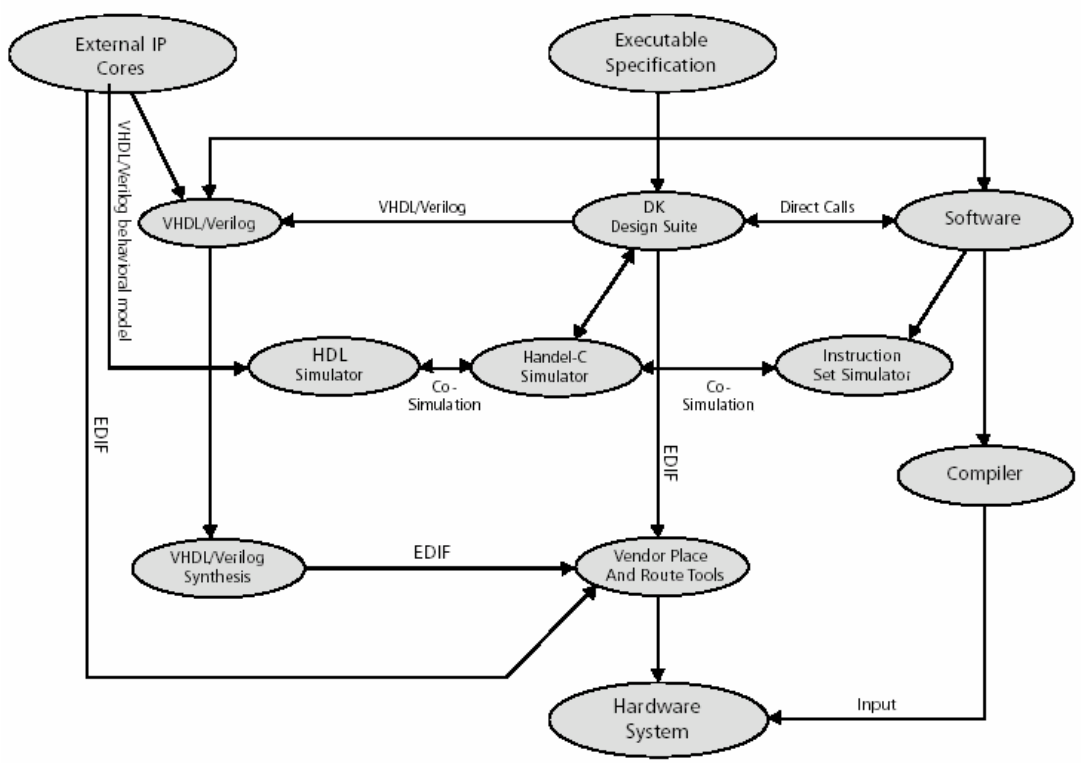


Figure 9.1 DK1 Design Flow

Celexica DK1 Design Suite, which is used in this project, enables direct migrating designs to hardware without requiring the generation, simulation, or synthesis of hardware description language. It uses the unique language Handel-C and the design suite focuses on the design, validation, iterative refinement and implementation of complex algorithms in hardware. Handel-C, an ISO/ANSI-C based programming language can be used to express algorithm without worrying about how the underlying computation engine works [66]. This philosophy makes Handel-C a programming language rather than a hardware description language. In some senses, Handel-C is to hardware what a conventional high-level language is to microprocessor assembly language. The output of the compiler is an architecture optimized EDIF netlist appropriate for FPGA or PLD devices, or RTL VHDL for existing tool suites. Thus, due to its high level nature, Handel-C has made it possible for the same person to do both software and hardware implementation, which greatly reduces the manpower and development costs.

Besides, a readily available development board, the RC100, also made by Celoxica, can be used to physically implement and test the designed router for this project. It features a high-performance Xilinx Spartan-II FPGA, with 200,000 system gates, 5,292 logic cells and 1,176 CLBs. It has a maximum of 284 user I/O and 56K block Ram Bits. System performance is supported up to 200MHz. As the centerpiece of the board and main reconfigurable logic that users can target, the FPGA is directly connected to [67].

- Two SSRAM banks
- Flash RAM (8M Bytes)
- Video DAC
- Video Input Decoder
- PS/2 connectors (Mouse and Keyboard)
- CPLD
- LEDs
- Two 7 segment displays

The Xilinx CoolRunner CR3128XL CPLD, which is used to configure the FPGA from various data sources and implement other glue logic, can configure the FPGA

with data received from host PC via File Transfer Utility or with a configuration file retrieved from the Flash RAM. The structure of RC100 board is showed in Figure 9.2-9.3 below.

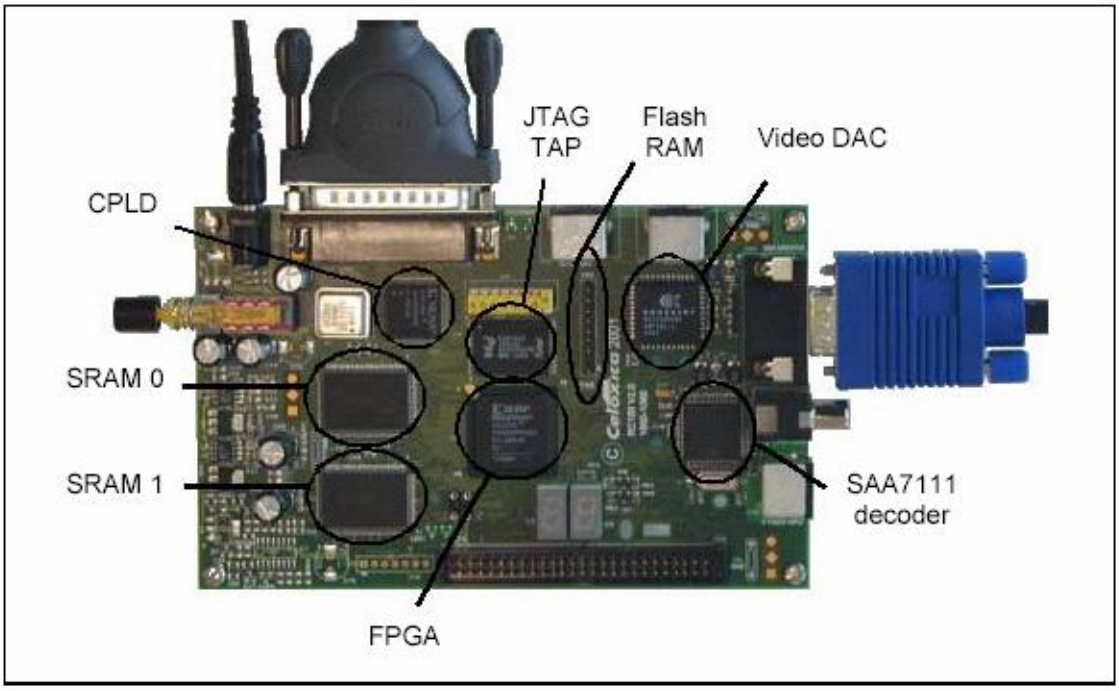


Figure 9.2 RC100 Board Components

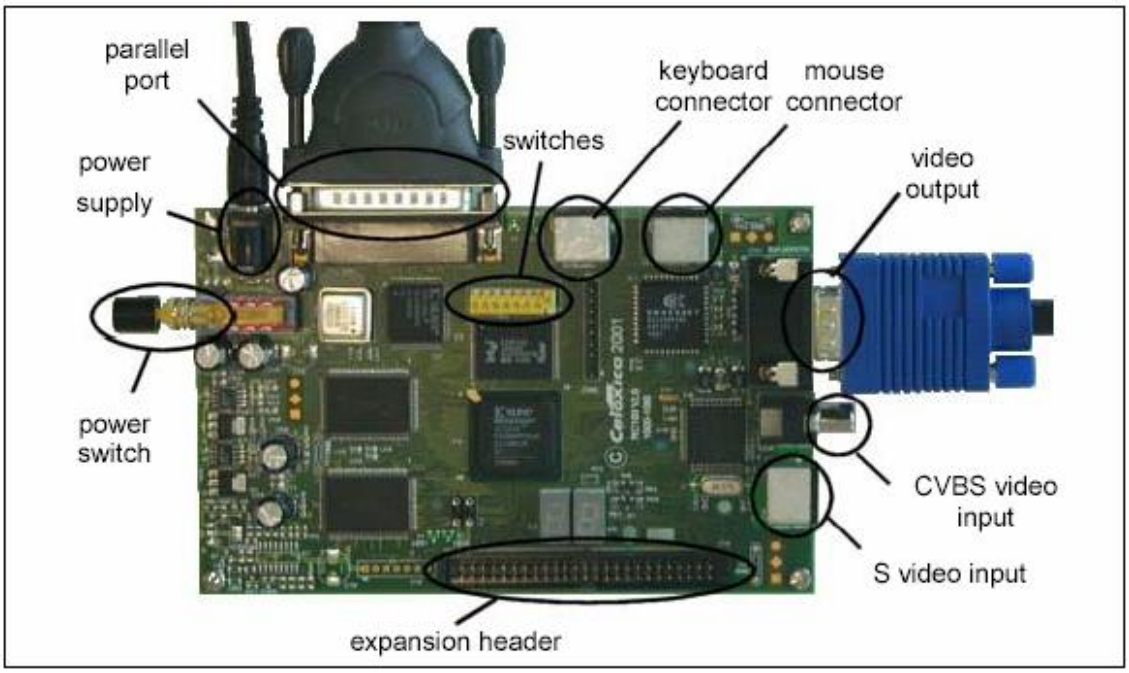


Figure 9.3 RC100 Development Board

Via parallel port cable, the File Transfer Utility can be used to [69]:

- Transfer Xilinx BIT files to the FPGA
- Transfer files or raw data from PC to a specified location in the Flash RAM
- Transfer files or raw data from the Flash RAM to PC

9.2 Overview of Experimental Methodology

Our objective is to obtain a circuit that implements FTFR correctly and efficiently. Obviously, two aspects are of our major interest:

- Correctness. The router must produce the correct decision that FTFR generates.
- High performance. This involves DK1 gate count, number of logical components (Luts and FFs), number of Slices/Routes, PAR timing and maximum clock frequency.

Therefore, we divide the experiment into two stages, namely software simulation stage and hardware implementation stage.

In the software simulation stage, we focus on programming the design in DK1, using Handel-C. It is easy to check the result because *chanin* and *chanout* can now be extensively utilized to show the value of critical variables directly, making debugging and verification of code correctness very simple. This stage is just like software development, with focus on the correctness of our program. Besides, DK1 Waveform Analyzer [72] can now be used to roughly estimate and analyze the performance of our router. Also the result of DK1 compilation can give the raw image of the relationship between total gate number and port number.

In the hardware implementation stage, the Celoxica DK1 Design Suite had to be set to compile the output file in EDIF format. When compiling in EDIF mode, DK1 would optimize away all unused code, i.e. those code that do not affect the final output. Similarly, if no meaningful output were specified, i.e. no I/O bus or Flash Ram specified, the design would not generate any EDIF files. Statements that could not be implemented in hardware such as *chanin* and *chanout* are required to be removed as well for error-free compilation. With optimized number of gates and LUTs (Look Up Tables), the generated EDIF file can be used by Xilinx Design Manager to generate BIT files, which is in turn downloaded onto the RC100 Development Board using Celoxica RC100 File Transfer Utility.

The performance indexes are easily available from the report of Xilinx Design Manager's implementation. However, without the availability of *chanin* and *chanout*, two problems arise: 1) how to initialize the data variable, 2) how to verify that the FPGA router was working correctly.

On the first issue there are three foreseeable solutions. Comparison is outlined in Table 9.1 [67][69].

S/No.	Input Method	Implementation Difficult	Additional Gate Counts	Multiple Test Data
1	Hardcoding	Easy	Negligible	Limited and Inflexible
2	Keyboard	Medium	Very Significant	Unlimited
3	Flash Ram	Hard	Acceptable	Nearly Unlimited

Table 9.1 Comparison of Input Methods

Since we are only testing the implementation, the number of additional gate count is of less importance because it will be finally removed after verification. If hardcoding is

used, then each time we want to test for a new set of data, the EDIF and BIT file will have to be regenerated, which costs a lot of time and thus inflexible. For keyboard, it needs the manpower of input each time. Flash memory can provide 8M byte space, which is enough for testing. If more testing cases are required, it is easy to transfer the testing data to RC100 again via FTU, which is far simpler and quicker than regenerating EDIF and BIT file. The strength of testing with Flash memory is that the process is automatic. A large amount of test can be carried out with no human interference. As for difficulty, both keyboard and Flash memory need additional conversion functions to change the data into binary or integer format for the router to execute, because these forms of input were in ASCII format, i.e. 0x30 represents 0b00, which is zero in integer terms. Furthermore, for numbers above 9, e.g. 10 that is 2 ASCII numbers of 1 and 0 in consecutive locations, a function would be needed to concatenate to their true value of 10.

In view of all, it was decided that Flash Memory is used for inputting the testing data. However, after the correctness is ensured, we have to remove the part for Flash Memory and adopt the real form of input. It is only at that stage can we take a fair comparison between the performance and scale of the router with respect to the port number. Moreover, it should be noted that due to the nature of DK1, designs of varying sizes would be generated for differing sets of data because of the optimization process. Thus, a fixed test case (extensible over various port sizes) would be hardcoded into routers of different port sizes so as to compare them in terms of gate counts, delay and maximum operating speed.

Going on to the issue of verification of the workings of the FPGA router, we need to be able to collect the output data generated by it. Five methods are proposed and the comparisons between them had been tabulated in Table 9.2.

S/No.	Output Method	Implementation Difficulty	Additional Gate Counts	Multiple Results

1	Direct File Output	Very Hard	Indeterminate	Indeterminate
2	VDU Display	Hard	Very Significant	Unlimited
3	Flash Ram	Medium	Acceptable	Limited by RAM size
4	Pin Outputs	Easy	Negligible	Unlimited
5	7 Segment Display	Easy	Negligible	Limited by display

Table 9.2 Comparison of Output Methods

Similar to the analysis before, taking account of the advantages and disadvantages of each method, it is decided that the Flash Ram Output option would be most suitable for use in saving multiple test results for routers of differing sizes. However, one would expect that the additional gate counts would limit the router that can be implemented on the FPGA.

Again, after the verification of the design, the Flash Ram functionality would be removed and replaced with the Pin Output options. This was because this option adds the least gates to the design and would be suitable when making comparisons for routings of differing port sizes in terms of gate counts, display and maximum operating speed.

9.3 Testing scheme

First, routers of different dimension will be compiled into different BIT files before the demo. It can be transferred to the RC100 by File Transfer Utility during the test.

For a fixed dimension, several testing cases can be designed in the input file. Then, they are transferred to RC100's Flash Memory. After the router makes decision, the result will also be recorded in the RC100's Flash Memory. Then, we use File Transfer Utility

again to transfer the result to PC and check the correctness. The format of input file that stores testing cases is as follows: (take a 3-dimension router as an example)

000	110	101
(Current node address)	(the availability vector for current node)	(input mask)

010 101 (the availability vector of the neighbors of the current node)
(the total number equals the number of 1's in the availability vectorof current node)

111	000
(destination)	(DT of the packet)

1 0 1 # (the history, meaning that the packet used dimension 1, 0 and 1 successively, the '#' signifies termination)

@ termination of the whole file

In real practice, the numbers above are in hexadecimal, so it is written as:

0 6 5
2 5
7 0
1 0 1 #
@

If several testing cases are to be used, the character '$' is used for separating cases:

0 6 5
2 5
7 0
1 0 1 #

$ // there is another testing case in the following

1 3 5

1 4

6 2

2 1 0 #

$ // there is another testing case in the following

0 5 5

3 4

3 1

0 2 1 #

@ // no more testing cases. File ends

The output will be arranged in the following format:

000 dimension = 2 DT = 0

001 Abort

002 Destination Reached

The first column is the testing case number. If the destination is reached, it will write "Destination Reached". If aborted, it writes 'Abort'. Otherwise, it outputs the dimension that is chosen to use, and the updated DT after the routing process.

Different testing cases can be posed and transferred to RC100 dynamically. This makes testing more flexible.

9.4 Result of implementation

	With Flash Memory for I/O	Without Flash Memory, use pin
NAND Gates after compilation	42558	11250
NAND Gates after optimization	32761	9627
NAND Gates after expansion	68997	36476
NAND Gates after optimization	15157	4039

Table 9.3 Comparison of NAND Gate Number between with/without Flash Memory for 4-dimension regular Fibonacci Cube using classical approach

	With Flash Memory for I/O	Without Flash Memory, use pin
NAND Gates after compilation	251952	226213
NAND Gates after optimization	117044	97743
NAND Gates after expansion	138142	109376
NAND Gates after optimization	48232	39285

Table 9.4 Comparison of NAND Gate Number between with/without Flash Memory for 4-dimension binary hypercube using FNN

9.5 Useful Tips for development

Although DK1 Development Suite provides a lot of freedom to FPGA design, its compiler is far from perfect. Some procedural tips are drawn from experience and are summarized in this section for future reference.

9.5.1 Error report problem

The errors reported by the compiler are very inaccurate, especially concerning the location. Some times, the real location of error and the place reported by the compiler may be a few hundred lines apart. To overcome this problem, the incremental debugging

approach is used. First, comment out most of the suspected parts of the program, leaving a small portion that is controllable. Now, it is easy to locate the problem and fix it in the small range. After that, release the commented parts little by little, with each round ensuring that no error occurs. The advantage is we can now focus the problems in the newly released parts, no matter where the compiler reports that the error exists. This method is proved very useful.

9.5.2 Runtime Error

This wired kind of error occurs during debugging. For example, if three sentences a, b, c are to be executed successively. If we debug it step by step, then maybe when executing b, a runtime error is reported. But if we place a breakpoint at c, then after executing a, we use 'go' or press 'F5' to run to the nearest breakpoint, c, the runtime error doesn't occur. This problems shows that Handel-C must have not encapsulated the lower hardware particulars completely, and problems in that level are looming in an unpredictable way. As this problem does not influence the final result we only need to pay attention to it and refrain from being stuck by this irregularity.

9.5.3 Compiling strategy

The time for compiling EDIF file is long for DK1. The time for Xilinx implementation is even longer. Therefore, we should use the debug mode as much as possible. It is only after ensuring that no logic error exists can we proceed to hardware implementation, during which, the only possible problem left is concerning hardware interface, or I/O utilities. This will be very helpful because debugging the program logic on RC100 is impossible. To save some time, we can set the option in Xilinx Design Manager to fastest, and then set it to optimal after ensuring no problem exists.

9.5.4 Programming methodology

It is not wise to write Handel-C in an object-oriented thinking. So it is free to use global variables. Besides, its unique macro expression is helpful in making the routine generic. Another important method to minimize the changes necessary for different port size is using macros like Num_Bits, Log_Num_Bits…. They can calculated the proper width of variables with respect to the port size. It is found that Handel-C is not excellent in processing stacks, so macro expressions is preferred to functions and recursion had better to be avoided.

As for loops, the traditional 'for(;;)' format is not welcomed in Handel-C. We had better use 'while'. As we often deal with an array in a loop, the following problem looms. The index for an array with the length of L is restricted to be $\log_2(L)$. However, to control the loop, we often need to use :

```
while( i  < L)
{
        do something on array[i].
        i++;
}
```

This is improper when L is power of 2. For example, when L=4, then the bit width of *i* is 2, so when *i* =3, after *i*++, *i* is 0. So *i* <L is always satisfied and a dead loop is formed. If we set the bit width of *i* to 3, then it can't be used as array's index. One compromising method is to set the width of the control variable *i* to $\log_2(L+1)$. Then in the 'while' loop body, use another variable of width $\log_2(L)$, say *ii*, to index the array. At the beginning of the loop body, let *ii* = *i* [$\log_2(L) – 1: 0$]. In this way, the problem is solved in a unified fashion. If the first sentence of the body does not quote *i*, then the assignment can be executed parallelly, incurring no extra time. To be economic, such a technique can be used only for those arrays whose length is power of 2. For other cases, the control variable with width $\log_2(L)$ can be used as index without any problem.

9.5.5 Design of common interface

To drive hardware on RC100 board, it is advisable to develop some higher-level interface libraries. The primitives provided by the RC100 are not powerful and are unwieldy to realize a useful function. It is helpful if some library functions or MFC-like encapsulated hardware calls be designed so that the following developers can program on a higher level and focus on problem-specific logics.

9.5.6 Floating point library

When implementing the router using fuzzy neural network, real numbers are used in addition to integer. Thus, floating point library is incorporated [73]. However, the library is not perfect and possibly contains bugs. One most significant problem is that when using floating-point numbers, the resource consumption for compiling is very huge, both in memory and in time, making it difficult to debug.

Thus, the strategy actually used in this implementation is scaling up. For example, if we calculate 1.5/0.3, then the result is same as (1.5*10)/(0.3*10) = 15/3. Of course, there exist some loss of precision if it is not wholly divided. This disadvantage is overcome by delaying division operation to the last step, because addition, subtraction and multiplication all result in no loss of precision. So avoiding division as intermediate steps can eliminate the accumulation of error. Besides, we used a scaling factor of 1000, as a result, the precision is very satisfactory.

Chapter 10 Conclusion

This chapter concludes the report by discussing the accomplishment, project limitations, and future work.

10.1 Conclusion

Fuzzy neural network has been successfully applied in many areas, such as clustering, prediction of time series, traffic and stock market, as well as automated control of large, complex systems. However, just as no model in artificial intelligence can apply to all applications, so does FNN. The problem in nature is that the application of routing in interconnection network is a based on binary discrete numbers. The FNN is heavily dependent on the clustering of each input (horizontal reduction). So it works efficiently in situations where the range of each input is large but the number of input is not too high. However, our binary application makes each input attached with two linguistic labels and the number of input is linear to network dimension. In consequence, the time and space complexity is exponential to the dimension. If we combine several independent binary inputs into one corresponding decimal value as input, then the number of linguistic labels required for each input will grow exponentially with network dimension. So it does not help.

On the other hand, an encouraging result is that efficient fault-tolerant routing strategies have been designed for such link/node diluted hypercubic networks as Gaussian Cube and Fibonacci-class Cube. They can tolerate more faults than the trivial bound of node availability. The simulation result demonstrated the desirable properties of these algorithms and the implementation on FPGA also shows the feasibility of physical manufacture.

Finally, it is proved theoretically that the Exchanged Hypercube can efficiently reduce the number of links from binary hypercubes, preserving nearly all topological and communication merits. The author believes that it is a promising type of network as a substitute for binary hypercubes in many applications.

10.2 Accomplishments

In reviewing the purpose of this project as defined in section 1.2, the author has illustrated that the fuzzy neural network is not suitable for the problem of routing in interconnection network, at least at present. An encouraging result is that despite the intrinsic low node availability in node/link diluted hypercubic networks, still a fairly high number of faulty components can be tolerated by our fault-tolerant routing strategy. The simulation result also shows that the performance of our algorithm is reasonable.. Besides, it is demonstrated that the implementation of it on hardware such as FPGA is feasible.

The *Exchanged Hypercube* provides one more possible topology when constructing multi-computer systems.

10.3 Project Limitations

Although extensive experiment on the Fault-tolerant Fibonacci Routing (FTFR) algorithm finds on exception in which routing aborts when the number of faulty components is less than the minimum node availability, it is extremely difficult to prove it theoretically.

Furthermore, the simulation tool still has some deficiencies. The most important one is how to simulate a parallel architecture with only on CPU. Some problem can and has

been satisfactorily solved while some other problems, such as how to calculate the time for the computation of throughput in the presence of unevenly distributed workload, still leave much to be desired.

Last but not least, as the new approach of fault categorization is adopted in the discussion of our routing algorithms, it is hard to compare our strategy with ordinary ones. Besides, the comparison of reliability between different network topologies will also be difficult.

10.4 Future Work

The following are a number of areas where future work and research can be conducted for this project.

Firstly, further investigation into the feasibility of applying FNN to fault-tolerant routing can be conducted. There are two possible directions. If FNN is intrinsically inapplicable to this application, then rigorous theoretical proof, may be based on Vapnik-Chervonenkis dimension, need to be given. Otherwise, a new architecture of FNN or pure artificial neural network should be designed for this kind of high-dimension binary application. After that, the performance of fuzzy routing and traditional routing strategy can be compared on various network topologies. Whether fuzzy routing can apply to a wide variety of networks in a unified way is also worth research.

Then, it will contribute to desirable theoretical soundness if FTFR is proved to always work properly given the restriction on the number of faulty components is met. *Theorem* 4.2 and the discussion after that have presented an initial and useful result that paves way for a complete proof.

Thirdly, the architecture of the simulator needs to be improved to achieve results of higher accuracy. Multi-threaded or multi-process algorithms can be used to simulate the concurrency in the real network. Although the simulator will still depend on time slicing, the result can be possibly more accurate than the current model.

Lastly, new metrics for comparison of fault- tolerant routing strategies need to be designed and introduced, especially for *GC*, Fibonacci-class Cubes and other node/link dilution cubes. The author deems it advisable that three aspects about a faulty component should be taken into consideration:

1) Number of faulty components;
2) Type: faulty node or faulty link.
3) Location: Similar to the discussion in *GC*. We should also discriminate different types of fault distribution: evenly distributed or clustered.

REFERENCES

[1] Hsu, W. J., Chung, M. J., and Hu, Z., "Gaussian Networks For Scalable distributed Systems", *The Computer Journal*, Vol. 39, No. 5, pp. 417-426, 1996.

[2] Hsu, W. J., Chung, M. J. and Hu, Z. "A New Gaussian networks and Their Applications" *Int'l Symp. Parallel and Distributed Supercomputing*, Japan, 1995.

[3] Douglas B. West, "*Introduction to Graph Theory* - Second edition" Chapter 2 N.J.: Prentice Hall, 2001.

[4] Peter K. K. Loh, H. Schröder, W. J. Hsu, "Fault-tolerant routing on complete Josephus Cubes". *Proc. 6th Australasian Conf. Computer systems architecture*, IEEE Computer Society Press. Queensland, Australia. pp. 95-104, 2001.

[5] Wu, Jie, "Reliable Unicasting in Faulty Hypercubes Using Safety Levels", *IEEE Transactions on Computers*, Vol. 46, No. 2, pp. 241-247, February 1997.

[6] Lan, Youran, "An Adaptive Fault-Tolerant Routing Algorithm for Hypercube Multicomputers", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 11, pp. 1147-1152, November 1995.

[7] D.P.Bertsekas and J.N.Tsitsiklis, "*Parallel and Distributed Computation: Numerical Methods"*. Englewood Cliffs, NJ: Prentice-Hall, 1989, ch. 1, pp. 27–68.

[8] Y. Saad and M. H. Schultz, "Topological properties of the hypercubes," *IEEE Transactions on Computer*, vol. 37, no. 7, pp. 867-872, July 1988.

[9] L. N. Bhuyan and D. P. Agrawal, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Transactions on Computer*, vol. C-33, pp. 323-333, 1984.

[10] Ziavras, S.G., "RH: A Versatile Family of Reduced Hypercube Interconnection Networks", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 11, pp. 1210-1220, November 1994.

[11] Handel-C Language Reference Manual Version 3.1 (2002). Celoxica Limited.

[12] Hsu, W.J., "Fibonacci Cubes-A New Interconnection Topology", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 1, pp. 3-12, January 1993.

[13] Liu, J., Hsu, W.J., and Chung, M.J., "Generalized Fibonacci Cubes Are Mostly Hamiltonian", *Journal of Graph Theory*, Vol. 18, No. 8, pp. 817-829, 1994.

[14] Qian, H. and Wu, J., "Enhanced Fibonacci Cubes", *The Computer Journal*, Vol. 39, No. 4, pp. 331-345, 1996.

[15] Wu, Jie, "Extended Fibonacci Cubes", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 12, pp. 1203-1210, December 1997.

[16] Hsu, W.J. and Chung, M.J., "Generalized Fibonacci Cubes", *Proc. International Conference on Parallel Processing*, pp. 299-302, 1993.

[17] Gaber, J., Toursel, B., and Goncalves, G., "Embedding arbitrary trees in the hypercube and the *q*-dimensional mesh", *Proc. 1996 3rd International Conference on High Performance Computing*, HiPC, IEEE, Piscataway, NJ, USA, pp. 170-175, 1996.

[18] Fu, A. W. and Chau, S., "Cyclic-Cubes: A New Family of Interconnection Networks of Even Fixed-Degrees", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 12, pp. 1253-1268, December 1998.

[19] Chen, M.-S. and Shin, K.G., "Depth-First Search Approach for Fault-Tolerant Routing in Hypercube Multicomputers", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, pp. 152-159, April 1990.

[20] Wong C. V., "Maze Exploration on a PC", Nanyang Technological University, 2002.

[22] Yan Yan, "Design and Simulation of Fault-Tolerant Routing Algorithms", Nanyang Technological University, 2003.

[23] E.H. Mamdani & S. Assilian, "An experiment in linguistic synthesis with a fuzzy logic controller", *International Journal of Machine Studies*, **7**(1), 1975.

[24] T. Takagi & M. Sugeno, "Derivation of fuzzy control rules from human operator's control actions", *Proc. Of the IFAC Symp. On Fuzzy Information, Knowledge Representation and Decision Analysis*, 55-60, July 1983.

[25] T. Takagi & M. Sugeno, "Fuzzy identification of systems and its application to modeling and control", *IEEE Transactions on Systems, Man and Cybernetics*, **15**(1), 116-132, 1985.

[26] M. Sugeno & K.T. Kang, "Structure identification of fuzzy model", *Fuzzy Sets and Systems*, **28**, 1988.

[27] B. Kosko, "*Fuzzy Engineering*", Prentice Hall, 1997.

[28] I. B. Turksen and Z. Zhong, "An approximate analogical reasoning schema based on similarity measures and interval-valued fuzzy sets," *Fuzzy Sets Syst.*, vol. 34, pp. 323–346, 1990.

[29] R. L. Mantaras, *Approximate Reasoning Models*. Chichester, West Essex, U.K.: Ellis Horwood Limited, 1990.

[30] L. A. Zadeh, "Calculus of fuzzy restrictions," in *Fuzzy Sets and Their Applications to Cognitive and Decision Processes*. New York: Academic, 1975, pp. 1–39.

[31] W. L. Tung & C. Quek, "GenSoFNN: a generic self-organizing fuzzy neural network", *IEEE Trans. on Neural Networks*, **3**(5), 1075-1086, 2002.

[32] K.K. Ang, C. Quek, M. Pasquier, "POPFNN-CRI(S): pseudo outer product based fuzzy neural network using the compositional rule of inference and singleton fuzzifier", to appear in *IEEE Trans. On Systems, Man, and Cybernetics (B)*, 2002.

[33] Peter, K K, Loh and W. J. Hsu, "Performance Analysis of Fault-tolerant Interval Routing", *Proc. ISCA 11th International Conference on Parallel and Distributed Computing Systems*, Chicago, Illinois, USA, pp. 274-281, Sept. 1998

[34] Peter, K K, Loh and W. J. Hsu, "A Grouped Adaptive Packet-Switched Communications Model", *Proc. IEEE Asia Pacific Conference on Communication / Singapore International Conference on Communication Systems*, Singapore, pp. 357-361. Nov. 1998

[35] *NCUBE 6400 Processor Manual*. NCUBE Company, 1990.

[36] J. Rattler, "Concurrent Processing: A New Direction in Scientific Computing," *Proc. AFIPS Conf.*, vol. 54, pp. 157-166, 1985.

[37] Wilkinson B. and Allen M., "*Parallel Programming: Techniques and Applications Using networked Workstations and Parallel Computers*," N.J.: Prentice Hall, Inc. 1999

[38] Ziavras, S.G., "RH: A Versatile Family of Reduced Hypercube Interconnection Networks", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 11, pp. 1210-1220, November 1994.

[39] Wu, J. and Huang, K., "The Balanced Hypercube: A Cube-Based System for Fault-Tolerant Applications", *IEEE Transactions on Computers*, Vol. 46, No. 4, pp. 484-490, April 1997.

[40] Mellor-Crummey, J.M., "Experiences with the BBN Butterfly", *Digest of Papers – Compcon Spring 88: Intellectual Leverage, 33rd IEEE Computer Society InternationalConference*, San Francisco, CA, USA, pp. 101-104, 1988.

[41] Preparata, F. and Vuillemin, J., "The cube-connected cycles: a versatile network for parallel computation", *Communications of the ACM*, Vol. 24, No. 5, pp. 300-309, May 1981.

[42] Stone, H., "Parallel processing with the perfect shuffle", *IEEE Transactions on Computers*, Vol. C-20, No. 2, pp. 153-161, February 1971.

[43] Samatham, M.R. and Pradhan, D.K., "De Bruijn multiprocessor network: a versatile parallel processing and sorting network for VLSI", *IEEE Transactions on Computers*, Vol. 37, No. 7, pp. 567-581, July 1988.

[44] P.T. Gaughan and S. Yalamanchili, "Adaptive Routing Protocols for Hypercube Interconnection Networks," *Computer*, vol. 26, no. 5, pp. 12-24, May 1993.

[45] L.M. Ni and P.K. McKinley, "A Survey of Routing Techniques in Wormhole Networks," *Computer*, vol. 26, no. 2, pp. 62-76, Feb. 1993.

[46] Y. Saad and M.H. Schultz, "Data Communication in Hypercubes," Technical Report YALEU/DCS/RR-428, Dept. of Computer Science, Yale Univ., June 1985.

[47] H. Sullivan, T. Bashkow, and D. Klappholz, "A Large Scale, Homogeneous,Fully Distributed Parallel Machine," *Proc. Fourth Ann. Symp. Computer Architecture*, pp. 105-124, Mar. 1977.

[48] M.S. Chen and K.G. Shin, "Adaptive Fault-Tolerant Routing in Hypercube Multicomputers," *IEEE Trans. Computers*, vol. 39, no. 12, pp. 1,406-1,416, Dec. 1990.

[49] J.M. Gordon and Q.F. Stout, "Hypercube Message Routing in the Presence of Faults," *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, pp. 251-263, Jan. 1988.

[50] T.C. Lee and J.P. Hayes, "A Fault-Tolerant Communication Scheme for Hypercube Computers," *IEEE Trans. Computers*, vol. 41, no. 10, pp. 1,242-1,256, Oct. 1992.

[51] C.S. Raghavendra, P.J. Yang, and S.B. Tien, "Free Dimension–An Effective Approach to Achieving Fault Tolerance in Hypbercubes," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing*, pp. 170-177, 1992.

[52] El-Amawy, A. and Latifi, S., "Properties and Performance of Folded Hypercubes", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 1, pp. 31-42, January 1991.

[53] Tzeng, N.-F. and Wei, S., "Enhanced Hypercubes," *IEEE Transactions on Computers*, Vol. 40, No. 3, pp. 284-294, March 1991.

[54] J. Wu, "Adaptive Fault-Tolerant Routing in Cube-Based Multicomputers Using Safety Vectors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 4, pp. 321-334, April 1996.

[55] Peter, K K, Loh and W. J. Hsu, "Fault-tolerant Communications on Hypercube-clusters," Journal of Interconnection Networks, Vol. 1, No. 4, pp. 315-329, December 2000.

[56] Schwabe, E.J., "On the computational equivalence of hypercube-derived networks", *SPAA '90 – Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 388-397, 1990.

[57] Silberschatz, A., Peterson, J. L., Galvin, P. B., "*Operating System Concepts, 3rd ed.*", Addison-Wesley, 1991.

[58] M. D. Grammatikakis, Frank D. Hsu and M. Hraetzl, "*Parallel System Interconnection and Communications*", USA: CRC Press, 2001

[59] W. J. Dally and C. Seitz, "Deadlock Free Message Routing In Multiprocessor Interconnection Networks," IEEE Transaction on Computers, vol. C-36, no. 5, pp. 547-553, 1987.

[60] L. A. Zadeh, "Fuzzy sets", *Information and Control*, vol. **8**, pp. 338-353, 1965.

[61] L. A. Zadeh, "Outline of a new approach to the analysis of complex systems and decision processes", *IEEE Trans. On Systems, Man, and Cybernetics*, vol. 3, no. 1, pp. 28-44, Jan. 1973.

[62] L. A. Zadeh, "Fuzzy logic", *Computer*, vol. 1, no. 4, pp. 83-93, 1988.

[63] L. A. Zadeh, "A Theory of Approximate Reasoning," *Fuzzy Sets and Applications: Selected Papers by L. A. Zadeh*, John Wiley & Sons, 1979.

[64] Peter, K K, Loh and V. J. Hsu, "The Josephus Cube: A Novel Interconnection Network," Journal of Parallel Computing, vol. 26, pp. 427-453, Sept. 1999.

[65] HANDEL-C Language Overview, Celoxica Ltd., 2002.

[66] Handel-C Language Reference Manual Version 3.1, Celoxica Ltd., 2002.

[67] RC100 Hardware Manual, Celoxica Ltd., 2001.

[68] RC100 Installation Guide, Celoxica Ltd., 2001.

[69] RC100 Function Library Manual, Celoxica Ltd., 2001.

[70] RC100 Tutorial Manual, Celoxica Ltd., 2001.

[71] DK1 Design Suite User Manual Version 3.1, Celoxica Ltd., 2002.

[72] DK1 Waveform Analyzer Manual Version 1.0, Celoxica Ltd., 2001.

[73] Handel-C Floating-point Library Manual, Version 1.1, Celoxica Ltd., 2001.

[74] Tang Kong Choy, "Router Design for Regular Networks", Nanyang Technological University, 2002.

Appendix I Proof of Case III for *Theorem* 4.2

The case III for *Theorem* 4.2 is:

In a fault-free *Enhanced Fibonacci Cube*, there is always a preferred dimension available at packet's present node before the destination is reached.

Proof:

For convenience, the definition of *Enhanced Fibonacci Cube* is copied here.

Let $EFC_n = < V_n, E_n >$ denote the *Enhanced Fibonacci Cube of order n*, then $V_n = 00 \| V_{n-2} \cup 10 \| V_{n-2} \cup 0100 \| V_{n-4} \| \cup 0101 \| V_{n-4}$. Two nodes in EFC_n are connected by an edge in E_n if and only if their labels differ in exactly one bit position. As initial conditions for recursion, $V_3 = \{1,0\}$, $V_4 = \{01,\ 00,\ 10\}$,

$V_5 = \{001,\ 101,\ 100,\ 000,\ 010\}$ and

$V_6 = \{0001,\ 0101,\ 0100,\ 0000,\ 0010,\ 1010,\ 1000,\ 1001\}$.

For *Enhanced Fibonacci Cubes of* low dimension, it is easy to prove the theorem by enumeration. So now, we assume that dimension n is larger than 6. According to the definition above, the leftmost four bits of any valid *Enhanced Fibonacci Cube* with dimension over 6 can only be: 0000, 0001, 0010, 0100, 0101, 1000, 1001, 1010. Suppose the address of current node is $a_{n-1}a_{n-2}\cdots a_1a_0$ while the address of the destination node is $b_{n-1}b_{n-2}\cdots b_1b_0$. We prove the theorem by induction. Assume that the theorem hold for dimensions less than *n*.

1) If $a_{n-1}a_{n-2}a_{n-3}a_{n-4} = 0000$, then:

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4} = 0000$, then either destination is reached or apply the induction assumption for dimension n - 4.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4} = 0001$, then as $a_{n-3}a_{n-4}\cdots a_1a_0$ and $b_{n-3}b_{n-4}\cdots b_1b_0$ are valid (n-2)-dimension *EFC* addresses and they are different, we can apply the induction assumption for dimension n-2.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4} = 0010$, then dimension n-3 is an available preferred dimension.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4} = 0100$ or 0101, then dimension n-2 is an available preferred dimension.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4} = 1000$, 1001 or 1010, then dimension n-1 is an available preferred dimension.

2) If $a_{n-1}a_{n-2}a_{n-3}a_{n-4} = 0001$, then:

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4} = 0000$, 0010 or 0100, then as $a_{n-3}a_{n-4}\cdots a_1a_0$ and $b_{n-3}b_{n-4}\cdots b_1b_0$ are valid (n-2)-dimension *EFC* addresses and they are different, we can apply the induction assumption for dimension n-2.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4} = 0001$, then either destination is reached or apply the induction assumption for dimension n - 4.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4} = 1000$, 1001 or 1010, then dimension n-1 is an available preferred dimension.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4} = 0101$, then the analysis goes the following way:

As $a_{n-1}a_{n-2}a_{n-3}a_{n-4} = 0001$, thus $a_{n-1}a_{n-2}a_{n-3}a_{n-4}a_{n-5} = 00010$. If $a_{n-6} = 0$, then inverting a_{n-2} to 1 will produce a new valid address and n-2 will be an available preferred

dimension. Otherwise, $a_{n-6} = 1$, $a_{n-1}a_{n-2}a_{n-3}a_{n-4}a_{n-5}a_{n-6} = 000101$. So b_{n-6} must be 1, otherwise dimension n-6 will be an available preferred dimension. Then, b_{n-5} must in turn be 0 and b_{n-7} must be 0, according to the definition of *EFC*. $b_{n-1}b_{n-2}b_{n-3}b_{n-4}b_{n-5}b_{n-6} = 010101$. The deduction flow is illustrated in the following Figure AI.1. The (1) represents that it is deduced by avoiding making dimension *n*-2 an available preferred dimension.

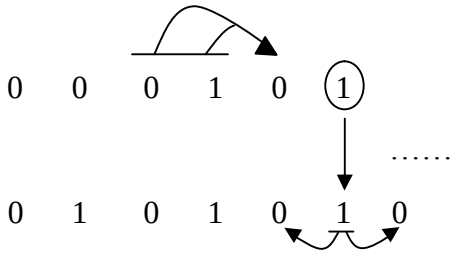


Figure AI.1 Deduction flow for step 1

Then, a_{n-7} must be 0, otherwise *n*-7 will be an available preferred dimension. If $a_{n-8} = 0$, then *n*-2 will be an available preferred dimension. So assume $a_{n-8} = 1$, $a_{n-9} = 0$. If $a_{n-10} = 0$, then *n*-2 will be an available preferred dimension. So assume $a_{n-10} = 1$. If $b_{n-10} = 0$, then *n*-10 will be an available preferred dimension. So assume $b_{n-10} = 1$. Thus, $b_{n-9} = 0$. The deduction flow is illustrated in Fig. AI.2.

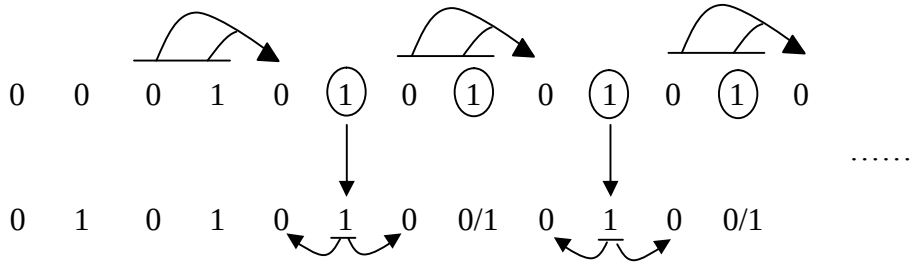


Figure AI.2 Deduction flow for step 2

So for a, 0 and 1 appear alternately until the least significant four digits are met. With careful analysis of the initial condition, it is easy to see that in such a worst case studied above, n-2 will finally turn out to be an available preferred dimension.

3) If $a_{n-1}a_{n-2}a_{n-3}a_{n-4} = 0010$, then:

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0000, 0100, 0101, 1000 or 1001, then dimension n-3 is an available preferred dimension.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0001, then as $a_{n-3}a_{n-4}\cdots a_1a_0$ and $b_{n-3}b_{n-4}\cdots b_1b_0$ are valid (n-2)-dimension *EFC* addresses and they are different, we can apply the induction assumption for dimension n-2.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0010, then either destination is reached or apply the induction assumption for dimension n - 4.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 1010, then dimension n-1 is an available preferred dimension.

4) If $a_{n-1}a_{n-2}a_{n-3}a_{n-4}$ = 0100, then:

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0000, 0001, 0010, 1000, 1001 or 1010 then dimension n-2 is an available preferred dimension.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0100, either destination is reached or apply the induction assumption for dimension n - 4.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0101, then dimension n-4 is an available preferred dimension.

5) If $a_{n-1}a_{n-2}a_{n-3}a_{n-4}$ = 0101, then:

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0000, 0010, 0100, 1000 or 1010, then dimension n-4 is an available preferred dimension.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0101, then either destination is reached or apply the induction assumption for dimension n - 4.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 1001 or 0001, then the proof is similar to the proof for $a_{n-1}a_{n-2}a_{n-3}a_{n-4}$ = 0001 and $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0101. Here, we only show the deduction flow in Figure AI.3.

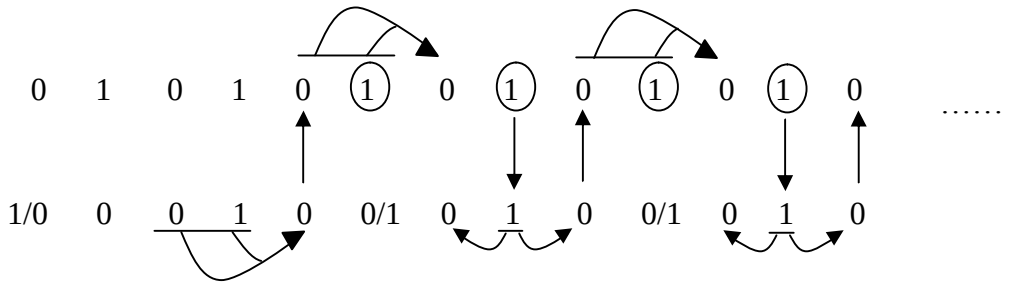


Figure AI.3 Deduction flow for case 5

6) If $a_{n-1}a_{n-2}a_{n-3}a_{n-4}$ = 1000, then:

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0000, 0001, 0010, 0100 or 0101, then dimension n-1 is an available preferred dimension.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 1000, then either destination is reached or apply the induction assumption for dimension n - 4.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 1001, then as $a_{n-3}a_{n-4}\cdots a_1a_0$ and $b_{n-3}b_{n-4}\cdots b_1b_0$ are valid (n-2)-dimension *EFC* addresses and they are different, we can apply the induction assumption for dimension n-2.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 1010, then dimension n-3 is an available preferred dimension.

7) If $a_{n-1}a_{n-2}a_{n-3}a_{n-4}$ = 1001, then:

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0000, 0001, 0010, 0100 or 0101, then dimension n-1 is an available preferred dimension.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 1000 or 1010, then as $a_{n-3}a_{n-4}\cdots a_1a_0$ and $b_{n-3}b_{n-4}\cdots b_1b_0$ are valid (n-2)-dimension *EFC* addresses and they are different, we can apply the induction assumption for dimension n-2.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 1001, then either destination is reached or apply the induction assumption for dimension n - 4.

8) If $a_{n-1}a_{n-2}a_{n-3}a_{n-4}$ = 1010, then:

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 0000, 0001, 0010, 0100 or 0101, then dimension n-1 is an available preferred dimension.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 1000 or 1001, then dimension n-3 is an available preferred dimension.

If $b_{n-1}b_{n-2}b_{n-3}b_{n-4}$ = 1010, then either destination is reached or apply the induction assumption for dimension n - 4.

With all the situations considered carefully, we have completely proved the case III of *Theorem* 4.2, and thus *Theorem* 4.2.

Appendix II

Implementation Code for algorithm 6.1:

```
int getPath(unsigned from, unsigned to)                 // Assume from != to
{
        int top, bottom, current;       // current stack is from 0 to top-1, current available
                                        //  record index is last;
        unsigned x1, x2, mask, diff, mid1, mid2;

        result[0].from=from;
        result[0].to = to;
        result[0].top1 = n;             // dimension is from 1 to n
        result[0].index = 0;

        top = 0;
        bottom = last;

        while (top >= 0)
        {
                x1 = result[top].from;
                x2 = result[top].to;
                current = result[top].top1;
                mask = 1 << (result[top].top1 - 1);
                diff = x1 ^ x2;

                while(1)                // it is guaranteed that no item in result array
                                                // has same from and to
                {
                        if ( mask & diff )
                                break;
                        mask >>= 1;
                        current --;
                }

                // x1 and x2 are different in dimension 'current' (1 to n)
                if (current == 1)
                {
                        result[bottom].from = x1;
                        result[bottom].to = x2;
```

```
                result[bottom].index= result[top].index + 1;
                bottom --;
                top --;
                continue;
            }
            else
            {
                top--;
                mask = (1<<(current - 1)) - 1;
                mid1 = x1 & (~mask);
                mid1 |= (current - 1);
                mid2 = x2 & (~mask);
                mid2 |= (current - 1);


                result[bottom].from = mid1;
                result[bottom].to = mid2;
                result[bottom].index= result[top+1].index + (1<<current);
                bottom --;

                if(mid1 != x1)
                {
                    top ++;
                    result[top].to = mid1;
                    result[top].top1 = current - 1;
                }

                if(mid2 != x2)
                {
                    top ++;
                    result[top].from = mid2;
                    result[top].to = x2;
                    result[top].top1 = current - 1;
                    result[top].index = result[bottom+1].index;
                }
            }
        }

        Sort(bottom + 1, last)
        return bottom + 1;
}
```

Appendix III Program that calculates the diameter of T_a

```
class entry
{
public:
	unsigned content;
	entry *previous;
};

class Stack
{
public:
	entry *current;
	Stack()
	{
		current=NULL;
	}

	void Push(unsigned i)
	{
		if(!current)
		{
			current = new entry;
			current->previous=NULL;
			current->content=i;
		}
		else
		{
			entry *temp;
			temp = new entry;
			temp->previous=current;
			temp->content=i;
			current=temp;
		}
	}

	unsigned Pop()
	{
		if(!current)
			return INFINITY;
```

```
            unsigned result;
            entry *pre=current->previous;
            result=current->content;
            delete current;
            current = pre;
            return result;
        }

        bool Empty()
        {
            if(current)
                return false;
            else
                return true;
        }
};


class node
{
public:
        bool visited;
        int all;
        int current;
        unsigned  *neighbors;

        node()
        {
            all=2;
            current=0;
            neighbors = NULL;
        }

        ~node()
        {
            delete []neighbors;
        }

        void Construct(unsigned p, int n)       // there are n bits
        {
            unsigned record[30];
```

```
        unsigned mask;

        all = 1;
        record[0]=0;
        mask = 1 << (n-1);
        mask --;
        for (unsigned i = n-1; i>0; i--)
        {
                if( (p & mask) == i)
                        record[all++] = i;
                mask >>= 1;
        }

        // now we get all the dimensions at which a link exists
        neighbors = new unsigned [all];
        current = 0;
        while (current < all)
        {
                mask = record [current];
                mask = 1 << mask;
                neighbors[current++] = (p ^ mask);
        }
        current = -1;
        visited = false;
        if(all==2)
                d2++;
        if(all==1)
                d1++;
}

unsigned getNext()              // get the next unvisited neighbor
{
        current++;
        while (current<all)
        {
                if( nodes[neighbors[current]].visited )
                {
                        neighbors[current]=0; // will not be chosen
                        current++;
                        continue;
                }
```

```
            return neighbors[current];
        }
        return -1;
 }

unsigned longestPath()          // return the longest path down.  By the way,
                                //  compare the max route with record
{
        unsigned temp1, temp2;
        int dimension1=0, dimension2=0, i;

        if(all==1)              // leaf, only one link (to father)
            return 0;

        temp1 = temp2 = 0;
        for (i=0; i<all; i++)
        {
            if(neighbors[i]>temp1)
            {
                temp1 = neighbors[i];
                dimension1 = i;
            }
        }
        if(all==2)
        {
            if(temp1 > max)
                max =temp1;
            return temp1;
        }

        for( i=0; i<all; i++)
        {
            if(i==dimension1)
                continue;
            if(neighbors[i]>temp2)
            {
                temp2 = neighbors[i];
                dimension2 = i;
            }
        }
```

```
            if( temp1 + temp2 > max )
                max = temp1 + temp2;

            return temp1;
        }

        void sonDepth()         // calculate the max of current son's longest path down
        {
            unsigned result = 0, son;
            if(all==1)
                return;
            son = neighbors[current];
            result = nodes[son].longestPath();
            neighbors[current] = result + 1;
        }
};

void main(void)
{
        nodes = NULL;


        for (n = 4; n < 27 ; n++)
        {
            N = 1 << n;
            d2=0;
        d1=0;
            if(!nodes)
                delete []nodes;
            nodes = new node[N];
            max = 0;

            for(unsigned i = 0; i < (unsigned) N; i++)
                nodes[i].Construct(i,n);

            // now we calculate the distance
            Stack stack;
            unsigned p = 0, q;

            while( !stack.Empty() || p != INFINITY)
```

```
		{
			if( p != INFINITY)
			{
				nodes[p].visited=true;
				stack.Push(p);
				p=nodes[p].getNext();
			}
			else		// backtrack
			{
				p = stack.Pop();
				nodes[p].sonDepth();
				q = nodes[p].getNext();
				if( q != INFINITY)
				{
					stack.Push(p);
					p = q;
				}
				else
					p = INFINITY;
			}
		}
		cout<<"\n The longest distance in the graph with n="<<n<<" N="<<N
			<<"  is: "<<max<<endl;
		cout<<"The percentage of 2 degree nodes is: "<<d2*100.0/N<<"%"<<endl;
		cout<<"The percentage of 1 degree nodes is: "<<d1*100.0/N<<"%"<<endl;
	}
}
```

Appendix IV Conversion functions for Extended Fibonacci Cube

```
unsigned CExtFibCube :: Dec2Fib
        (unsigned x, unsigned digit)
{
    unsigned result;
    result = 0;
    digit = Num_Bits;

    while(digit > k+1)
    {
        if( x >= FibNum[digit+1])
        {
            result |= (1 << (digit - 1));
            x -= FibNum[digit + 1];
            digit -= 2;
        }
        else
            digit --;
    }
    result |= x;

    return result;
}


unsigned CExtFibCube::Fib2Dec(unsigned
            x, unsigned digit)
{
    unsigned result, mask;
    digit = Num_Bits;  // how many
                       // digits are left

   resultlt = 0;
   mask = (1 << (Num_Bits - 1));
   while(digit > k+1)
   {

   if( mask & x )   // test the most
                    // significant bit
       {
                                // it is 1
           result += FibNum[digit+1];
           digit -= 2;
           mask >>= 2;
       }
       else                     //it is 0
       {
           digit --;
           mask >>= 1;
       }
   }

   if(digit == k+1)
       result += (x & ((1<<(k+1))-1));
   else
   {
       ASSERT(digit == k);
       result += (x & ((1<<k)-1));
   }

    return result;
}
```

Appendix V CTimer Implementation

```
#define PENTIUMSPEED        2457.6
#define MHZ                 1000000.0

class CTimer
{
public:

    // ULONGLONG is 64-bit unsigned
    ULONGLONG start;
    ULONGLONG duration;

     CTimer()
     {
         duration = (ULONGLONG) 0;
     }

     void Reset()
     {
         duration = (ULONGLONG) 0;
     }

     void  P()
     {
          unsigned temptime, temptime2;

          asm{
              _emit 0x0f;
                emit 0x31;          //rdtsc
                mov temptime, eax;
                mov temptime2, edx
           }

           start = temptime2;
           start <<= 32;
           start += temptime;
     }

     void V()
     {
         unsigned temptime, temptime2;
         ULONGLONG temp;

         __asm{
                 _emit 0x0f;
                   emit 0x31;        //rdtsc
                    mov temptime, eax;
                    mov temptime2, edx
            }

                 temp = temptime2;
                 temp <<= 32;
                 temp += temptime;


                 duration += (temp - start);
       }

    double getDuration()  // the unit
                          is micro-second
    {
         double temp, result;

         // there is no direct conversion
         from ULONGLONG to double
         available, so we have to convert
         ULONGLONG to unsigned first

         temp = (double) ((unsigned)
                   (duration >> 32));
          temp *= 4294967296;

         result = temp + (double)((unsigned)
       (duration & 0x00000000ffffffff));

          result = result * 1000000.0 /
       ((PENTIUMSPEED) * (MHZ));
         return result;
       }
};
```

Note:

RDTSC (**R**ea**D** **T**ime **S**tamp **C**ounter) is a set of assembly directives. The _emit directives are inline assembly code for directly insert/declare a byte into the current text location.
The assembly directive RDTSC returns the number of clock cycles since the CPU was powered up or reset. The number of clock cycles is measured by a 64-bit counter and is stored in processor register EDX:EAX, where EDX contains the higher 32-bit value and EAX the lower 32-bit.

The experiment is carried out on a 2.4GHz CPU, so PENTIUMSPEED is set to $1024 \times 2.4 = 2457.6$. Since the 64-bit counter can represent more than 82850 days, it is free from overflow. When running on other computers, this parameter may need to be modified correspondingly.

APPENDIX VI Raw Data of Simulation Result

MALatency: Mean Average Latency

AL SD: Average Latency Standard Derivation

Mthroughput: Mean Throughput

Throughput SD: Mean Throughput Standard Derivation

EN: Erroneous nodes, faulty nodes

EL: Erroneous links, faulty links

For regular Fibonacci Cube, with no fault. Simulation duration is 60 seconds.

Dimension	MALatency	AL SD	MThroughput	Throughput SD
5	5.731	0.212	2083902.953	39478.423
6	7.807	0.279	2509919.394	54907.034
7	9.992	0.305	3230724.711	40587.542
8	13.08	0.296	4200724.86	52155.955
9	16.095	0.521	5590315.3	50274.17
10	16.455	0.35	6649473.716	104408.736
11	20.155	0.406	5658421.504	139069.747
12	32.974	0.449	3191762.076	135310.57
13	54.566	0.979	2938218.173	100489.502
14	87.058	1.752	3701941.373	47461.994
15	114.863	2.767	7103450.59	74553.727
16	151.413	3.259	13099109.48	107931.935
17	190.072	3.628	23680371.47	404451.255
18	232.201	3.315	43698701.97	180854.325
19	275.059	4.76	75878409.95	711924.2
20	303.067	5.738	112260412.5	816434.957
21	289.345	1.61	162391187.7	2288984.601
22	242.265	2.305	243288065.4	10032688
23	189.133	1.747	350782433.1	18131088.72

For binary hypercube, with no fault. Simulation duration is 60 seconds

Dimension	MALatency	AL SD	MThroughput	Throughput SD
5	6.996	0.208	3046390.319	127271.508
6	10.093	0.383	4667176.013	95245.157
7	13.31	0.416	6313704.597	131947.345
8	15.935	1.056	6279516.186	166424.374
9	27.196	0.252	7334754.301	74366.35
10	44.961	0.776	13764698.95	190518.435
11	70.103	0.567	28554984.68	345918.469
12	106.431	1.369	58404606.18	527898.822
13	148.386	1.39	111371365.7	455649.988
14	170.444	2.784	193501166.9	1435244.624
15	157.378	0.81	346341553.8	9767170.654

For Enhanced Fibonacci Cube, with no fault. Simulation duration is 60 seconds.

Dimension	MALatency	AL SD	Mthroughput	Throughput SD
5	6.118	0.334	1867121.121	32300.797
6	7.754	0.153	2282489.291	104697.344
7	10.56	0.249	3092194.369	45348.138
8	14.422	0.634	3649553.66	46371.699
9	19.084	0.423	4574387.498	40678.36
10	19.623	0.302	4071091.475	58355.813
11	26.128	0.493	4620221.082	132135.598
12	44.148	1.543	3512369.454	80139.695
13	78.206	1.528	4530388.221	188164.791
14	113.292	1.345	6578548.289	109678.648
15	155.442	3.533	14938441.68	208868.714
16	202.393	4.199	23031271.55	276336.661
17	274.073	2.346	48805511.77	270534.491
18	347.074	7.622	74274771.66	172534.607
19	390.71	7.771	118787792.3	1045183.748
20	383.481	5.161	156077315.7	1477467.015
21	312.276	2.172	251850034.3	12029546.57
22	246.282	2.134	344473927.4	20065672.56

For Extended Fibonacci Cube XFC_1, with no fault. Simulation duration is 60 seconds.

Dimension	MALatency	AL SD	Mthroughput	Throughput SD
5	6.111	0.548	1997666.045	129741.239
6	8.398	0.441	2544305.343	68773.57
7	10.506	0.525	3022916.698	40881.233
8	13.589	0.435	3763136.654	61615.341
9	16.449	0.483	4091118.929	102616.141
10	19.643	0.232	4164382.708	25607.429
11	25.031	0.312	3275873.753	43656.724
12	41.716	0.815	2714505.551	73940.93
13	70.684	1.9	3192470.325	87453.553
14	102.635	2.163	4735510.016	68645.701
15	132.692	1.857	8988125.719	187721.677
16	165.983	3.745	16699058.48	123745.832
17	213.803	3.865	31539444.04	127224.894
18	261.356	5.957	57534241.64	425847.624
19	293.774	2.818	93071257.8	338624.828
20	299.232	1.542	133804007.5	822463.41
21	263.198	2.144	200622752.2	7332553.462

For binary hypercubes with faulty nodes only. Simulation duration is 60 seconds.

Dimension	EN	MALatency	AL SD	Mthroughput	Throughput SD
14	0	170.444	2.784	193501166.9	1435244.624
14	1	170.908	2.317	193452666.5	1099476.623
14	2	170.767	2.733	192553922.8	1775080.445
14	3	171.05	1.752	193229950.5	1196041.28
14	4	171.331	2.636	193922825.4	1274647.738
14	5	171.15	2.523	192788940.4	1433044.25
14	6	171.64	3.356	192156973.4	2101961.334
14	7	171.309	2.507	192740750.9	718702.298
14	8	171.457	2.965	192409636.3	2235970.311
14	9	170.89	3.169	193015906.2	2094651.531
14	10	171.153	2.205	192640297.3	903856.609
14	11	171.021	3.55	192151673.6	1964048.968
14	12	171.908	2.281	192509583.2	1060578.947
14	13	172.762	2.749	192678761.5	1179273.101

For regular Fibonacci Cube with faulty nodes only. Simulation duration is 60 seconds.

Dimension	EN	MALatency	AL SD	Mthroughput	Throughput SD
20	0	303.067	5.738	112260412.5	816434.957
20	1	302.155	3.766	111999010.1	541699.452
20	2	303.922	2.429	112390575.2	392661.017
20	3	300.991	4.642	112372574.4	1336658.26
20	4	301.532	5.531	112122284	530947.712
20	5	302.823	4.883	112363961.4	581985.825
20	6	303.247	5.94	111776616.1	1101095.928

For Enhanced Fibonacci Cube with faulty nodes only. Simulation duration is 60 seconds.

Dimension	EN	Average Latency	MALatency	Mthroughput	Throughput SD
19	0	389.103	7.771	118787792.3	1045183.748
19	1	389.226	4.438	119188267.7	863318.596
19	2	389.36	6.259	118588758.3	345219.298
19	3	389.487	6.328	119074739.5	819653.202
19	4	389.874	8.033	119112338.6	1245026.195
19	5	390.34	9.299	118558303.6	672637.849

For Extended Fibonacci Cube XFC_1 with faulty nodes only.
Simulation duration is 60 seconds.

Dimension	EN	MALatency	AL SD	Mthroughput	Throughput SD
18	0	261.356	5.957	57534241.64	425847.624
18	1	258.198	2.785	57444466.25	333138.835
18	2	259.587	2.886	57502706.45	305976.943
18	3	259.561	2.207	57398983.78	249533.696
18	4	257.964	5.649	57688070.44	1426735.889
18	5	260.374	4.093	57366672.28	220644.97
18	6	257.031	6.869	57740320.65	559855.318

Collective data for regular Fibonacci Cube. Simulation duration is 60 seconds.

Dimension	EN	EL	MALatency	AL SD	Mthroughput	Throughput SD
5	0	0	5.731	0.212	2083902.953	39478.423
6	0	0	7.807	0.279	2509919.394	54907.034
7	0	0	9.992	0.305	3230724.711	40587.542
8	0	0	13.08	0.296	4200724.86	52155.955
9	0	0	16.095	0.521	5590315.3	50274.17
10	0	0	16.455	0.35	6649473.716	104408.736
11	0	0	20.155	0.406	5658421.504	139069.747
12	0	0	32.974	0.449	3191762.076	135310.57
13	0	0	54.566	0.979	2938218.173	100489.502
14	0	0	87.058	1.752	3701941.373	47461.994
15	0	0	114.863	2.767	7103450.59	74553.727
16	0	0	151.413	3.259	13099109.48	107931.935
17	0	0	190.072	3.628	23680371.47	404451.255
18	0	0	232.201	3.315	43698701.97	180854.325
19	0	0	275.059	4.76	75878409.95	711924.2
20	0	0	303.067	5.738	112260412.5	816434.957
21	0	0	289.345	1.61	162391187.7	2288984.601
22	0	0	242.265	2.305	243288065.4	10032688
23	0	0	189.133	1.747	350782433.1	18131088.72
5	1	0	5.841	0.386	1618549.516	72194.816
6	1	0	7.351	0.39	2126026.976	113579.376
7	1	0	10.145	0.353	2761105.574	90267.137
8	1	0	13.292	0.504	3372483.103	64298.75
9	1	0	16.739	0.993	4304299.715	195288.141
10	1	0	16.5	1.097	4861723.763	143332.959
11	1	0	21.529	0.853	4273978.892	86374.399
12	1	0	33.45	0.728	2970892.611	118091.51
13	1	0	55.81	2.192	2822988.235	81006.148
14	1	0	87.192	1.149	3750585.78	92523.637

15	1	0	112.604	1.456	7171639.391	89578.961
16	1	0	146.284	1.068	13068279.02	129118.695
17	1	0	193.371	5.441	23525662.46	230885.207
18	1	0	228.362	3.619	43880389.04	236422.429
19	1	0	276.543	9.206	75734875.52	333895.796
20	1	0	302.155	3.766	111999010.1	541699.452
21	1	0	287.889	3.68	161551742.5	2901851.687
22	1	0	242.436	1.367	241733277.3	9015897.691
23	1	0	189.254	1.425	352411767.5	19880373.02
7	2	0	9.834	0.457	2673296.987	121027.335
8	2	0	12.962	0.365	3281306.28	87370.985
9	2	0	16.481	0.726	4248450.497	175355.554
10	2	0	17.117	0.447	4318529.501	112847.492
11	2	0	22.314	0.807	3802018.538	67406.071
12	2	0	34.379	1.836	2899333.034	279764.286
13	2	0	55.788	1.207	2827076.315	137049.032
14	2	0	86.611	1.384	3661348.803	111057.116
15	2	0	111.645	1.548	7167980.838	303127.802
16	2	0	147.621	1.313	12922902.61	227256.572
17	2	0	189.557	6.229	23715267.72	326830.328
18	2	0	232.7	4.317	43537637.1	116949.548
19	2	0	278.776	4.242	76261005.38	549153.146
20	2	0	303.922	2.429	112390575.2	392661.017
21	2	0	287.411	2.868	162237785.1	2675653.88
22	2	0	244.045	1.89	242105643.7	8285151.7
23	2	0	189.183	1.294	351914363.9	18288048.95
10	3	0	17.602	1.981	4053731.24	381424.032
11	3	0	21.675	0.657	3501868.693	432017.043
12	3	0	33.061	0.945	3072415.772	54872.898
13	3	0	55.599	1.47	2847679.812	84563.958
14	3	0	87.865	1.621	3792607.37	127188.396
15	3	0	117.009	2.888	7155727.834	115170.834
16	3	0	151.481	3.376	12895748.08	184784.944
17	3	0	193.618	4.854	23872469.42	225886.935
18	3	0	234.779	2.631	43623510.47	628782.563
19	3	0	276.977	6.405	75651568.99	1182848.891
20	3	0	300.991	4.642	112372574.4	1336658.26
21	3	0	289.744	1.73	160943842.6	2597467.327
22	3	0	243.417	1.349	242547865.8	9674494.215
23	3	0	188.369	1.149	351941789.4	19237030.7
13	4	0	55.289	1.597	2807826.854	68066.228
14	4	0	86.307	2.557	3697107.437	169624.455
15	4	0	115.346	1.603	7157687.081	275615.391
16	4	0	148.617	2.178	13037838.15	261051.447
17	4	0	191.644	3.571	23605535.18	125514.916
18	4	0	234.495	5.669	43405108.42	208137.297
19	4	0	277.937	3.948	75911122.39	617764.067
20	4	0	301.532	5.531	112122284	530947.712

21	4	0	287.936	4.619	160748440.7	2047598.54
22	4	0	242.596	1.688	242899126.6	9647192.063
23	4	0	188.031	0.893	352360852.8	19045274.37
16	5	0	151.647	3.61	12895138.08	129809.399
17	5	0	189.658	3.958	23711335.99	294883.227
18	5	0	234.082	2.601	43850272.85	441577.508
19	5	0	278.013	6.432	75747040.73	727315.024
20	5	0	302.823	4.883	112363961.4	581985.825
21	5	0	286.8	3.94	161293466.8	2432747.019
22	5	0	241.441	1.253	239650845.7	9331981.02
23	5	0	187.879	1.663	351191231.1	17631945.04
19	6	0	273.405	3.079	75397343.12	964119.355
20	6	0	303.247	5.94	111776616.1	1101095.928
21	6	0	288.296	1.199	161496358.9	2356425.875
22	6	0	242.883	0.266	243473027.6	10952392.96
23	6	0	188.372	1.216	352543696.4	15539182.15
22	7	0	243.434	1.387	242832192.6	9331375.855
23	7	0	188.519	0.904	351617990.2	17437587.55
5	0	1	6.097	0.462	1726640.593	98924.5
6	0	1	7.617	0.321	2182032.631	36757.316
7	0	1	9.876	0.264	2738781.763	42109.965
8	0	1	13.741	0.52	3377856.11	28762.92
9	0	1	17.135	0.446	3933021.1	100574.699
10	0	1	18.982	0.702	4176426.926	54626.762
11	0	1	22.066	0.55	3494240.258	105971.85
12	0	1	34.805	0.512	2945132.317	123370.165
13	0	1	55.586	1.306	2873227.469	122647.05
14	0	1	87.152	2.031	3697591.115	105786.394
15	0	1	113.826	2.418	7085188.141	85301.345
16	0	1	145.261	3.331	12794124.43	167862.113
17	0	1	186.044	5.554	23519854.81	287804.907
18	0	1	232.573	2.801	43135061.68	968377.075
19	0	1	278.354	5.378	76279807.61	646134.003
20	0	1	304.017	6.072	112479316.7	1401271.608
21	0	1	287.328	1.625	161746953.7	2281468.207
22	0	1	244.863	2.146	242518784.7	8230428.261
23	0	1	188.295	1.254	351867941.6	18445606.29
7	0	2	9.779	0.33	2802521.589	119372.526
8	0	2	12.656	0.593	3518964.724	35118.731
9	0	2	16.127	1.415	4349712.159	137129.389
10	0	2	17.628	0.756	4176483.821	91787.965
11	0	2	22.46	0.6	3581334.523	89022.239
12	0	2	34.24	0.723	2961031.809	97220.94
13	0	2	55.877	1.135	2864100.956	86629.739
14	0	2	89.591	1.079	3692789.936	85996.717
15	0	2	112.918	1.949	7219054.51	159439.033
16	0	2	150.229	2.882	13095535.83	83780.784
17	0	2	192.874	3.601	23625762.59	392668.198

18	0	2	233.495	4.189	43668385.69	175002.276
19	0	2	287.79	5.453	76274428.92	593318.292
20	0	2	304.549	4.808	112627107.9	585352.334
21	0	2	289.082	2.329	161944709.8	3531325.42
22	0	2	242.9	2.657	241599965.2	9807168.625
23	0	2	188.672	0.717	350587767	18556023.75
10	0	3	17.32	0.504	3711813.572	546014.565
11	0	3	22.032	0.931	3533562.056	53993.652
12	0	3	33.536	0.547	3019382.803	135932.695
13	0	3	55.69	1.729	2844721.862	65727.717
14	0	3	87.129	0.646	3721347.095	59245.802
15	0	3	114.518	1.331	7118970.867	144171.111
16	0	3	150.57	3.98	13052741.42	166682.883
17	0	3	188.052	2.865	23593775.54	362548.149
18	0	3	233.928	4.436	43451445.28	738088.005
19	0	3	278.64	6.693	76155587.76	501383.903
20	0	3	303.413	3.892	112228353.9	969003.748
21	0	3	290.387	5.056	162390931.8	3430973.882
22	0	3	243.535	1.639	242304501.6	8608072.177
23	0	3	189.263	1.051	352163094.2	18593234.63
13	0	4	57.292	1.753	2788187.992	59873.767
14	0	4	89.303	1.44	3654662.547	145141.466
15	0	4	112.32	0.809	7174184.481	247667.038
16	0	4	148.099	4.263	12945073.51	75679.358
17	0	4	190.316	2.809	23817456.21	362849.662
18	0	4	234.931	5.005	43424581.28	374666.141
19	0	4	276.827	3.517	75686593.32	211857.105
20	0	4	306.673	4.105	112476295.8	775496.677
21	0	4	287.735	5.154	160781004.7	4558053.091
22	0	4	244.242	1.344	243464016.5	12076438.39
23	0	4	189.009	1.679	350583164.3	15822572.61
16	0	5	148.207	3.971	13091493.97	252983.409
17	0	5	193.847	5.429	23797481.04	310887.189
18	0	5	232.828	3.864	43764896.03	196767.528
19	0	5	274.506	4.64	75819387.85	516854.958
20	0	5	306.385	5.223	112254312	426331.319
21	0	5	288.236	4.162	161979297.4	1634858.237
22	0	5	241.491	2.279	243080694.9	9227651.419
23	0	5	189.089	1.359	350743196.8	14963819.5
19	0	6	275.523	3.469	76123014.35	946650.689
20	0	6	301.867	8.611	111961538.8	1357248.707
21	0	6	288.467	1.411	161668584.8	2459544.243
22	0	6	242.885	1.266	243385355	9457780.62
23	0	6	188.927	1.768	352804366.1	18667659.5
22	0	7	243.267	1.537	242964575.4	10349595.84
23	0	7	187.924	1.022	351066937.1	19037467.13

Collective data for Enhanced Fibonacci Cube. Simulation duration is 60 seconds.

Dimension	EN	EL	MALatency	AL SD	Mthroughput	Throughput SD
5	0	0	6.118	0.334	1867121.121	32300.797
6	0	0	7.754	0.153	2282489.291	104697.344
7	0	0	10.56	0.249	3092194.369	45348.138
8	0	0	14.422	0.634	3649553.66	46371.699
9	0	0	19.084	0.423	4574387.498	40678.36
10	0	0	19.623	0.302	4071091.475	58355.813
11	0	0	26.128	0.493	4620221.082	132135.598
12	0	0	44.148	1.543	3512369.454	80139.695
13	0	0	78.206	1.528	4530388.221	188164.791
14	0	0	113.292	1.345	6578548.289	109678.648
15	0	0	155.442	3.533	14938441.68	208868.714
16	0	0	202.393	4.199	23031271.55	276336.661
17	0	0	274.073	2.346	48805511.77	270534.491
18	0	0	347.074	7.622	74274771.66	172534.607
19	0	0	390.71	7.771	118787792.3	1045183.748
20	0	0	383.481	5.161	156077315.7	1477467.015
21	0	0	312.276	2.172	251850034.3	12029546.57
22	0	0	246.282	2.134	344473927.4	20065672.56
5	1	0	5.71	0.233	1806157.496	69316.26
6	1	0	8.204	0.551	2274568.786	68934.367
7	1	0	10.208	0.404	3076542.438	86355.255
8	1	0	13.265	0.53	3623835.395	93915.813
9	1	0	18.327	0.438	4555702.767	85419.56
10	1	0	20.713	0.905	4152025.833	182298.417
11	1	0	25.916	0.722	4585471.892	80212.826
12	1	0	42.068	0.482	3642861.426	115700.707
13	1	0	78.27	1.757	4393053.915	109338.073
14	1	0	115.706	1.572	6441498.929	157801.34
15	1	0	152.267	1.944	15046982.34	203165.737
16	1	0	209.334	5.279	23003218.71	177685.27
17	1	0	270.691	2.291	48922694.62	199340.486
18	1	0	346.876	6.341	73737042.87	354749.655
19	1	0	389.226	4.438	119188267.7	863318.596
20	1	0	385.027	3.376	155825532.7	1769757.061
21	1	0	314.34	0.896	249601221	7778148.909
22	1	0	247.042	1.762	344035808.4	18404266.83
7	2	0	11.133	0.398	2771060.221	107246.006
8	2	0	14.65	0.514	3528522.659	63827.118
9	2	0	19.084	1.056	4433637.198	227404.401
10	2	0	20.871	0.834	4062656.006	224152.334
11	2	0	25.628	0.228	4598448.576	150059.556
12	2	0	43.028	0.732	3598269.413	92982.134
13	2	0	76.712	2.691	4262932.036	68213.402

14	2	0	111.339	0.988	6559022.755	80519.54
15	2	0	153.825	1.334	14930533.78	91939.792
16	2	0	202.543	5.096	22896998.21	276669.631
17	2	0	277.992	7.182	48969139.38	297260.358
18	2	0	345.463	10.003	73669825.81	684904.337
19	2	0	389.36	6.259	118588758.3	345219.298
20	2	0	383.166	3.139	156490431	1662560.385
21	2	0	314.969	1.242	249694022.6	9597898.654
22	2	0	246.796	1.627	343678362.9	17699673.98
11	3	0	26.426	1.042	4133120.75	755453.876
12	3	0	42.891	1.204	3574800.007	150248.202
13	3	0	75.908	3.086	4873139.713	733733.12
14	3	0	112.416	1.531	6539897.814	214966.603
15	3	0	152.845	2.237	15017944.48	158367.045
16	3	0	208.456	6.422	22967991.32	91091.56
17	3	0	275.9	2.421	48962135.26	289153.107
18	3	0	346.384	7.898	73936986.34	188041.717
19	3	0	389.487	6.328	119074739.5	819653.202
20	3	0	380.408	3.562	155081077.5	1991664.857
21	3	0	313.956	2.471	250556457.7	8478897.847
22	3	0	246.449	2.823	342505667.2	20639260.24
15	4	0	156.31	3.007	14834361.69	174277.391
16	4	0	206.398	5.587	23057417.98	237013.718
17	4	0	274.663	5.086	48863649.37	315978.506
18	4	0	344.365	4.659	73784511.28	486207.859
19	4	0	389.874	8.033	119112338.6	1245026.195
20	4	0	382.306	7.16	155713921.5	1206456.477
21	4	0	314.385	1.608	250967736.2	7878142.644
22	4	0	246.371	3.922	343893082.2	15642781.34
19	5	0	390.34	9.299	118558303.6	672637.849
20	5	0	383.077	2.094	156335015.9	2237025.259
21	5	0	314.231	2.089	250423555.3	10536274.42
22	5	0	245.701	3.081	345318373.1	14166987.47
5	0	1	5.913	0.606	1738072.97	54078.904
6	0	1	8.454	0.157	2128601.381	57988.136
7	0	1	10.314	0.329	3030071.585	35245.483
8	0	1	14.105	0.305	3712643.842	68783.47
9	0	1	18.817	0.4	4589546.343	145671.878
10	0	1	20.157	0.325	3920837.921	67124.555
11	0	1	25.427	0.462	4383760.221	104734.372
12	0	1	42.895	0.853	3511298.085	94526.91
13	0	1	78.235	0.601	4421358.734	92258.085
14	0	1	112.703	1.924	6550176.688	154399.633
15	0	1	152.969	1.905	14870029.12	149333.778
16	0	1	211.559	5.71	22865480.7	181100.807
17	0	1	271.56	4.735	48709198.79	356366.441
18	0	1	347.174	4.646	74148256.74	467713.211
19	0	1	391.082	3.139	118899222	373324.041

20	0	1	381.917	4.058	156285886.8	1646609.492
21	0	1	313.457	2.209	250591097.2	9896198.119
22	0	1	246.09	4.139	343276434.8	18627791.84
7	0	2	10.861	0.463	2814787.708	134346.328
8	0	2	15.161	0.417	3613595.657	77756.136
9	0	2	20.597	0.891	4056322.533	100120.965
10	0	2	20.915	0.63	3660536.852	101170.579
11	0	2	26.315	0.588	4637002.448	174019.213
12	0	2	43.356	0.808	3532656.083	80737.812
13	0	2	79.142	1.361	4467803.84	140991.044
14	0	2	113.31	1.789	6512454.05	154709.263
15	0	2	154.629	0.897	14996845.95	77816.02
16	0	2	204.266	4.345	22998459.83	364659.954
17	0	2	275.528	6.063	49273206.53	333990.466
18	0	2	347.004	6.016	73913898.29	1628582.269
19	0	2	393.537	3.98	119019819.5	459906.788
20	0	2	383.687	3.15	155959662	1495539.401
21	0	2	314.682	1.553	250923979.6	11023890.63
22	0	2	246.645	1.588	346237730.9	19729948.56
11	0	3	27.622	0.513	4111040.053	419462.646
12	0	3	44.571	0.967	3401995.803	106968.962
13	0	3	77.243	1.586	4410671.022	113806.882
14	0	3	112.682	2.115	6656273.537	103666.951
15	0	3	151.975	2.888	14950240.91	180415.723
16	0	3	206.454	4.427	22983022.88	128445.298
17	0	3	272.376	4.855	48783163.68	504878.403
18	0	3	344.358	7.897	73917757.8	413888.247
19	0	3	394.308	2.715	118612891.1	241032.091
20	0	3	382.58	1.863	156519151.8	1779986.328
21	0	3	313.468	2.54	250977576.2	9411008.265
22	0	3	246.389	4.667	343592958.9	16184394
15	0	4	156.033	2.155	14610265.86	318944.431
16	0	4	199.774	6.628	22947311.83	261322.581
17	0	4	270.988	1.392	48830965.89	307736.668
18	0	4	343.493	6.995	74185642.88	566054.476
19	0	4	390.904	8.031	118380790.8	331526.558
20	0	4	384.115	5.605	156363799.5	1741559.548
21	0	4	314.838	1.682	250898967.1	8157147.348
22	0	4	245.821	3.64	344261416.4	21121078.05
19	0	5	390.806	15.652	118847907.8	1102955.553
20	0	5	382.057	1.05	156022912.9	1447976.38
21	0	5	312.647	2.212	250825025.8	7926472.788
22	0	5	245.92	2.193	344638066.7	19158723.89

Collective data for Extended Fibonacci Cube XFC_1. Simulation duration is 60 seconds.

Dimension	Subscript	EN	EL	MALatency	AL SD	Mthroughput	Throughput SD
21	1	0	0	263.198	2.144	200622752.2	7332553.462
20	1	0	0	299.232	1.542	133804007.5	822463.41
19	1	0	0	293.774	2.818	93071257.8	338624.828
18	1	0	0	261.356	5.957	57534241.64	425847.624
17	1	0	0	213.803	3.865	31539444.04	127224.894
16	1	0	0	165.983	3.745	16699058.48	123745.832
15	1	0	0	132.692	1.857	8988125.719	187721.677
14	1	0	0	102.635	2.163	4735510.016	68645.701
13	1	0	0	70.684	1.9	3192470.325	87453.553
12	1	0	0	41.716	0.815	2714505.551	73940.93
11	1	0	0	25.031	0.312	3275873.753	43656.724
10	1	0	0	19.643	0.232	4164382.708	25607.429
9	1	0	0	16.449	0.483	4091118.929	102616.141
8	1	0	0	13.589	0.435	3763136.654	61615.341
7	1	0	0	10.506	0.525	3022916.698	40881.233
6	1	0	0	8.398	0.441	2544305.343	68773.57
5	1	0	0	6.111	0.548	1997666.045	129741.239
21	1	1	0	265.219	3.161	199088120.6	7303813.555
20	1	1	0	296.97	2.504	133436741.5	771502.698
19	1	1	0	293.887	1.872	92905390.07	1351086.941
18	1	1	0	258.198	2.785	57444466.25	333138.835
17	1	1	0	215.471	4.197	31525904.34	249118.385
16	1	1	0	167.814	3.105	16841782.87	237712.569
15	1	1	0	134.265	1.759	9088189.582	157547.852
14	1	1	0	100.534	0.458	4783634.634	277797.116
13	1	1	0	70.75	1.362	3236308.469	117527.675
12	1	1	0	40.924	1.132	2706430.195	45522.29
11	1	1	0	24.75	1.21	3481050.429	737040.567
10	1	1	0	19.234	0.312	4155260.1	83642.79
9	1	1	0	16.534	0.351	4350247.757	72987.535
8	1	1	0	13.743	0.588	3704667.997	118894.186
7	1	1	0	10.714	0.189	3059107.766	59656.619
6	1	1	0	7.885	0.217	2495627.764	29566.618
5	1	1	0	6.091	0.224	2052280.149	66398.359
21	1	2	0	265.971	1.285	200331426.7	6732831.867
20	1	2	0	299.762	2.077	133444323.9	878399.877
19	1	2	0	292.358	2.463	93298069.21	533850.245
18	1	2	0	259.587	2.886	57502706.45	305976.943
17	1	2	0	210.825	3.521	31619763.34	230008.725
16	1	2	0	169.772	4.991	16732718.49	183719.05
15	1	2	0	132.692	2.691	8940323.215	117265.276
14	1	2	0	102.837	2.337	4807856.749	81899.904

13	1	2	0	70.127	1.916	3131084.288	71053.123
12	1	2	0	41.236	1.931	2918028.031	737677.167
11	1	2	0	24.933	0.264	3314056.278	523110.038
10	1	2	0	19.363	0.796	4149873.43	250568.257
9	1	2	0	17.199	1.136	4550593.055	228117.283
8	1	2	0	14.096	0.481	3758240.471	118674.347
7	1	2	0	10.463	0.463	2949211.73	42588.169
6	1	2	0	7.756	0.362	2513267.792	40896.691
5	1	2	0	6.256	0.377	1901078.027	77532.58
21	1	3	0	265.262	4.414	199684343.4	7468818.212
20	1	3	0	296.77	1.682	133274466.3	1366219.362
19	1	3	0	294.193	2.314	92832158.88	574327.548
18	1	3	0	259.561	2.207	57398983.78	249533.696
17	1	3	0	210.228	3.894	31315760.46	290424.634
16	1	3	0	171.007	2.464	16590856.54	333729.919
15	1	3	0	133.238	1.655	9043464.201	152959.693
14	1	3	0	102.363	2.097	4789679.141	70501.799
13	1	3	0	68.799	1.66	3263554.472	68176.457
12	1	3	0	41.952	0.812	2796851.13	81633.806
11	1	3	0	24.405	1.378	3519281.432	768508.652
10	1	3	0	19.434	1.013	4285503.26	397783.673
9	1	3	0	17.385	1.334	4710003.484	272999.955
8	1	3	0	14.669	0.286	3950009.379	188964.102
21	1	4	0	266	3.273	200881711.6	7309829.236
20	1	4	0	296.992	2.113	133419870.7	1229102.729
19	1	4	0	292.927	2.508	92883316.93	229330.581
18	1	4	0	257.964	5.649	57688070.44	1426735.889
17	1	4	0	211.891	4.025	31475005.84	361711.042
16	1	4	0	165.688	3.052	16768644.21	149272.382
15	1	4	0	131.462	1.222	8905400.523	155797.718
14	1	4	0	102.64	1.61	4776977.777	26544.413
13	1	4	0	67.502	0.968	3159847.425	366031.919
12	1	4	0	41.317	1.191	3572525.236	471813.232
11	1	4	0	24.709	0.814	3547430.753	334499.955
21	1	5	0	265.047	3.059	201178621.7	8274876.642
20	1	5	0	296.934	3.088	133606854.8	666252.068
19	1	5	0	292.911	4.067	92802440.8	920262.299
18	1	5	0	260.374	4.093	57366672.28	220644.97
17	1	5	0	214.567	2.777	31545974.96	189719.366
16	1	5	0	168.595	4.15	17003736.79	318931.2
15	1	5	0	131.737	1.697	9017893.049	309165.247
14	1	5	0	102.002	2.825	4980801.951	316371.054
21	1	6	0	266.969	1.374	199302577.7	7297633.83
20	1	6	0	297.524	1.572	133976585.7	1177269.866
19	1	6	0	292.98	1.61	93147551.45	627383.95
18	1	6	0	257.031	6.869	57740320.65	559855.318
17	1	6	0	217.06	3.138	31522824.11	460011.848
21	1	7	0	266.117	1.682	200664295	6536290.811

20	1	7	0	300.143	1.94	132977548.1	609171.502
21	1	0	1	265.572	2.172	199385330.5	5733065.957
20	1	0	1	297.015	2.151	133627413.9	1255472.491
19	1	0	1	294.224	3.016	92647415.64	665297.084
18	1	0	1	255.241	1.876	57350700.07	422341.553
17	1	0	1	208.725	5.344	31796454.76	460435.078
16	1	0	1	168.918	4.693	16724094.66	99699.997
15	1	0	1	132.915	2.042	8991897.849	133163.856
14	1	0	1	99.754	2.007	4776551.37	125197.212
13	1	0	1	70.186	1.208	3202758.981	36236.102
12	1	0	1	40.987	0.853	2716972.14	33192.018
11	1	0	1	24.418	0.263	3329010.302	31272.175
10	1	0	1	19.422	0.301	4359067.714	115008.857
9	1	0	1	16.275	0.262	4450664.645	85206.181
8	1	0	1	13.609	0.742	3905100.304	89791.69
7	1	0	1	11.101	0.359	2964917.13	37992.225
6	1	0	1	8.498	0.226	2445427.197	57025.154
5	1	0	1	6.005	0.461	2026803.782	72590.5
21	1	0	2	265.967	1.687	199600120.9	5273566.624
20	1	0	2	295.156	3.415	134119309.3	884143.156
19	1	0	2	292.187	1.405	92992569.53	297372.361
18	1	0	2	257.205	4.534	57560594.59	613404.955
17	1	0	2	212.226	1.638	31203823.75	323857.221
16	1	0	2	168.068	3.747	16691231.79	135221.991
15	1	0	2	133.163	1.653	8997325.64	93368.63
14	1	0	2	102.299	1.638	4740069.734	68881.209
13	1	0	2	69.697	1.967	3169418.418	59662.239
12	1	0	2	40.487	0.353	2719432.101	51808.733
11	1	0	2	24.952	0.37	3193031.031	38434.247
10	1	0	2	19.666	0.898	4100552.058	364657.107
9	1	0	2	16.683	0.834	4485865.045	279743.274
8	1	0	2	13.677	0.194	3791233.691	76439.006
7	1	0	2	10.696	0.207	3052559.3	48833.829
6	1	0	2	8.305	0.469	2570836.011	40704.135
5	1	0	2	6.449	0.497	2121097.91	39028.761
21	1	0	3	264.61	2.163	199448640	5780222.65
20	1	0	3	297.657	2.243	133299337.4	1238084.378
19	1	0	3	293.128	2.175	93308591	800260.93
18	1	0	3	256.248	1.703	57419114.64	340689.725
17	1	0	3	212.189	1.858	31243565.65	389286.49
16	1	0	3	173.482	6.424	16875165.17	156700.73
15	1	0	3	132.887	0.998	9003571.707	187107.415
14	1	0	3	101.812	2.216	4841758.091	117702.954
13	1	0	3	68.596	1.659	3306859.903	66659.529
12	1	0	3	41.719	1.012	2727866.412	60890.266
11	1	0	3	24.598	0.615	3179344.055	42413.222
10	1	0	3	19.421	0.267	3933510.969	118726.283
9	1	0	3	16.637	0.377	4134940.935	122416.438

8	1	0	3	13.833	0.263	3769035.305	45632.055
21	1	0	4	266.315	0.584	199320506.1	5795558.376
20	1	0	4	297.983	4.193	133773502.2	1081045.327
19	1	0	4	295.06	3.178	93237995.58	755807.748
18	1	0	4	258.236	3.077	57144973	356155.614
17	1	0	4	209.739	1.502	31286770.17	308067.937
16	1	0	4	165.187	4.429	16766313.59	238018.662
15	1	0	4	131.898	3.021	8947275.425	101865.77
14	1	0	4	103.619	1.135	4751042.126	106725.325
13	1	0	4	69.624	2.041	3144186.163	59149.646
12	1	0	4	41.893	0.871	2719853.494	61261.43
11	1	0	4	24.457	0.632	3222281.782	46398.769
21	1	0	5	266.437	2.069	199165263	5403703.717
20	1	0	5	296.127	3.09	133332157.9	334061.544
19	1	0	5	293.935	3.089	93250037.55	2847316.239
18	1	0	5	255.161	2.673	57278709.4	527434.724
17	1	0	5	213.272	4.04	31349854.73	177993.277
16	1	0	5	171.146	3.825	16690803.19	135721.406
15	1	0	5	132.203	1.531	9027390.502	87184.27
14	1	0	5	101.292	3.262	4846336.806	134941.318
21	1	0	6	265.403	4.977	200817858.7	7238988.242
20	1	0	6	295.625	3.309	133099205.8	912861.232
19	1	0	6	293.273	1.718	92802155.01	772981.447
18	1	0	6	256.141	2.614	57368258.19	394580.88
17	1	0	6	213.501	2.389	31458945.42	98421.339
21	1	0	7	266.257	1.361	199405100.1	6124989.436
20	1	0	7	296.987	2.688	133764361.4	1793139.211
5	4	0	0	6.996	0.208	3046390.319	127271.508
6	5	0	0	10.093	0.383	4667176.013	95245.157
7	6	0	0	13.31	0.416	6313704.597	131947.345
8	7	0	0	15.935	1.056	6279516.186	166424.374
9	8	0	0	27.196	0.252	7334754.301	74366.35
10	9	0	0	44.961	0.776	13764698.95	190518.435
11	10	0	0	70.103	0.567	28554984.68	345918.469
12	11	0	0	106.431	1.369	58404606.18	527898.822
13	12	0	0	148.386	1.39	111371365.7	455649.988
14	13	0	0	170.444	2.784	193501166.9	1435244.624
15	14	0	0	157.378	0.81	346341553.8	9767170.654
5	4	1	0	6.939	0.349	2904603.881	159899.65
6	5	1	0	10.773	0.199	4439517.649	48139.175
7	6	1	0	14.148	0.707	6083592.593	126975.738
8	7	1	0	16.516	0.345	6147552.167	177640.208
9	8	1	0	27.989	0.69	7593261.941	308897.363
10	9	1	0	45.293	0.406	14064062.68	394192.369
11	10	1	0	71.351	1.245	28428847.63	342288.559
12	11	1	0	107.561	0.593	58121355.85	320391.082
13	12	1	0	147.194	2.58	110715573.5	614928.394
14	13	1	0	170.908	2.317	193452666.5	1099476.623

15	14	1	0	156.89	0.503	347934785.8	7977134.759
5	4	2	0	7.189	0.207	2893658.421	107783.017
6	5	2	0	10.406	0.411	4567365.966	83543.657
7	6	2	0	13.807	1.585	6256001.046	170926.584
8	7	2	0	15.898	0.542	6075238.647	147433.674
9	8	2	0	27.457	1.021	7502926.896	133500.389
10	9	2	0	45.088	0.416	14019681.22	137883.768
11	10	2	0	70.579	1.045	28315185.56	149717.532
12	11	2	0	106.924	1.485	58292679.22	328786.053
13	12	2	0	148.029	1.38	111327053.7	123739.399
14	13	2	0	170.767	2.733	192553922.8	1775080.445
15	14	2	0	156.661	0.789	347418253.5	10375803.85
5	4	3	0	7	0.146	2739784.172	179562.55
6	5	3	0	10.665	0.251	4400795.826	89770.126
7	6	3	0	16.063	1.793	6251088.944	390011.29
8	7	3	0	16.259	3.022	6092019.233	808084.973
9	8	3	0	28.104	0.257	7572503.963	140305.981
10	9	3	0	44.47	0.773	14027080.82	116070.111
11	10	3	0	71.918	1.28	28432228.07	195548.586
12	11	3	0	105.119	1.716	58228688.11	187552.559
13	12	3	0	149.372	2.081	111134159.2	865932.664
14	13	3	0	171.05	1.752	193229950.5	1196041.28
15	14	3	0	156.421	0.747	347456875.4	8350103.319
5	4	4	0	7.492	0.3	2792660.961	131947.574
6	5	4	0	10.757	0.161	4370231.437	87749.177
7	6	4	0	13.838	1.904	6194980.013	164195.208
8	7	4	0	16.788	0.626	6399213.129	221580.161
9	8	4	0	27.473	0.41	7343743.137	236726.238
10	9	4	0	45.447	0.877	13859678.23	186995.459
11	10	4	0	71.262	1.336	28324951.07	248040.651
12	11	4	0	106.25	0.681	58385494.3	278939.133
13	12	4	0	147.661	1.159	111109710.4	385580.968
14	13	4	0	171.331	2.636	193922825.4	1274647.738
15	14	4	0	157.254	0.999	346514751.7	7540128.137
6	5	5	0	10.775	0.274	4022610.738	223538.391
7	6	5	0	16.318	1.305	6200325.815	212043.521
8	7	5	0	17.534	1.932	6255143.692	534857.59
9	8	5	0	28.289	4.289	7782234.333	1551388.438
10	9	5	0	45.608	0.748	14100033.61	495740.525
11	10	5	0	71.401	1.126	28462490.59	195105.138
12	11	5	0	107.56	1.133	58157575.14	584658.02
13	12	5	0	149.457	3.545	111132527.7	824398.498
14	13	5	0	172.352	2.523	192788940.4	1433044.25
15	14	5	0	156.837	0.678	347646415.8	9153185.065
7	6	6	0	16.981	1.942	5859901.675	659189.695
8	7	6	0	20.346	2.962	7299854.515	623134.316
9	8	6	0	27.847	0.79	7509677.564	289007.292
10	9	6	0	45.19	0.817	14015130.96	415881.124

11	10	6	0	70.997	1.519	28114803.2	675272.084
12	11	6	0	107.201	0.585	58160213.51	149942.024
13	12	6	0	148.688	2.72	110791288.8	562550.443
14	13	6	0	172.595	3.356	192156973.4	2101961.334
15	14	6	0	157.251	0.771	347672230.7	8394355.957
8	7	7	0	21.603	4.586	6907788.422	1178227.283
9	8	7	0	29.294	0.863	7596715.165	330899.676
10	9	7	0	46.199	0.781	14087613.81	286269.539
11	10	7	0	70.86	1.151	28641216.65	232329.467
12	11	7	0	107.699	1.328	58192091.67	135172.158
13	12	7	0	146.929	2.536	111165305.7	538515.66
14	13	7	0	171.309	2.507	192740750.9	718702.298
15	14	7	0	157.146	0.916	345187885.8	7408504.968
9	8	8	0	28.74	0.399	7372820.282	272652.915
10	9	8	0	45.26	0.865	13924882.68	138739.332
11	10	8	0	71.335	0.927	28261354.27	254365.414
12	11	8	0	107.097	1.4	57847402.81	622371.801
13	12	8	0	147.411	1.618	110916538.3	775369.518
14	13	8	0	171.457	2.965	192409636.3	2235970.311
15	14	8	0	156.942	0.305	347539294.2	8850733.981
10	9	9	0	46.399	1.053	13495376.2	287664.548
11	10	9	0	71.391	0.721	28290302.93	294095.098
12	11	9	0	107.919	1.427	58318546.74	245064.371
13	12	9	0	149.891	4.269	110257451.3	637700.94
14	13	9	0	170.507	3.169	193015906.2	2094651.531
15	14	9	0	156.304	0.686	347221450	12457080.79
11	10	10	0	71.69	1.367	28035602.92	624584.138
12	11	10	0	108.578	2.393	58306790.31	652080.775
13	12	10	0	148.736	3.193	110390844.1	823003.364
14	13	10	0	171.153	2.205	192640297.3	903856.609
15	14	10	0	156.025	1.468	344956233.7	9869207.002
12	11	11	0	107.92	2.363	57798658.24	778117.187
13	12	11	0	149.658	2.475	110765264.8	202992.59
14	13	11	0	171.021	3.55	192151673.6	1964048.968
15	14	11	0	156.872	1.548	346225642.4	11153337.19
13	12	12	0	148.78	3.327	110570940	1236820.907
14	13	12	0	171.908	2.281	192509583.2	1060578.947
15	14	12	0	157.175	0.908	345023942.9	8548642.995
14	13	13	0	172.762	2.749	192678761.5	1179273.101
15	14	13	0	156.797	1.708	346159779.1	10638850.57
15	14	14	0	156.487	0.671	347135456.5	1128675.143
5	4	0	1	7.224	0.305	2976237.719	151923.069
6	5	0	1	10.049	0.381	4649847.616	67300.762
7	6	0	1	13.14	0.57	6067049.682	223415.446
8	7	0	1	16.399	0.627	6270192.849	64958.029
9	8	0	1	28.401	0.597	7323173.757	171772.214
10	9	0	1	45.787	0.621	13785636	260939.184
11	10	0	1	70.874	1.038	28360853.57	472101.182

12	11	0	1	106.394	2.2	58225078.46	225717.316
13	12	0	1	147.892	2.882	111050144	525464.019
14	13	0	1	171.819	4.282	193534296.8	1441738.331
15	14	0	1	157.467	1.078	346726973.4	9821532.467
5	4	0	2	6.903	0.573	2928395.078	144456.924
6	5	0	2	11.119	0.419	4392884.992	108882.996
7	6	0	2	14.671	1.044	5945086.41	179405.474
8	7	0	2	17.436	0.769	6194750.304	277585.981
9	8	0	2	27.958	0.352	7466378.595	288416.214
10	9	0	2	45.471	0.263	14197258.9	156298.66
11	10	0	2	71.116	0.981	28576008.4	105238.943
12	11	0	2	107.04	1.04	58470547.31	332405.668
13	12	0	2	148.784	1.678	111314290.1	278061.67
14	13	0	2	172.054	4.119	193137404.7	1641426.254
15	14	0	2	156.818	0.639	346402694.3	8773904.119
5	4	0	3	7.317	0.359	3044738.527	142753.738
6	5	0	3	10.151	0.184	4604233.797	78807.059
7	6	0	3	12.778	0.82	6153807.814	87270.029
8	7	0	3	16.028	0.373	6066570.649	195700.828
9	8	0	3	27.438	0.338	7293597.319	125286.199
10	9	0	3	45.65	0.375	13973645.07	139020.09
11	10	0	3	70.903	1.051	28518298.2	170219.562
12	11	0	3	106.446	1.566	58169500.68	417764.978
13	12	0	3	147.888	1.495	111500054.5	401595.324
14	13	0	3	170.517	1.65	193329573.4	487962.667
15	14	0	3	157.265	1.261	347220229.2	8141063.129
5	4	0	4	7.432	0.371	2996061.425	111059.02
6	5	0	4	10.637	0.423	4459197.67	123195.448
7	6	0	4	14.765	0.709	6163835.606	82880.013
8	7	0	4	17.215	0.789	6214966.041	257967.745
9	8	0	4	28.523	1.008	7575435.268	221482.715
10	9	0	4	45.785	0.655	14191702.43	209666.025
11	10	0	4	71.227	1.106	28566715.29	230027.908
12	11	0	4	106.948	0.764	58129429.83	243867.414
13	12	0	4	148.71	2.247	111498674.8	497863.165
14	13	0	4	170.067	3.322	193548818.6	1522675.71
15	14	0	4	157.36	1.827	346203141	10794851.4
6	5	0	5	10.178	0.725	4159785.493	307999.688
7	6	0	5	12.889	1.582	6314942.118	227088.176
8	7	0	5	16.162	0.427	6229874.567	223950.713
9	8	0	5	27.432	0.699	7214464.762	229529.685
10	9	0	5	45.568	0.853	13848597.5	209301.952
11	10	0	5	70.413	0.653	28475517.1	269055.332
12	11	0	5	107.543	1.356	58242633.95	127149.111
13	12	0	5	147.666	1.591	111302190.5	596135.33
14	13	0	5	171.026	2.396	193525774.2	1358971.175
15	14	0	5	156.921	1.657	348831834.4	10530220.86
7	6	0	6	15.285	0.592	5484434.268	544077.969

8	7	0	6	17.182	0.835	6162744.09	307916.091
9	8	0	6	28.49	0.584	7681289.709	188870.48
10	9	0	6	45.802	0.461	14103017.62	265478.118
11	10	0	6	71.168	1.429	28354888.59	123432.986
12	11	0	6	106.456	2.491	58509059.51	577638.168
13	12	0	6	149.148	2.139	110689261.8	422471.024
14	13	0	6	170.871	2.231	194340436.2	1205155.304
15	14	0	6	157.838	0.952	346974953.6	9151231.64
8	7	0	7	17.177	0.441	5784399.355	388582.958
9	8	0	7	27.658	0.367	7374922.631	824005.562
10	9	0	7	45.039	0.899	14022578.19	215119.657
11	10	0	7	70.793	0.275	28359340.44	232704.911
12	11	0	7	106.793	1.472	57984436.07	337678.25
13	12	0	7	148.855	0.938	111501156	3882276.469
14	13	0	7	171.712	4.365	193775717.3	1646256.46
15	14	0	7	156.456	0.658	347420505.1	7747460.256
9	8	0	8	27.646	1.169	7198566.799	666937.619
10	9	0	8	45.458	0.828	13949458.61	131589.539
11	10	0	8	70.737	1.849	28315670.01	423897.943
12	11	0	8	107.716	1.662	58459451.98	331545.658
13	12	0	8	148.901	2.988	111752800	323325.79
14	13	0	8	170.905	3.354	193222046.3	2115076.761
15	14	0	8	156.792	1.755	348288594.8	11387147.18
10	9	0	9	45.398	0.944	13827736.11	279760.033
11	10	0	9	71.914	0.591	28658313.96	416963.472
12	11	0	9	106.248	2.072	58090655.99	684261.11
13	12	0	9	149.353	1.959	110968846.4	291893.08
14	13	0	9	171.481	3.195	192688093.3	413870.59
15	14	0	9	157.335	0.564	346845486.8	9060549.088
11	10	0	10	71.419	0.968	28535022.93	582933.744
12	11	0	10	106.797	1.458	58333715.82	200604.098
13	12	0	10	146.616	2.117	111353300	499563.935
14	13	0	10	170.562	2.535	192612127.9	1862913.429
15	14	0	10	156.384	1.076	346119679.1	11757862.09
12	11	0	11	106.627	2.371	58128060.98	326238.498
13	12	0	11	147.524	2.227	110588975.1	2418652.134
14	13	0	11	171.576	2.383	193255779.8	1807463.352
15	14	0	11	156.645	0.864	347981099.6	8414212.546
13	12	0	12	148.082	3.651	111212805.3	280511.406
14	13	0	12	170.271	1.373	192453849	1198149.646
15	14	0	12	156.938	0.79	347324557.3	8524685.337
14	13	0	13	171.32	3.062	193543984.1	1997670.176
15	14	0	13	156.49	2.153	348944051.6	12000917.35
15	14	0	14	156.886	1.082	347372413.8	1332908.132

Collective data for Gaussian Cube $GC(n,2^a)$. Simulation duration is 60 seconds.

Dimension	alpha	EN	EL	MALatency	AL SD	Mthroughput	Throughput SD
4	0	0	0	16.478	1.101	1014948.407	14421.665
5	0	0	0	19.629	0.603	1642427.059	16317.775
6	0	0	0	26.055	0.459	2564353.717	26780.176
7	0	0	0	29.295	1.056	3137559.255	110083.552
8	0	0	0	38.268	1.535	3663865.425	105516.024
9	0	0	0	60.386	1.148	5817991.134	176886.629
10	0	0	0	96.235	1.186	11200003.98	108042.277
11	0	0	0	144.969	1.304	22498200.47	101143.502
12	0	0	0	204.232	6.55	43263557.45	271537.391
13	0	0	0	262.707	2.039	72886951.2	575698.986
14	0	0	0	278.187	4.472	124317207.8	1644591.009
14	1	0	0	360.481	3.602	131736072.6	378251.439
14	2	0	0	474.811	22.763	130652414.8	333566.087
14	3	0	0	556.398	1.752	109795726.5	265873.96
14	4	0	0	729.073	23.195	80735616.85	343121.33
14	5	0	0	728.018	21.316	67800669.19	167683.468
14	6	0	0	685.395	9.742	56271109.76	73072.555
14	7	0	0	712.927	1.135	41897420.08	153843.086
14	8	0	0	730.201	5.548	28499073.51	39518.5
14	9	0	0	849.952	5.03	19986746.05	55838.537
14	10	0	0	828.24	11.1	11622718.54	247973.241
14	11	0	0	869.195	13.273	8195355.179	188334.684
14	12	0	0	812.87	6.632	5868188.759	202647.123
14	13	0	0	799.375	10.811	5472558.749	251256.536
4	0	1	0	16.056	1.27	973346.46	10773.785
5	0	1	0	20.583	0.967	1605887.863	15965.111
6	0	1	0	30.108	0.99	2573076.862	7615.069
7	0	1	0	52.527	2.272	3741495.795	20390.439
8	0	1	0	63.389	4.023	5086352.034	115188.064
9	0	1	0	78.195	1.849	8114327.751	93044.31
10	0	1	0	102.528	1.173	13346409.13	132200.752
11	0	1	0	146.67	2.637	22852337.77	196863.149
12	0	1	0	199	5.619	41285418.55	374940.42
13	0	1	0	254.276	5.33	71762647.47	494324.092
14	0	1	0	274.143	5.239	121477299.5	586216.824

APPENDIX VII

A New Approach to Routing in Hypercube Based on Fuzzy Neural Network

VII.1 Two architectures of decision-making using Fuzzy Neural Network

There are two ways to use Fuzzy Neural Networks (FNNs) for decision-making. The first is called implicit system, in which all possibly related information is fed into the FNN. Then the FNN outputs the result of decision. For example, in the area of financial market decision-making, the architecture of implicit trading is illustrated in Figure VII.1.

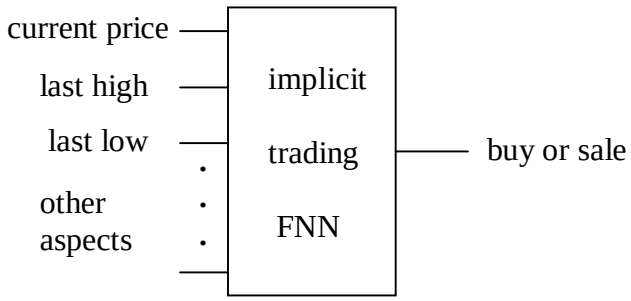


Figure VII.1 architecture for implicit trading

The second architecture is called explicit processing. Here FNN is used only as a component while traditional algorithms are also incorporated. Figure VII.2 and VII.3 show examples of this explicit processing system.

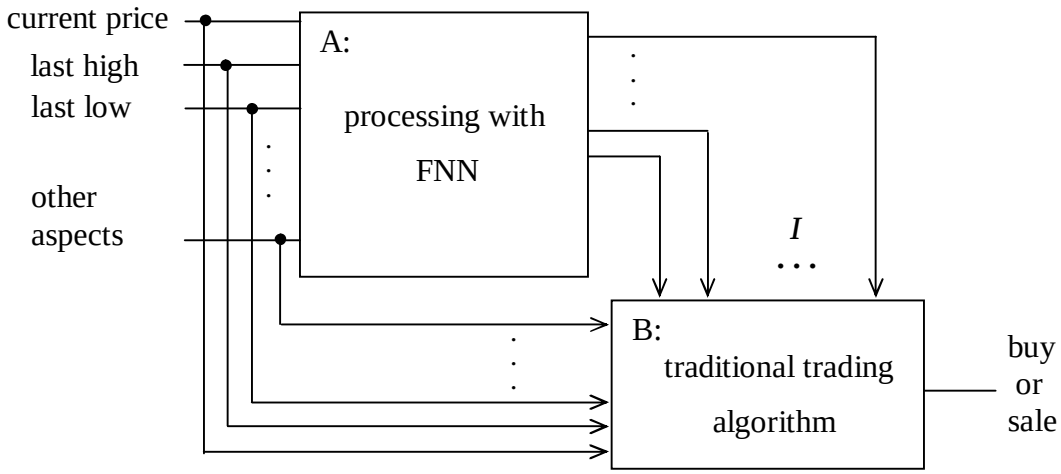


Figure VII.2 one architecture for explicit trading

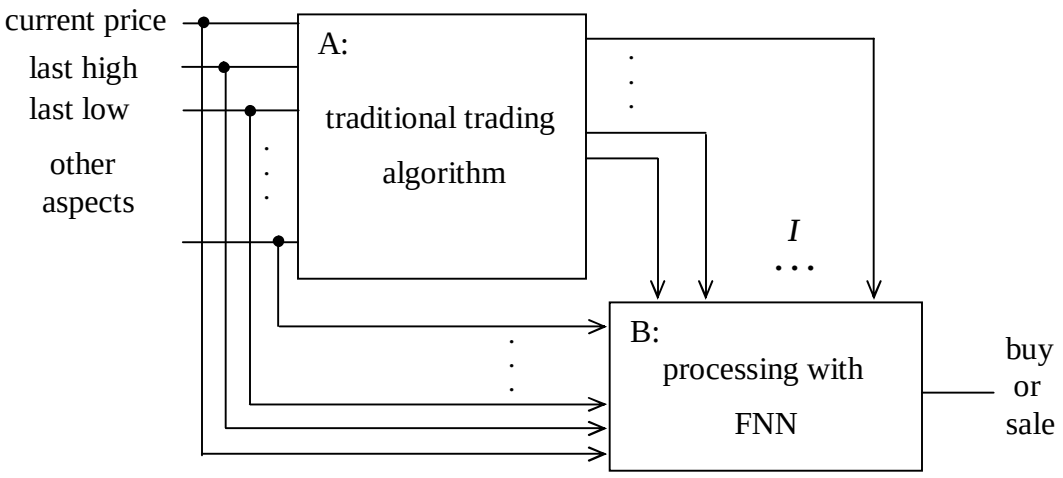


Figure VII.3 another architecture for mixed explicit trading

Here, *I* stands for the intermediate results produced by A, and inputted to B. In this trading example, *I* may encompass the prediction of the price of one hour later or three days later.

Actually, the mixed structure reflects a decomposition of the original problem. Some tasks can be efficiently done by FNN, especially learning and predicting. However, some other jobs maybe more suitable for traditional approaches. A case in point is learning bitwise XOR operation, on which most existing routing algorithms depend. It can be easily proven that for learning the XOR function between two *n*-bit binary numbers or two decimal numbers both ranging from 0 to 2^n - 1, FNN must use $O(2^n)$ rules. However, this function can be realized by hardware in one clock cycle. So by carefully and properly dividing tasks into different functional components (A or B), the original problem can be solved far more efficiently than purely using FNN or traditional algorithms.

VII.2 Design of input and output of FNN

If the explicit architecture is to be adopted, then the first challenge lies in the decomposition of the task. What is to be done by FNN and what is supposed to be done by traditional algorithms? What is the proper interface? From the angle of FNN, these questions are equivalent to what is the input and output of FNN.

As the space and time cost for gathering global information is too high and such information is too intractable for FNNs, it is more feasible to use local information. One type of such strategy is to base the routing decision solely on the status of links incident to current node. For binary hypercube, this simple strategy can achieve good performance [6]. A more far-sighted approach is to take into consideration the status of the links incident to all of the current node's neighbors. We call it 1-hop look-ahead. For example, in Figure VII.4, the routing decision made at P not only incorporates the status of links from P to A, B, C, and D, but also considers the status of e_1, e_2, …, e_{10}.

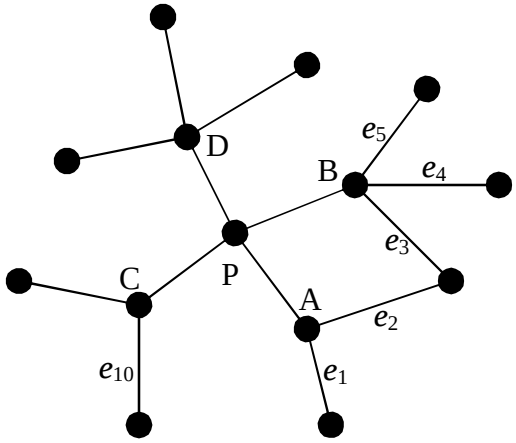


Figure VII.4 Illustration for 1-hop look-ahead approach

So for each packet that arrives at P, say from A, P uses a new metric M to compare all the possible outlet ports. This metric is based on B, C, and D's link status, packet destination, and encoded history that helps to avoid deadlock and livelock. M can be a tuple of several crisp values or fuzzy values, or combination. If we view the crisp values as fuzzy values in the form of singleton, then M is actually a set of fuzzy values. This process is also known as feature selection. It helps to reduce the number of total factors that require to be considered in the next step of comparison. The number of final rules will also be reduced significantly (possibly exponentially) with this horizontal reduction.

With regard to comparison, as we are only interested in the best alternative, there is no need to rank all neighbors according to their corresponding M and pick up the highest one. That approach costs time complexity $O(n\log n)$, where n is the network dimension.

Instead, only $O(n)$ comparisons are needed to derive the best one. This comparison is suitable for FNN. The mechanism is illustrated in Fig. VII.5.

VII.3 Choice of *M*

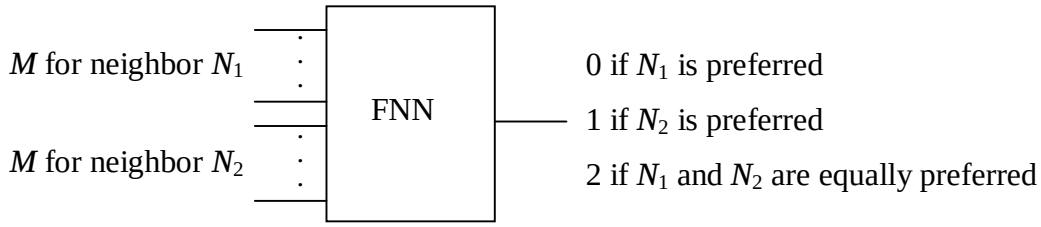


Figure VII.5 mechanism of comparison by FNN

The choice of *M* is critical for the whole strategy. It can not include any binary value (or its corresponding decimal value) related to node address. Otherwise, the number of rules will inevitably grow exponentially with network dimension.

It should also be applicable to all kinds of fault distributions. It is our goal that one fuzzy neural network be used for evenly distributed faults, concentrated faults and other types of distribution. So for different underlying fault distributions, different parts of the rule base in FNN are to be fired so that the system has adaptivity to fault distribution. This requires that the input of FNN under different fault distribution types should also be sufficiently discriminable.

Lastly, *M* should contain or encode enough information that can 'deduce' the result of comparison. One possible design is to introduce three fuzzy variables called optimistic distance, pessimistic distance and neutral distance from respective neighbors to the destination. Such fuzzy values are calculated based on the neighbors' address, packet destination and the status of links incident to the neighbor. For example, the membership function for the three fuzzy variables might be like Figure VII.6:

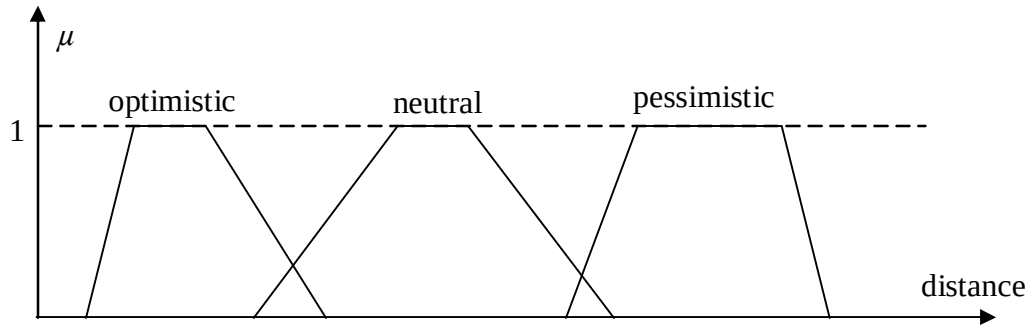


Figure VII.6 membership function of possible fuzzy variables

VII.4 Generating training examples

As the routing strategy is based only on the information of connectivity within 2 hops' distance, there is no point in allocating a faulty component over 2 hops away from current node. In other words, if we are focusing on node 0^n (n straight 0's) i.e. collecting training examples by examining routing decisions at 0^n, then we can locate all faulty nodes in S = $\{ a_{n-1}a_{n-2}...a_1a_0 \mid a_i \in \{0,1\} \ \ for\, i \in [0, n-1]$ and $\sum_{i=0}^{n-1} a_i \le 2 \}$ with all faulty links in $S \times S$. Otherwise, the training example set will be inconsistent.

Start off simulating the network communication and focus on the packets arriving at node 0^n. In Figure 5, M for N_1 and N_2 are easily available. Whether N_1 or N_2 is preferred is decided by applying Dijastra's shortest path algorithm. If they are not equally preferred, then we can exchange the M for N_1 or N_2 and exchange the result of preference. Thus one example can be made use of twice.

VII.5 Combining FNN and traditional algorithms

The whole picture of the routing strategy is as follows. Each node maintains an n-bit fault vector F that records the status of local links. If the link on the corresponding dimension is faulty, then the corresponding bit in F is 0. Otherwise it is 1. The packet overhead is composed of the destination address and an n-bit traversal vector DT. At the

source, DT is set to straight 1's. Whenever a preferred dimension is used, the corresponding bit in DT is masked to 0. And all dimensions masked by 0 in DT can not be used as spare dimensions any more. When a packet is received, the router calculates the optimistic, neutral and pessimistic distance from all neighbors to the packet's destination, except those that are faulty (as is recorded in F) and those that are masked by DT. Finally, FNN is used to determine the best outlet port.

VII.6 Problems

The major problem here is that there has already been saturated research in this area of network routing. One algorithm uses the similar strategy [4]. It first examines non-faulty preferred dimensions. If there are more than one preferred dimensions available, then it chooses a neighbor on a preferred dimension that has least faulty incident components. If there is no non-faulty preferred dimension, then it chooses a neighbor on a spare dimension that has least faulty incident components. Rigorous theoretical deduction is available to demonstrate that this algorithm generates deadlock and livelock free routes. It also has a route with strictly bounded length and the message overhead and time for making routing decision are both $O(n)$. It is very easy to be physically implemented. So it has already provided a set of rules and choice of M that are applicable to hypercube and its symmetric variants with satisfactory performance.

Let us go back to the motivation of using FNN. We adopt it with an eye to deriving a unified or generalized routing strategy for as many variants of binary hypercube as possible. However, without considering binary address, the current approach to using FNN is not suitable for asymmetric networks, which is the majority of hypercubic variants.

Appendix VIII User's Guide

This guide includes the usage of software simulation tool and introduces the source code of FPGA implementation written in Handel-C.

VIII.1 Using software simulation tool

The simulation tool is called SimuRt. It can simulate three types of Fibonacci-class Cube and Gaussian Cube. There are two things to be specified before running simulation: parameters in the code and testing cases in the input file.

VIII.1.1 Setting parameters

The following parameters must be set according to the computer platform and testing objective:

#define PENTIUMSPEED 2048.0
It is defined in file Structure.cpp. It specifies the speed of CPU. The unit is MHz.

#define BUFFER_SIZE 10
It is defined in Common.h. The influence of BUFFER_SIZE on the simulation result is discussed in Chapter 8.

#define NO_READINGS 5
Defined in Common.h, it specifies how many rounds of test are carried out for each testing case.

VIII.1.2 Input file

It is driven by an input file, in which all the testing cases are enumerated and the simulation is run on a batch mode. The input file is named as "input.txt". It should be placed in the same directory of the executable file. If running under Visual C++, then it should be placed in the working directory (specified in Project:\\settings\Debug\Working

Directory). For example, if the input file is as follows:

```
Begin
1   15   2   3   1   500   60
2   13   3   0   1   500   50
3   11   10  5   2   1     500   60
4   11   3   1   500   60
0
```

then

'Begin' means the beginning of testing cases. All characters before 'Begin' are filtered so that it is possible to add some comments at the beginning of the file as long as the string 'Begin' does not appear in the comments.

Line 1: '1' means the testing case is for regular Fibonacci Cube. '15' means the dimension is 15 (strictly speaking, it means we are testing a regular Fibonacci Cube of order 17). '2' means that the number of faulty nodes is 2. '3' means that there are three faulty links. '1' means that packets are generated according to even distribution. '500' means that the hop time is 500ns. '60' means that the simulation runs as long as 60 seconds.

Line 2: '2' means the testing case is for Enhanced Fibonacci Cube. '13' means the dimension is 14 (strictly speaking, it means we are testing an Extended Fibonacci Cube of order 15). '2' means that the number of faulty nodes is 3. '0' means that there is no faulty link. '1' means that packets are generated according to even distribution. '500' means that the hop time is 500ns. '50' means that the simulation runs as long as 50 seconds.

Line 3: '3' means the testing case is for Extended Fibonacci Cube. '11' means the dimension is 11 (strictly speaking, it means we are testing an Extended Fibonacci Cube of order 13). '10' means that the subscription is 10. So we are testing $XFC_{10}(11)$. '5' means that the number of faulty nodes is 5. '2' means that there are two faulty links. '1' means that packets are generated according to even distribution. '500' means that the hop time is 500ns. '60' means that the simulation runs as long as 60 seconds.

Line 4: ‘4’ means that the testing case is for Gaussian Cube. ‘11’ means that the dimension is 11. ‘3’ means that the $M = 2^3$. So we are testing *GC*(11, 8). ‘0’ means that there is one faulty node. ‘500’ means that the hop time is 500ns. ‘60’ means that the simulation runs as long as 60 seconds. Now we have only implemented having one faulty node and no faulty link (see: void CGaussianCube::BuildFault()). The faulty node is fixed as $\underbrace{00\ldots0}_{n}$, where n is the dimension of the Gaussian Cube. The program has provided two functions to add faulty nodes and faulty links respectively:

void CGaussianCube::AddFaultyNode(unsigned address)

void CGaussianCube::AddFaultyLink(unsigned address1, unsigned address2).

The only task left is to design and interface so that faulty links and over one faulty node can be added to the network.

Line 5: ‘0’ stands for the end of the input file. The user can add comments after this line and these characters will not be processed.

VIII.1.3 Output file

There are two output files:

Regular Fibonacci Cube	RegOutput.txt	RegTable.txt
Enhanced Fibonacci Cube	EnhOutput.txt	EnhTable.txt
Extended Fibonacci Cube	ExtOutput.txt	ExtTable.txt
Gaussian Cube	GaussianOutput.txt	GaussianTable.txt

Table VIII.1 output files of simulation

These files are automatically created. If they exist before running the simulation, then the results will be appended to the file. The XOutput.txt records the result for each reading of one testing case. XTable.txt records the statistical result for each testing case by processing the result of all readings. In the batch mode, this provides a succinct presentation of result.

VIII.2 FPGA implementation with Handel-C

Very detailed comment has been added to the source code of both programs. Macros are extensively used in the programs so that it is very easy to change the dimension of the network, which is controlled by a macro called Num_Bits. Some other macros also need to be modified if a new Num_Bits is used. Please refer to the comments in the source code. Equations of calculating these macros are given in detail.

A useful programming skill is using conditional compiling. This makes it possible to switch the source code between Debug mode and EDIF mode by only commenting out or releasing '#define MYDEBUG'. If this macro definition is released, then the code is for Debug mode. If it is commented out, then the code is for EDIF mode. Likewise, if '#define FINAL' is released, then the router's input and output are fixed and the router eliminates all the gates needed for controlling Flash Memory that stores testing cases and results. However, with regards to fuzzy router, rules are stored in Flash Memory, it is impossible to completely exclude gates used for controlling Flash Memory. Thus the comparison of number of gates between classical router and fuzzy router is not on a fair ground. In other words, the comparison is not based only on the complexity of logic.

In the Debug mode, the testing files are transferred to the Flash Memory of RC100 board beginning at address READ_START_ADDRESS. The results are stored in the Flash Memory starting at address WRITE_START_ADDRESS. The rules are stored from address RULE_BASE.

To generate the circuit diagram, we need to use Schematic Editor. It is a tool of Xilinx Project Manager. Choose File: \\Generate from netlist. Then choose the .edf file. The Schematic Editor will automatically generate the circuit graph. However, errors occur frequently because .edf file is generated by DK1, a product of Celoxica Ltd, while Xilinx is another company. So there are some discrepancies and small modifications on .edf is necessary for successful conversion. Some technical problems can be solved by posting them in Xilinx's forum. The circuit generated in put in the CD attached with the report.