Algorithms in fault-tolerant Clos networks

Lee, Hyunyeop, Ph.D.

New Jersey Institute of Technology, 1994

ALGORITHMS IN FAULT-TOLERANT CLOS NETWORKS

by
Hyunyeop Lee

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Electrical and Computer Engineering

October 1994

APPROVAL PAGE

ALGORITHMS IN FAULT-TOLERANT CLOS NETWORKS

Hyunyeop Lee

Dr. John D. Carpinelli, Dissertation Advisor — Date
Director of Computer Engineering
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Sotirios Ziavras, Committee Member — Date
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Michael Palis, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Frank Hwang, Committee Member — Date
Member of Technical Staff, AT&T Bell Lab., Murray Hill, NJ

Dr. Vaclav Benes, Committee Member — Date
Research Associate

BIOGRAPHICAL SKETCH

Author: Hyunyeop Lee

Degree: Doctor of Philosophy

Date: October 1994

Undergraduate and Graduate Education:

- Ph. D. in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1994
- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1990
- Bachelor of Science in Electrical Engineering,
  Yonsei University, Korea, 1980

Major: Electrical Engineering

Presentations and Publications:

Hyunyeop Lee and John D. Carpinelli "Algorithms in Fault-tolerant Clos Interconnection Networks," 1994 Conference on Information Sciences and Systems, Princeton University, NJ

This work is dedicated to
my family

ACKNOWLEDGMENT

The author wishes to express his sincere appreciation to his advisor Dr. John D. Carpinelli for his guidance and assistance throughout his research. His continued friendship, and support throughout my career as a graduate student at the New Jersey Institute of Technology is greatly appreciated. Also, I would like to thank all the committee members, Dr. Ziavras, Dr. Palis and Dr. Benes of NJIT, and Dr. Hwang of AT&T Bell Lab., for sparing their precious time for this defense, and for their valuable suggestions. Finally, the author wishes to thank his family for their support, help and encouragement throughout his doctoral studies.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

1.1.1 Parallel Processing

Many of today's scientific and industrial problems require enormous processing power, and the desire for faster computers appears boundless as the complicated applications that require the processing of enormous amount of data emerge. Multi-microprocessors are used in areas requiring one or more of the following:

- Very high computational bandwidths and/or short response times
- High system resilience and fault-tolerance capabilities
- Ability to operate under adverse environmental conditions
- Geographically distributed computing with an associated need for effective communication between centers
- Storage and retrieval of large volumes of data within a relatively short period of time
- Very close interactions between equipment and human beings

Advances in technology have achieved some increase in computing power. Integrated circuit (IC) technology replaced conventional vacuum tubes and transistors, and improved performance both in speed, size, and density. The improvements in device technology, versatile instruction sets, large addressing ranges, and operating systems also contributed to the increase in processing power. The development of microprocessor architectures, accompanied by bigger and more powerful instruction sets, has enabled the overall throughput provided by a single microprocessor to increase by more than three orders of magnitude during the past few decades. However, this development is approaching the limit where these technologies can no longer keep up with the need for more speed. To meet these problems requires deviation from the restriction of the von Neumann architecture which uses a single processor to fetch instructions from memory and execute them one at a time.

Long before the advent of microprocessor technology, designers had proposed the concept of parallel systems as a mechanism to go beyond the upper bound on performance attainable with a single processor. A single processor can fetch instructions from memory and execute them one at a time. Parallel systems, however, are based on the principle that more than one task can be performed simultaneously. An evolutionary change such as parallel computer architectures and super fast microprocessors makes parallel processing feasible. Parallel processing can be realized either at the software level or at the hardware level or at both. At the software level, parallelism is obtained by time-sharing the computer resources among different programs. Here, the operating system divides the CPU time among the different programs so that no one program monopolizes the CPU for a long time while others are waiting. This technique has been used on computers with a single processor to achieve parallelism in the form of multiprogramming, multitasking, multiuser and time-sharing capabilities.

When parallelism is implemented at the hardware level, it can take place at the computer level, at the processor level, or at the subprocessor level. One hardware strategy is the use of pipelining [12]. The concept of pipeline processing in a computer is similar to assembly lines in an industrial plant. To achieve pipelining, one must subdivide the input task (process) into a sequence of subtasks, each of which can be executed by a specialized hardware stage that operates concurrently with other stages in the pipeline. Successive tasks are streamed into the pipe and are executed in an overlapped fashion at the subtask level. The pipeline consists of a cascade of processing stages. The stages are pure combinational circuits performing arithmetic or logic operations over the data stream flowing through the pipe. The stages are separated by high-speed interface latches. According to the levels of processing, Handler has proposed the classification scheme of pipeline processors as arithmetic pipelining, instruction pipelining, and processor pipelining [6]. Vector pipelines are special form of pipelines which are specifically designed to handle vector instructions over vector operands. Computers with vector instructions are called vector processors.

Multiprocessor computers include all systems that use more than one processor to perform a desired application. The spectrum of such systems ranges from low-cost personal computers which frequently utilize a second microprocessor for decoding the key depressed on the keyboard, to powerful supercomputers and array processors which contain hundreds of processors working in parallel. These processors cooperate to execute the instructions of a program. In the ideal case, a system with n identical processors could offer n times the throughput available with a single processor. Alternatively, the additional processors can be used as backups, on an automatic basis, in case the primary processor malfunctioned.

Parallel computer systems can be grouped according to Flynn's classification [5], which is based on the number of concurrent instruction and data streams in a computer. An instruction stream is the sequence of instructions executed by a computer. The data stream is the sequence of data accessed to be processed by the instructions. Flynn defines the four classes as

- SISD (single instruction single data stream)
- SIMD (single instruction multiple data stream)
- MISD (multiple instruction single data stream)

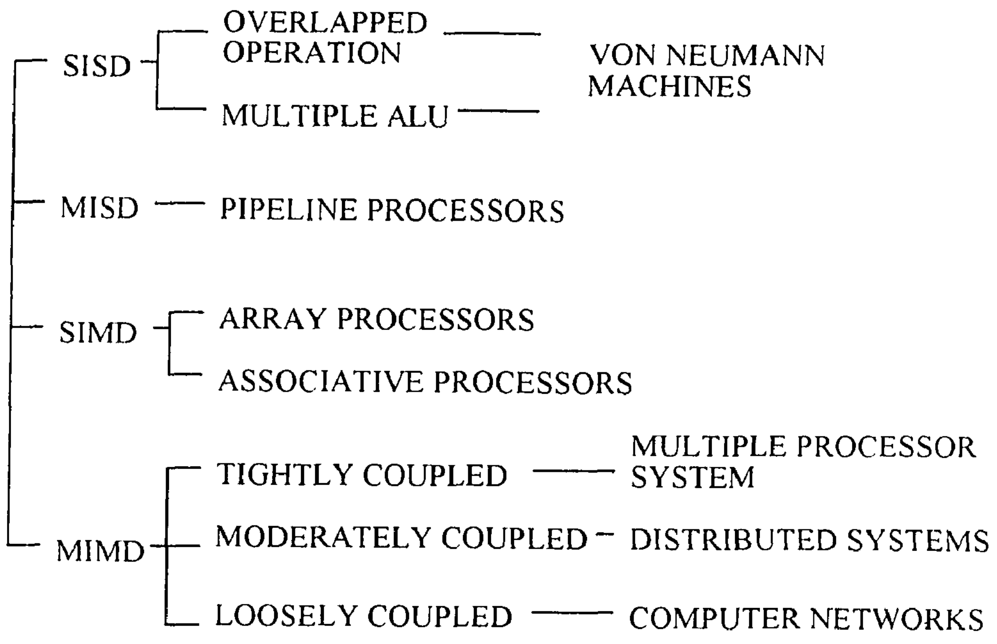

Figure 1.1 Flynn's classification of multicomputers

- MIMD (multiple instruction multiple data stream)

Figure 1.1 shows Flynn's classification of parallel computers.

An array computer [14] is a synchronous array of parallel processors which consists of many processing elements under the supervision of one control unit. An array processor can handle single instructions and multiple data streams (SIMD). Each processing element (PE) consists of a processor with a local memory. Because of its large numbers of PEs, the array computer is suitable for applications in image processing, matrix manipulation, parallel sorting, and fast Fourier transform.

Another form of parallel processing is distributed processing, which is also called "computer networking". A computer network is a multicomputer arrangement where the computers communicate via special processor-to-processor data links. This is a looser coupling than the shared memory communication of multiprocessing systems. A network can link computers hundreds of miles or just a few feet apart.

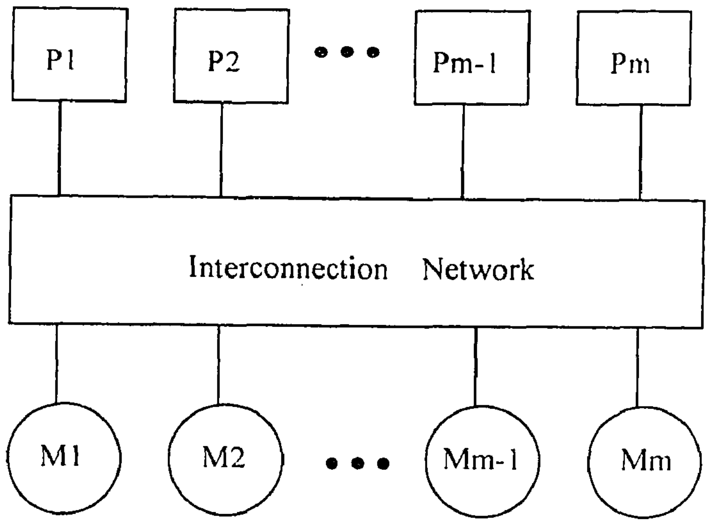

Figure 1.2 Multiprocessor system

Short-distance networks, perhaps contained in one building, are referred to as "local" networks. Here the computation load is distributed among more than one computer. Communications between the different computers take place in the form of passing messages to obtain data or exchange results. The advantage of the distributed computing system include fast response, high availability, fault tolerance, resource sharing, high adaptability to the changes in the work load, and better expandability. These advantages have been enhanced by the availability of low-cost microprocessors and data link interfaces produced by LSI circuit techniques.

1.1.2 Interconnection Networks in Multiprocessor Systems

Clearly, using many processors in the same system yields more speed than using one processor. Recent advances in VLSI technologies, coupled with the need for fast computers, have made large-scale multiprocessor systems economically feasible. In such systems, hundreds or even thousands of processors are used to carry out the computations of a program concurrently, thereby speeding up the execution of

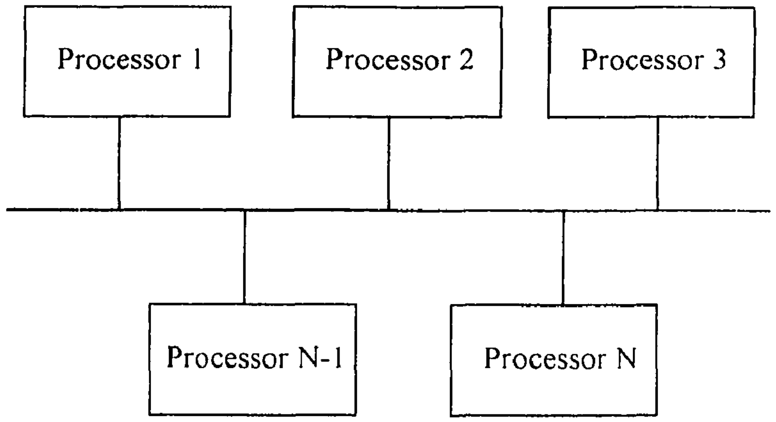

Figure 1.3 A shared bus system

the program. Many applications can benefit from this enormous computing power. The basic architecture of a multiprocessor system is shown in Figure 1.2. In this configuration, the N processors carry out computations on data stored in the M memory modules. For the interaction between the processors and memory, there must be a communications mechanism to enable any processor to access any memory module in the shortest possible time. This communication channel is denoted as the interconnection network which plays important roles in multiprocessor systems.

Interconnection networks were first proposed for use in telephone exchanges to allow subscribers to talk with one another. Some decades later, researchers began to consider how networks could be incorporated into computers. Many different approaches have been considered and some implemented. These include the use of buses, hierarchies of buses, direct links, single stage networks, multistage networks and crossbars. The shared bus is shown in Figure 1.3. When several processors are connected together via a bus, these processors should be capable of communicating with each other. It is obvious that, as the number of processors increases, the load on the interface increases sharply. If one provides a different bus for each path, the cost of such multiple-bus connections increases as the square of the number of processors. On the other hand, if only one bus is used, the contention problem between different messages may become critical. With more processors/memories, the bus becomes a performance bottleneck. Most designers opt for multiple-bus solutions. The resulting network is named on the basis of its geometry as a star, a cube, a hypercube, a hypertree, a cluster, and by other similar self-explanatory names. In all of these cases, a few pairs of resources have direct links with each other, but other pairs must communicate via one or more intermediate nodes, thus introducing time delays and performance degradation. In order to reduce the load on the bus, it is now becoming common for individual processors to have cache memories.

The next simplest form of interconnection mechanism is the crossbar [17]. In a crossbar switch, every input port can be connected to a free output port without blocking. This is simple, but impractical as the number of processors increases. A more practical method is the use of multistage interconnection networks (MINs) which consist of small-sized crossbars and links between them in a way unique for each MIN. Usually, a multistage network consists of more than one stage of switching elements and is capable of connecting an arbitrary input terminal to an arbitrary output terminal. These can be divided into three classes: blocking, rearrangeable, and nonblocking. In blocking networks, simultaneous connections of more than one terminal pair may result in conflicts in the use of network communication links. Examples of this type of network include the data manipulator [24], baseline [17], SW banyan [23], omega [17], flip [25], and delta [28] networks. A network is called a rearrangeable network if it can realize all possible connections between its inputs and outputs by rearranging its existing connections so that a connection path for a new input-output pair can always be established. A well-defined network, the Benes network, belongs to this class. A network which can handle all possible connections without blocking is called a nonblocking network; some varieties of the Clos network are in this class.

As systems become more complex, the reliability of the system has become a major concern because just one fault in the system can degrade system performance or cause the system to fail completely. The function of fault-tolerance is to preserve the delivery of expected system services in the presence of errors. There are two major aspects to fault tolerance: (1) detecting and diagnosing faults; and (2) avoiding known faults if such a capability exists. Techniques such as test patterns, dynamic parity checking, and write/read-back/verify can be used in various interconnection networks for detecting and diagnosing fault tolerance. In order to achieve fault tolerance, the topology of the network can be modified, usually by adding spare links and switches. Other method involve error-correcting codes, bit-slice implementation with spare bit slices, and duplicating the entire network [57]. Many of the known interconnection network can be made fault tolerant. Some of the examples are the Extra Stage Cubes (ESC) [56], the multipath omega network [59], the F-network [63], the enhanced IADM network [30], the merged delta [28], the extra stage gamma network [58], the $\beta-$network [66] and the INDRA network [61]. The fault tolerant Clos network (FTC) has been proposed by Nassar [60]. Little about the properties and routings of fault tolerance of the Clos networks is available in the literature.

1.2 Background

Interconnection networks have been widely studied since they play important roles in telephone switching networks and other communication, data networks and computing systems. In multiprocessor systems, they are needed as a means of interprocessor communications. The three-stage Clos network [31] served as a basis for the Benes network [32] and the Waksman network [35]. Later, other networks such as the omega network [29], the baseline network [17] and the cube-connected network [26], were proposed in order to simplify the Benes network.

Several control algorithms for Clos networks have been proposed. The earlier algorithms were based mostly on matrix decomposition methods. Neiman [33] has proposed an $O(n^2k^2)$ algorithm which consists of two phases: a relatively simple preparatory phase, followed by a complex iteration phase. Here, n represents the number of switches in the second stage, and k the number of switches in the outer stage of the Clos network. Tsao-Wu [34] has presented two modifications to the preparatory phase, which result in lowering the probability that the second phase will be needed. However, this algorithm does not lower the worst case complexity of Neiman's algorithm. Waksman introduce another new algorithm [35], and Opferman and Tsao-Wu suggested the Looping algorithm for the Benes network [32]. A different algorithm has been proposed by Ramanujam [36]. However, Kubale [37] showed that Ramanujam's algorithm fails for some permutations. Also, the matrix decomposition algorithm suggested by Jajszczyk [38] has been proved to fail by Cardot [39]. These algorithms select elements from the matrix according to certain rules, and backtrack when they are unable to obtain a permutation matrix. These rules are rather intuitive and do not work in some cases.

Many algorithms are based on the minimum edge coloring on a bipartite multigraph. Hwang [40] suggested that edge coloring algorithms for bipartite graphs can be adapted to decompose Clos networks. Vizing's method uses $O(n^2k)$ time to perform a complete coloring since it needs $O(k)$ time to find the alternate path to color an edge. The Euler partitioning approach to edge coloring uses a divide-and-conquer technique and was formalized by Gabow and Kariv [46], whose algorithm runs in time $O(nk^{\frac{3}{2}} \lg\ k)$. A modified version of the previous algorithm was presented by Gabow [45], and it runs in time $O(nk \lg\ k)$. Cole and Hofcroft [47] also proposed an algorithm by preprocessing the edges while keeping the degree of a vertex invariant. Lev, Pippenger and Valiant [52] developed parallel edge coloring algorithms for routing on Clos networks. Recently, Gordon [43] introduced an algorithm which runs in time $O(nk^{3/2})$ with the aid of specification and count matrices. Chiu [44] demonstrated that Gordon's algorithm displays errors for some permutations.

Parallel algorithms were proposed by Nassimi and Sahni [49]. The time bounds of these algorithms may be reduced if all of the switch sizes are integral powers of two. Another parallel algorithm was proposed by Carpinelli [50] which eliminates backtracking by introducing the concept of partitioning. The Benes network control algorithm for frequently used permutations was reported by Lenfant [53].

The self-routability of Clos networks has been studied by Douglass and Oruc [54]. This study shows that the Clos network is self-routing if and only if $N/m \leq 2$ or $m = 1$. Raghavendra [55] also reported self-routing algorithms in Benes and shuffle-exchange networks.

Meanwhile, a great deal of effort has been directed to the fault tolerant multistage interconnection network in order to make the network more reliable and fault tolerant. A single fault in the interconnection network can cause a severe degradation in performance unless measures are provided to make the network tolerant to such faults. With developments in VLSI technology, large scale multiprocessor systems with fault-tolerant interconnection networks have become feasible. Many fault tolerant interconnection networks have been proposed. However, few fault tolerant Clos networks have been studied until Nassar [60], who provided alternate paths by adding multiplexers and switches to the network.

Although several control algorithms have been proposed in order to reduce the run time in the Clos interconnection network, little effort has been made in improving the performance by extending the algorithm to the fault-tolerant cases. Nassar's control algorithm [60] for the fault tolerant Clos network is based on Neiman's algorithm, and has the same time complexity as Neiman's algorithm. Considering that the spare switches can provide alternative paths in the system, his algorithm could have been faster if he could utilize these paths during the routing process. The extra switches in the fault-tolerant Clos (FTC) network have been found to give great flexibility to the routing algorithms by providing alternative paths to the system, and thus can be used to improve the run time significantly when the system displays few or no faults. No studies have been made so far about the utilization of the extra spare switches for the improvement of routing speeds in the fault tolerant Clos network.

In this thesis, Gordon's algorithm is shown to display errors in some permutations. Then, the new simple algorithm which works for all permutations for the control of rearrangeable Clos networks is proposed, which is based on his algorithm. The new algorithm is extended to the fault tolerant Clos (FTC) network. The extra switches in the fault-tolerant Clos (FTC) network are used to improve the run time significantly since they provide alternative paths to the system when the system displays few or no faults. The effect of increasing the number of extra switches on system routing time, reliability and cost in fault tolerant Clos networks is analyzed. Finally, the optimum number of extra switches on the fault tolerant Clos network is determined which will best satisfy the run time, reliability and cost constraints.

1.3 Outline

This research has demonstrated that Gordon's algorithm displays errors for some permutations. Next, a new algorithm is proposed for the Clos networks which is based on Gordon's algorithm. This algorithm is extended to the FTC networks, and resulting run times are compared with the ordinary networks. The FTC network has been classified into three types for the purpose of developing algorithms systematically. The reliabilities for these networks are examined, and the optimum number of extra switches which satisfies the reliability, run time and cost constraints is considered.

The rest of the thesis is organized as follows. In chapter 2, basic concepts and relevant notation which will be used in the thesis are introduced. These include the representation of interconnection networks, fault tolerance, and reliability of the system. In chapter 3, the implementation of important MINs such as the crossbar network, Clos network and Benes network are examined. Routing algorithms based on the matrix decomposition, edge coloring and matching, and parallel decomposition are discussed in Chapter 4. Next, Gordon's algorithm is examined and then a counter example is given which demonstrates that his algorithm has a flaw. A new algorithm for routing on ordinary Clos networks and three kinds of swaps used in the algorithm are introduced in chapter 5. In chapter 6, some of the fault tolerant multistage interconnection networks, such as Extra Stage Cube (ESC) and Fault Tolerant Clos (FTC) networks, are addressed. In chapter 7, three types of FTC network are discussed, and swapping rules and conditions in each case are considered. A new algorithm for the FTC network is proposed, which is extended from the algorithm illustrated in chapter 5. Reconfiguration of the FTC network is considered next. In chapter 8, reliabilities of the fault tolerant Clos network are considered and corresponding space complexities are examined. Also, the fault detection and location of the FTC network is considered. Finally, conclusions and open problems are presented in Chapter 9.

CHAPTER 2

MODELLING OF INTERCONNECTION NETWORK

2.1 Introduction

The modelling of interconnection networks is important in order to analyze them. In this chapter, the concept of permutations as well as basic definitions and notations that are used in interconnection networks are introduced in section 2.2. These provide a basis for representing interconnection networks in the various matrix forms by setting each of the stages of the network. Section 2.3 introduces bipartite multi-graphs, which is another method of representing interconnection networks. These can be used to route a permutation for Clos network in edge coloring algorithms, as will be shown in chapter 4. The concept of fault tolerance is described in section 2.4, followed by the concept of reliability in section 2.5. These will be used to describe the fault tolerant Clos network, which will be discussed in chapter 5.

2.2 Interconnetion Networks

2.2.1 Representation

A set is a collection of distinct elements. A mapping or a function from a set A into a set B is a rule which assigns to each element a of A exactly one element b of B. It is written as $b{=}(a)f$ to imply that a is mapped to b by f. Let f be a mapping of A into B. It is said to be one-to-one if, whenever $a_1 \neq a_2$, $(a_1)f \neq (a_2)f$ and it is said to be onto if for each $b \in B$, there exists $a \in A$ such that $(a)f = b$. Let f be a mapping of set A into set B and let g be a mapping of set B into set C. The mapping $f \cdot g$, defined by $(a)f \cdot g = ((a)f)g = (b)g = c$, $a \in A, b \in B, c \in C$, is called the composition of f and g. A permutation of a set S is a one-to-one mapping of S onto itself. It is written as $(x)P = y$ to imply that x is mapped onto y by permutation P. Both x and y belong to S.

A group is a set G with a binary operation dot$(\cdot)$ on G, where the binary operation is associative, there is an identity element e in G such that $\epsilon \cdot x = x \cdot \epsilon = x$ for all x in G, and for each x in G, there is an inverse element x' in G with the property that $x' \cdot x = x \cdot x' = \epsilon$. A subgroup of a group G is a subset of G which also forms a group with respect to the group operation of G. The set of all permutations of N elements on S form the symmetric group, denoted as $\sum_N$. The cardinality of $\sum_N$ is $N!$.

Two notations are used for representing permutation P. In standard notation, also called two-row matrix form, there are two rows of elements; the first row contains the source elements to be permuted and the second row contains the destination elements that they are mapped onto. It is written as $P = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \end{pmatrix}$ to imply that $(x_i)P = y_i$, $1 \leq i \leq n$, where $S = \{x_1, x_2, \cdots, x_n\} = \{y_1, y_2, \cdots, y_n\}$. For example, $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix}$ is a permutation which maps 1 to 2, 2 to 4, 3 to itself and 4 to 1, where $S = \{1, 2, 3, 4\}$. In cyclic notation, the permutation is of the form $(x_1, x_2, \cdots, x_n)$ where x_1 is mapped onto x_2, x_2 is mapped onto x_3, and so on. The final element x_n is mapped onto the first element x_1. It is written as $P = (x_1, x_2, \cdots, x_n)$ to imply that $(x_1)P = x_2$, $(x_2)P = x_3, \cdots$, $(x_n)P = x_1$, where $x_i \in S$, $S = \{x_1, x_2, \cdots, x_n\}$. The previous example $P = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix}$ will be represented by the cycle (1 2 4). Any element which is mapped onto itself is not written explicitly, so 3 is not included in the cycle (1 2 4). Particularly, the permutation ϵ is called the identity permutation, and $(x)\epsilon = x$ for all $x \in S$. For example, the permutation $\epsilon = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix}$ maps every element onto itself.

Given a switch with N inputs and N outputs, the setting can be expressed as an $N \times N$ matrix, K. The rows of the matrix represent the inputs of the switch and the columns represent its outputs. $K[i,j] = 1$ if the switch is set so that input i is connected to output j; otherwise, $K[i,j] = 0$.

The K matrix can be used to represent a stage. A stage is a set of switches which are disjoint, that is, there is no possible connection from the output of one switch to the input of another in the set. Notice that the permutations for the two 2×2 switches were combined to form one permutation encompassing four elements. The matrix approach is similar; the 2×2 matrices corresponding to the switch settings are embedded in a 4×4 matrix which defines the setting of the entire stage. Given the settings of two switches

$$K_{12} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad K_{34} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

the 4×4 matrix which results from their embedding is

$$K = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In the permutation notation, 1 and 2 had to be mapped amongst themselves, as did 3 and 4. This is because the switches are disjoint, and the input of one switch cannot be mapped onto the output of the other. In the matrix this is accomplished by setting the elements of the quadrants not on the main diagonal to zero. In this example, rows 1 and 2 cannot have non-zero entries in columns 3 and 4. Figure 2.1 shows the resultant switch settings.

A multistage permutation network consists of several stages of switches. The outputs of one stage serve as the inputs to the next stage. The mapping realized by the network is derived from the mappings of the individual stages. If the maps of each stage are represented as matrices, the matrix representing the map of the entire network is the ordered product of the stage matrices. As an example, consider a three-stage network as shown in Figure 2.2. The matrices corresponding to the stage settings are, respectively,

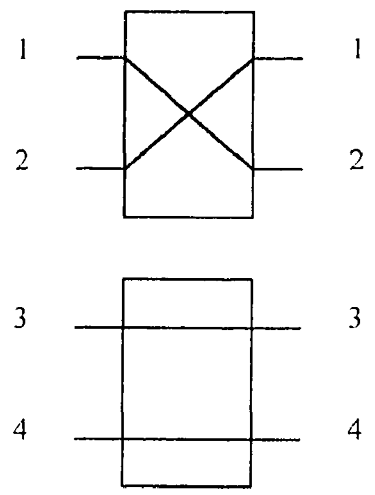

Figure 2.1 Switch settings of two 2×2 switches

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Their ordered product is

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

The matrix of this kind becomes very sparse as the number of inputs increases. For N inputs, a matrix of this form has N^2 entries. In order to reduce this size, a compacted matrix is often used. The $N \times N$ matrix is consolidated into a $k \times k$ matrix, H_m, where $m = N/k$. The first row of H_m is the sum of the first m rows of the original matrix; the second row of H_m is the sum of the second m rows, and so on. The columns are compacted in a similar manner. The 2×2 matrix, H_2, corresponding to the ordered product matrix derived above is

$$\begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix}$$

Note that the sum of the elements in each row and in each column is exactly m. There is a trade-off that results from the savings in matrix size; one compacted matrix may represent more than one mapping. The compacted matrix $\begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix}$ may represent any of the following matrices.

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

When dealing with compacted matrices, additional information is required to distinguish between mappings. The matrix H_m is the representation used by the matrix decomposition class of algorithms for routing on Clos networks. The H_m matrix is a typical way of representing the permutation. However, as the matrix becomes sparse, further reduction in size can be possible with the use of S and C matrices, which are called the *specification* matrix, and the *count* matrix. In the S matrix, the $k \times k$ matrix H_m is consolidated into a $k \times n$ matrix where n is the sum of the elements in any row or column of H_m. This is especially useful in representing the Clos network as is explained in chapter 5. In order to obtain the S matrix, let $P = \begin{pmatrix} 0 & 1 & \cdots & i & \cdots & N-1 \\ y_0 & y_1 & \cdots & y_i & \cdots & y_{N-1} \end{pmatrix}$, where $0 \leq i \leq N-1$ and $N = nk$. For each signal i, calculate x and t where $x = \lfloor i/n \rfloor$ is the first-stage input switch at which the signal arrives, and $t = \lfloor y_i/n \rfloor$ is the last-stage output switch to which it should be routed, and set any unassigned element which is the next unassigned element in the xth row of S to t. On the other hand, each element of C, $c[x, y]$, $0 \leq x \leq k-1$, $0 \leq y \leq n-1$, is the number of occurrences of the integer x in column y of S. As an example, the permutation matrix is given as

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 10 & 3 & 5 & 6 & 11 & 7 & 1 & 9 & 4 & 0 & 8 \end{pmatrix}$$

The H_3 matrix is

$$H = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

The S and C matrices are

$$S = \begin{bmatrix} 0 & 3 & 1 \\ 1 & 2 & 3 \\ 2 & 0 & 3 \\ 1 & 0 & 2 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

A multistage permutation network consists of several stages of switches. The output of one stage must be connected to the inputs of the next stage. The permutation realized by a multistage network is the ordered composition of the permutations realized by its stages. Consider a 4-input, 3-stage network. The first stage realizes the permutation $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix}$, the second stage realized the permutation $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix}$, and the third stage the permutation $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 3 & 2 & 4 \end{pmatrix}$. Composing stage permutations in order, the first permutation maps input 1 onto output 2. This output 2 of the first stage is assigned to input 2 of the second stage; the second stage permutation map routes this to output 4 of the second stage. Finally, input 4 of the third stage is routed to output 4, so the network routes input 1 to output 4. Repeating this for the other inputs, the permutation realized by entire network is $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 3 & 1 \end{pmatrix}$, as shown in Figure 2.2.

2.3 Bipartite Multigraphs

The bipartite multigraph also can be used to represent a permutation for Clos networks, which will be introduced in chapter 3. A graph $G = (V, E)$ is an ordered pair of finite sets V and E. The elements of V are called *vertices*, and the elements of E are called *edges*. An edge (v, w) is an unordered set of two distinct vertices. If an edge (v, w) can occur more than once, G is a *multigraph*. Edge (v, w) is incident to v and to w, and vertices v and w are adjacent. A *subgraph* of G is a graph whose vertices and edges are in G. To delete edge e from G means to form the subgraph $G - e$, consisting of all vertices of G and all edges of G except e. To delete vertex v from G means to form the subgraph $G - v$, consisting of all vertices of G except v.

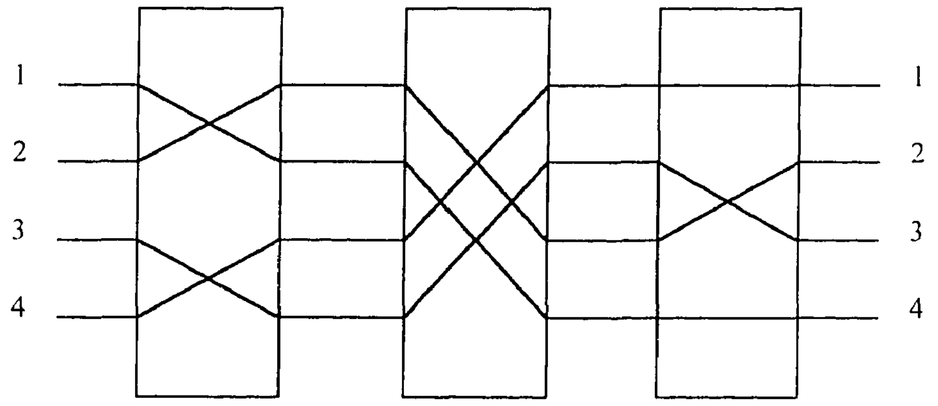

Figure 2.2 Switch settings of three stage network

and all edges of G except those incident to v. A graph corresponding to a function has the property that the vertex set can be partitioned into two disjoint subsets R and D (R corresponds to the set of range vertices and D to the set of domain vertices) such that all edges in the graph join a vertex in D to one in R. There are no edges that join two vertices in R or two vertices in D.

A graph whose vertex set can be partitioned in this way is called a *bipartite graph*. All graphs that correspond to functions are bipartite. The degree of a vertex v is the number of edges incident to v. An example of a bipartite multigraph is shown in Figure 2.3. A graph is *regular* if all vertices have the same degree. A *path* P is a sequence of edges (v_1, v_2), (v_2, v_3) . . . . (v_{n-1}, v_n). The ends of P are vertices v_1 and v_n . If $v_1 \neq v_n$, P is open, otherwise P is closed. A graph is connected if there is a path between any two distinct vertices. A connected component of a graph is a maximal connected subgraph. A *matching* M of G is a set of edges, no two of which are incident to the same vertex; M covers any vertex incident to an edge in M. An edge coloring of G is an assignment of a color to each edge in G such that

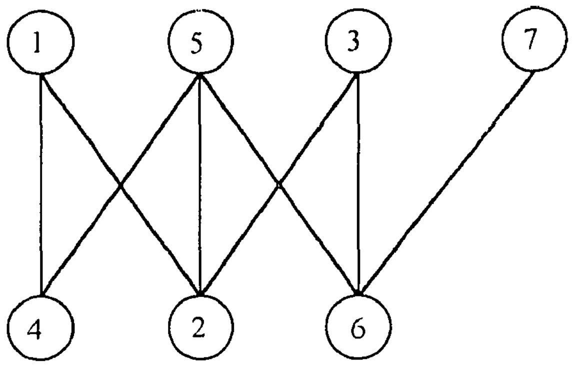

Figure 2.3 A bipartite multigraph

no edges incident to a vertex have the same color. Thus all edges of a given color form a matching. A minimal edge coloring uses the fewest number of colors possible. The application of coloring and matching to routing Clos network will be discussed in Chapter 4.

2.4 Fault Tolerance

A fault tolerant MIN is one that provides service even when it contains a faulty component or components. A fault can be either permanent or transient. Fault tolerance is defined only with respect to a chosen fault tolerance model, which has two parts. The fault model characterizes all faults assumed to occur, stating the failure modes for each network component. The fault tolerance criterion is the condition that must be met for the network to be said to have tolerated a given fault or faults. The fault model is the type of faults that can occur in the network. Implicitly, the fault model specifies the type of faults that can be recovered from using the proposed fault tolerance design. Different designs specify different fault models. A good design, however, is one whose fault model includes as many fault types as possible. To illustrate, a typical fault model is as follows.

- Any network component can fail: MINs are made up of two types of components which are switches and links.
- Switches and links are likely to fail.
- The network is capable of recovering from any such fault.
- A link fails if it is open or short circuited. A switch fails due to some internal malfunction.

The extra hardware added to provide fault tolerance to the network fails at a lower rate than the network hardware. This assumption is usually made for two reasons. First, if the extra hardware added to the network to make it fault tolerant could be assumed to fail at any significant rate, then it would not be possible to propose any fault tolerance design. In addition, this assumption can be justified for MINs because these components usually remain idle under normal conditions. Thus they can be expected to have higher lifetime than the actively working components of the network.

The fault tolerance criterion is the condition that must be met in order for the system to be called fault tolerant. The fault tolerance criterion for the networks is mainly full-access retention. That is, after a fault occurs, each processor must still be able to communicate with any memory module. However, the two fault tolerant designs can offer a higher criterion, i.e., full recovery. Full recovery is the ability of the network to regain its pre-fault connectivity after a fault occurs. A network is single-fault tolerant if it can function as specified by its fault tolerance criterion despite any single fault conforming to its fault model. Generally, if any set of i faults can be tolerated, then a network is i-fault tolerant. A network that can tolerate some instances of i faults is i-robust although not i-fault tolerant. Many fault tolerant systems require fault diagnosis such as fault detection and location to achieve their fault tolerance. Techniques such as test patterns, dynamic parity checking, and write/read-back/verify can be used in various interconnection networks.

Fault tolerance can be achieved at various level in a system. Techniques for fault tolerant design can be categorized by whether they involve modifying the topology of the system. Three well known methods that do not modify topology are error-correcting codes [64], bit-slice implementation with spare bit slices [63], and duplicating an entire network [65].

2.5 Reliability

2.5.1 Fundamentals

The reliability of a system is defined as the probability that the system will perform a required function under stated condition for a stated period of time t. Mathematically, the reliability, R, of a system is a function of λ and t, where λ is a constant representing the failure rate (per unit time). To simplify the analysis in this thesis, the time factor will be only implicit. In other words, when it is said that the reliability of a switch is r, it will mean the reliability of the switch over a given period of time t. This is done because the focus will be on comparing reliabilities, rather than obtaining the absolute reliability value. In comparing two networks, for instance, the two networks should be under the same circumstances, including the period of time, t, hence the omission of the time factor. Predicting reliabilities usually involves dealing with probabilities. It stands to reason then that an overview of probability theory should be given before discussing the fundamentals of reliability.

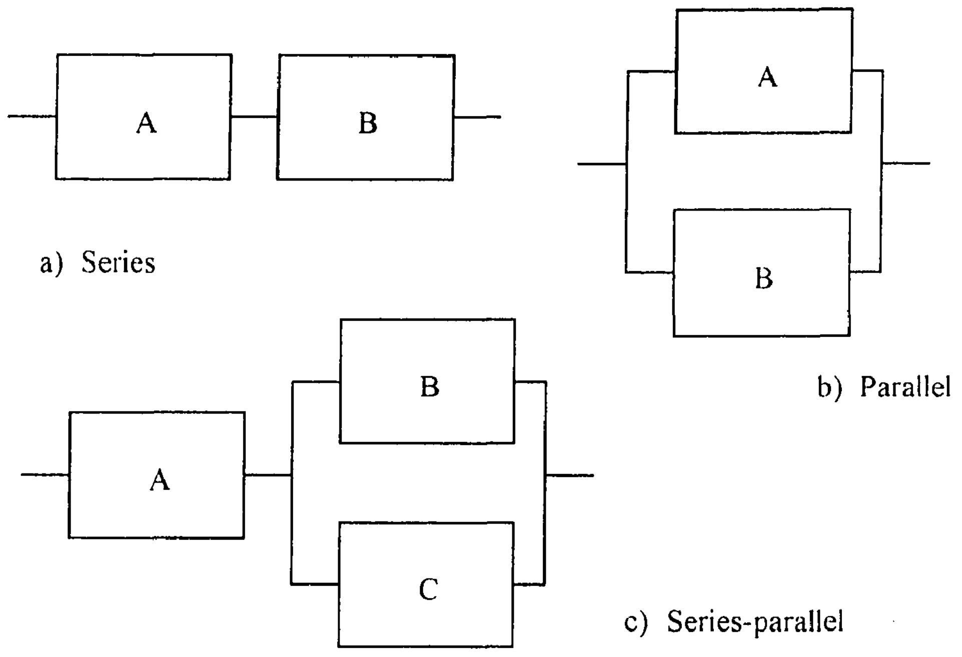

Figure 2.4 Series, parallel, and series-parallel systems

2.5.2 System Reliability

Simple systems can generally be classified into three categories as shown in Figure 2.4: series, parallel, and series-parallel. A system can be broken down into isolated components. First, a series system is defined as a complex system of independent units connected together, or interrelated, in such a way that the entire system will fail if any one of the units fails. It is assumed that the failure of one component has no effect on the probability of any other component failing. Thus, the system can be no better than its weakest component. Series reliability is calculated using the product rule as

$$R_s = \prod_{i=1}^{n} P_i$$

where P_i is the probability that a component i of the system will function properly. When component reliabilities are equal, the reliability of the system is

$$R_s = (P_i)^n$$

The unreliability of the system is defined as 1−reliability. Thus, the unreliability of a system is

$$U_s = 1 - R_s = 1 - \prod_{i=1}^{n} P_i$$

On the other hand, a parallel system is defined as a set of interrelated components connected in such a way that a redundant, or standby part can take over the function of a failed part to save the system. Redundancy refers to the use of more than one part for the same function. The calculations for parallel reliability are more complex than those for series reliability and include the concept of unreliability. The parallel reliability of a system is

$$R_p = 1 - \prod_{i=1}^{n} U_i$$

For equivalent component unreliability,

$$R_p = 1 - (U_i)^n = 1 - (1 - P_i)^n$$

Parallel reliability increases as the number of components increases, which is the opposite of the series systems. Parallel systems also display marginal probability, which refers to the increase in reliability as components are added. As the redundancy is increased in parallel systems, it is important to balance the costs involved.

Mixing the two kinds of systems, there can be a series-parallel system which includes both series and parallel components. Reliability for these systems can be determined by computing the reliabilities separately, using the rules that apply to either series or parallel systems, until the entire system is completed.

Sometimes a system has n parallel components but needs at least m of them to remain operational. This problem is a binomial distribution. The reliability of the system in this case can better be expressed as unity minus the probability of the complementary event (that is, failure occurring from having between 0 and $m - 1$

operational components). The operational components are indistinguishable from each other, and so are the non-operational components. Recall that the way to count the number of ways these components can be arranged together is a combination problem. Thus the reliability of the system is

$$R = 1 - \sum_{i=0}^{m-1} \binom{n}{i} P^i (1-P)^{n-i}$$

This equation is used in chapter 8 to obtain the reliability of the fault-tolerant Clos network.

CHAPTER 3

IMPLEMENTATIONS OF MINS

3.1 Introduction

The overall performance of multiprocessor configurations is affected by the number and the type of processors, the communication mechanism between the computing sources, the characteristics of the computational workload, and the control program. Whereas the major constraint in uniprocessor systems is the speed of the processor, the critical factor in multiprocessor systems is the speed of the interconnection mechanism. The performance of the interconnection mechanism, on the other hand, is determined by network structures and routing algorithms. A broad spectrum of networks has been studied ranging from simple linear arrays to the completely connected situation, with all other configurations falling in between. In many application, the choice of an appropriate interconnection network is a key issue in the design of any system with multiple processing resources. Nonblocking networks which work for all permutations are particularly well suited in these purposes. Rearrangeable nonblocking networks and their routing methods are also studied for their potential uses.

In this chapter, design factors for interconnection networks are discussed in section 3.2. In next two sections, the fully-connected and crossbar networks, which are the most straightforward in design, are examined. In section 3.5, the construction of the Clos network capable of mapping its N input terminal to its N output terminal is described. Finally, the Benes network is discussed in section 3.6, followed by discussion in section 3.7.

3.2 Design Factors of Interconnection Networks

There are fundamental decisions in determining the appropriate architecture of an interconnection network. The decisions are the operation mode, control strategy, switching method, and network topology [17]. Among the four decisions, network topology is a key factor in determining a suitable architectural structure. A network can be depicted by a graph in which nodes represent switching points and edges represent communication links. The topologies of interconnection networks tend to be regular and can be classified into the following two categories: static networks and dynamic networks. In a static network, links between two processors are passive and dedicated buses that cannot be reconfigured for direct connections to other processors. Topologies in the static category can be classified according to the dimensions required for layout, for example, one-dimensional, two-dimensional, three-dimensional and hypercube. In a dynamic network, links can be reconfigured by setting the network's active switching elements.

There are three topological classes in the dynamic network: single-stage, multistage, and crossbar. A single-stage network is composed of a stage of switching elements cascaded to a link connection pattern. The shuffle-exchange network is a single-stage network based on a perfect-shuffle connection cascaded to a stage of switching elements. A multistage network consists of more than one stage of switching elements and is usually capable of connecting an arbitrary input terminal to an arbitrary output terminal. Multistage networks can be one-sided or two-sided. The one-sided networks have input-output ports on the same side. The two-sided networks have separate input and output sides.

The control-setting function can be managed by a centralized controller or by the individual switching element. The latter strategy is called distributed control and the first strategy is called centralized control. Generally, the centralized control is simple, but takes a longer time. In contrast, the distributed control is fast but

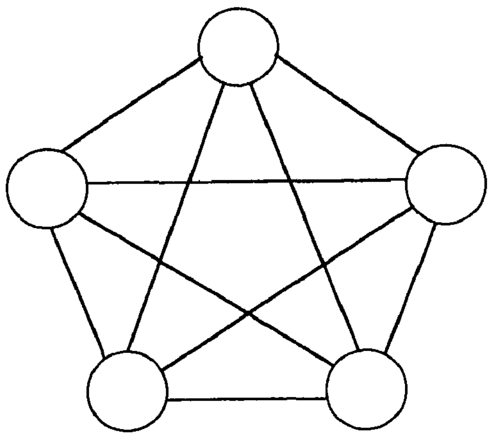

Figure 3.1 The completely connected network

requires additional computing sources in each switch. The typical operation modes of interconnection networks can be classified into three categories: synchronous, asynchronous, and combined. Also, three switching methodologies can be identified as circuit switching, packet switching, and integrated switching, which are not covered in this thesis.

3.3 Completely Connected Network

The ideal situation would be to link directly each processor to every other processor so that the system is completely connected as shown in Figure 3.1. Unfortunately, this is highly impractical for large N because it requires $N-1$ connections for each processor, and the total number of connections needed in the network would reach $N(N-1)$. For example, if $N = 2^9$, then $2^9(2^9-1) = 261,632$ links would be needed.

3.4 Crossbar Network

The simplest connection network is the Crossbar network, which has one switch for each possible input-output connection. Given N inputs and N outputs, a crossbar

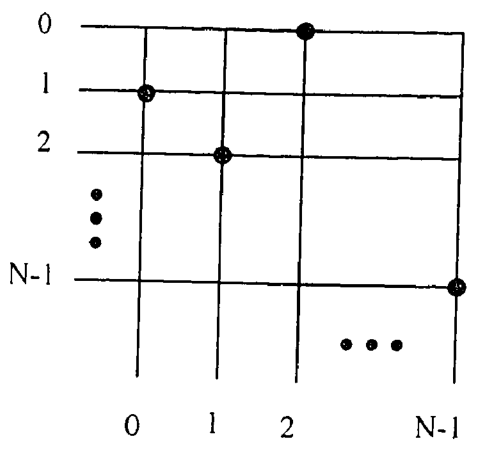

Figure 3.2 The $N \times N$ crossbar interconnection network

network would have N^2 switches and $O(N^2)$ area. The routing algorithm to set the switches is trivial. The $N \times N$ Crossbar network is shown in Figure 3.2. All Crossbar networks are strictly non-blocking. The difficulty with crossbar networks is that the cost of the network or the number of crosspoint switches which grows with N^2. This makes the crossbar network infeasible for large systems.

3.5 Clos Interconnection Networks

The interconnection networks shown above are impractical as the number of inputs increases. Many other networks are reported in the literature. Most of them are blocking networks which can not implement all the permutations. Rearrangeable nonblocking networks such as the Clos network and cellular networks are networks without blocking properties. The three-stage Clos interconnection network, which is illustrated in this section, is shown in Figure 3.3.

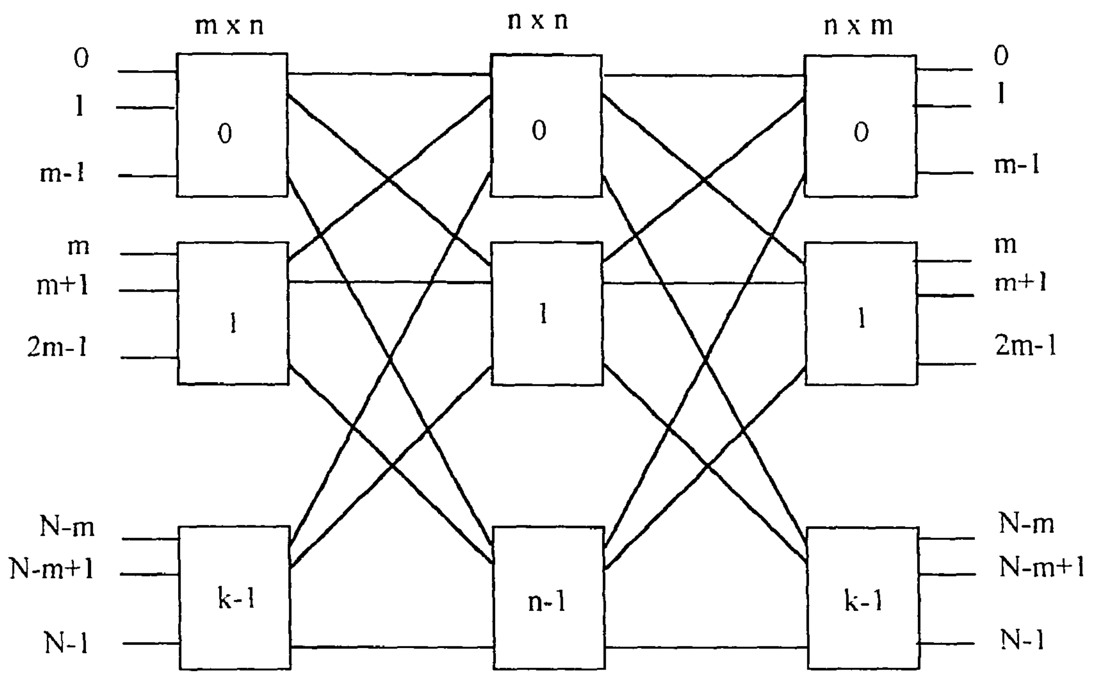

Figure 3.3 The three-stage Clos network

3.5.1 Network Structures

The three-stage Clos network [31] consists of two symmetrical outer stages of rectangular switches, with an inner stage of square switches. It is completely determined by the integer parameters n, m,and k that give the switch dimensions. The first stage contains k switches, each of which has m inputs and n outputs. Each switch is actually a simple crossbar switch which can realize any mapping of its inputs onto its outputs on a one-to-one basis. The second stage consists of n $k \times k$ switches, each of which receives exactly one input from each first-stage switch. The output stage has k $n \times m$ switches, each of which receives exactly one input from each second stage switch. The number of inputs to the network is $N = mk$. Inputs and outputs to the first-stage switch or third-stage switch i are numbered from $(i - 1)\ m + 1$ to im, $1 \leq i \leq k$. The Clos network can reduce the area of the crossbar switches for the same number of inputs. For example, when $N = 12$ with $n = m = 3$ and $k = 4$, the number of cross points in the crossbar is $12^2 = 144$, while in the Clos network, total number of cross points is $2 \times 4 \times 3^2 + 3 \times 4^2 = 120$. The Clos network is much easier to visualize when it is illustrated in three dimensions as shown in Figure 3.4.

3.5.2 Properties of the Clos Networks

In contrast to most other interconnection networks, the Clos network satisfies some important characteristics. One of the properties of the Clos network is the rearrangeability if the network satisfies the condition $n \geq m$. The interconnection network is *rearrangeable* if it can connect any idle input to any idle output by possible rearrangement of its existing paths. If the network satisfies $m = n = k$, then at most $n - 1$ existing calls need be moved in the Clos network in order to connect an idle input-output pair. Also, Clos showed that for $m \geq 2n - 1$, the network is nonblocking in the strict sense [31]. The network is *strictly nonblocking* if it is always

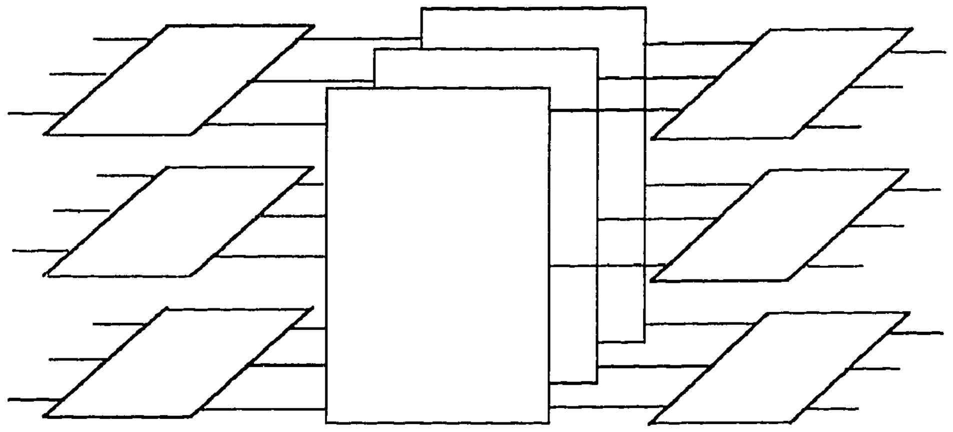

Figure 3.4 The three dimensional Clos interconnection network.

possible to connect together an idle pair of input-outputs without disturbing the routing already established, no matter in what state the network may be. Note here that the network is *nonblocking in the wide sense* when putting up new calls results in avoiding all the blocking states, so that the system is effectively nonblocking.

3.6 Benes Networks

Benes considered the class of rearrangeable 3-stage Clos networks with $n = m = 2$ and $k = 2^t$ for some positive integer t. He showed that any such network can be recursively decomposed into $2t + 1$ stages, each consisting of $N/2$ 2×2 cells. Benes networks have $2(\lg N) - 1$ stages and $O(N \lg N)$ crossbars, where $N = mk = 2^{t+1}$. To illustrate Benes's decomposition, consider the 3-stage Clos network with $n = m = 2$ and $k = 4$ which is depicted in Figure 3.5. The first and last stages consist of four 2×2 crossbars, and the center stage consists of two 4×4 cells. These cells are decomposed into 2×2 crossbars and the total number of stages is five, each of the stage consists of four switches. This yields the final 8×8 Benes network.

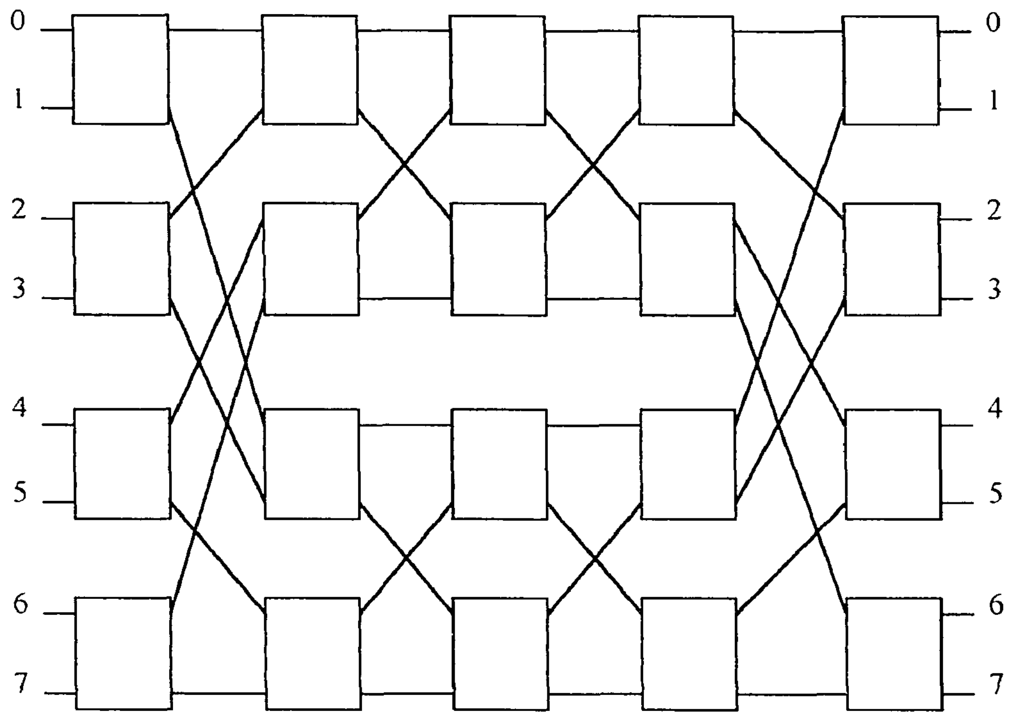

Figure 3.5 The 8 × 8 Benes network

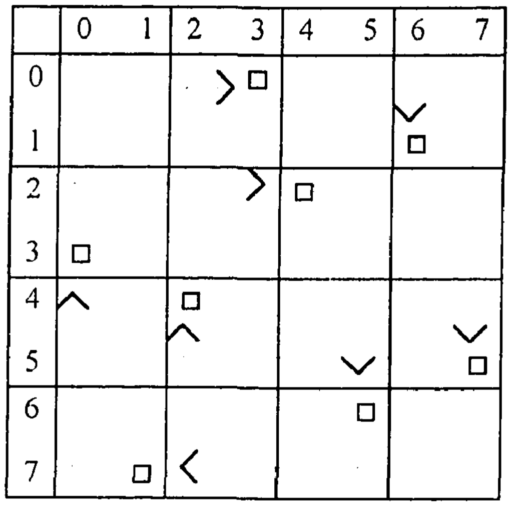

Figure 3.6 An example of Looping Algorithm

Sequential routing algorithms [42] need $O(N\log N)$ steps where N is the network size. Other methods such as the parallel processing method. heuristic method. or recursive approach are used to improve this time complexity. One of the basic algorithms is the looping algorithm. In order to illustrate the looping algorithm. consider a permutation matrix P

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 7 & 4 & 0 & 2 & 6 & 1 & 5 \end{pmatrix}$$

The looping algorithm starts recording the permutation. P as shown in Figure 3.6.

The two output numbers of a switching element in the output stage are shown in the same column. and the two input numbers of a switching element in the input stage are shown in the same row. Then choose an arbitrary entry in the chart as a starting point. For example. starting at row 23 and column 01. then look for a same-row or column entry to form a loop and choose row 23 and column 45. The process continues until a loop is obtained by re-entering row 23 and column 01. The loop's member entries are then assigned "a" and "b" alternately. The second loop can be formed in the same way. Then. assign input and output lines named "a" to subnetwork "a" and those named "b" to subnetwork "b". The looping algorithm can be applied recursively to the two subnetworks. Figure 3.6 shows an 8×8 Waksman network [35].

3.7 Discussion

The Clos network is nonblocking and rearrangeable. Any idle input terminal of the network can always be connected to any idle output terminal by rerouting the existing connections if necessary. Also, the Clos network has an area complexity less than $O(N^2)$. For systems with a large number of processors, the Clos network has the advantage of area complexity when compared with crossbar switches. The propagation delay is also an important consideration in permutation network design. Clos networks have propagation delays ranging from $O(\lg N)$ to $O(N)$, depending on the values of the design parameters. Other networks such as Benes networks are of importance. Various methods of implementing control algorithms have been developed, and will be discussed in chapter 4.

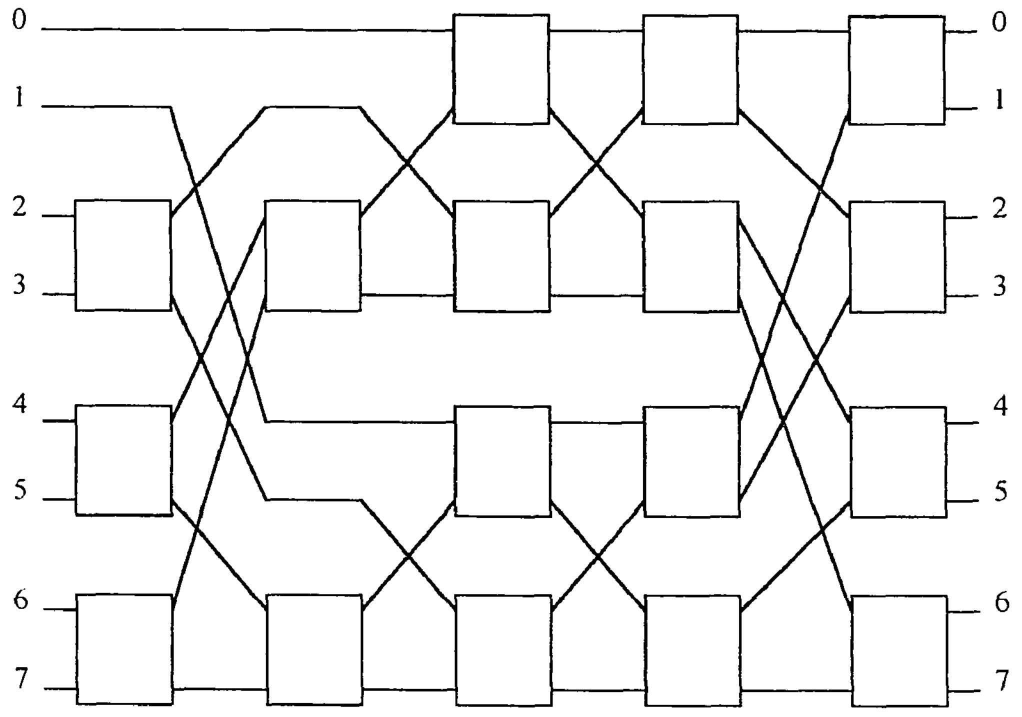

Figure 3.7 The 8 × 8 Waksman network

CHAPTER 4

DECOMPOSITION OF CLOS MINS

There are many algorithms reported in the literature for routing the Clos network. These algorithms can be classified basically in three categories: matrix decomposition, edge coloring and matching, and parallel decomposition. These algorithms determine the setting of the switches of a permutation network to realize a given permutation, or a connection pattern of every stage from the inputs to the outputs. First, in this chapter, the matrix decomposition algorithms of Neiman, Ramanujam, and Jajszczyk are studied. Also, the counter examples of Kubale and Cardot are considered. The class of routing algorithms for Clos networks which make use of edge coloring on bipartite graphs are presented. These two decomposition methods are reported to be basically the same [50]. Also, the parallel algorithms of Carpinelli are examined. Finally, Gordon's algorithm is discussed, which is a basis of the new algorithm that will be introduced in chapter 6.

4.1 Introduction

In the Clos network, central routing units are required whose function is to receive a permutation, and to find the corresponding settings for each individual switch to realize that permutation. Many routing algorithms have been developed for the Clos networks. But routing processes of the Clos network are extremely serial in nature and there often occur routing conflicts, which result in backtracking. The backtracking is going back to the previous steps when the conflict occurs in order to keep decomposing the matrix. The basic approach in routing the Clos network is to find the switch settings of the second stage switches, and from there, set the first and third stages switches accordingly. Once we know the second stage settings, we can set the rest of the switches very easily, without any calculations. However, this is not true if we try to decide the switch settings from outside to inside. Also, setting the second stage switches involves conflicts and the algorithm must backtrack to find the right switch settings. This keeps the algorithm relatively slow, and make the algorithm highly sequential. This is one of the reasons why few parallel algorithms have been developed so far. For this reason, Carpinelli's [50] parallel algorithm checks the possibility of backtracking using a partitioning technique before decomposing the matrix. One of the ways to improve the speed of the algorithm is preprocessing, which arranges the switch settings to be closer to the final settings before the algorithm starts, so the total workload can be reduced. Three approaches have been explored in the literature for decomposing the matrix: the matrix decomposition approach, the coloring and matching approach and the parallel approach, which are covered in this chapter.

4.2 Matrix Decomposition

4.2.1 Neiman's Algorithm

Neiman's algorithm consists of two stages. The first step tries to mark all k elements from the matrix. If the first step could not mark all k elements, then the second step takes over and finishes marking the elements. The algorithm is illustrated as follows.

Step 1: Starting with the left-most column, mark a non-zero element which has no marked elements in its row. Repeat this process on the next column, continuing until all columns are processed. If during this marking process, a column is found to have no non-zero entry whose previous entries in its row are not marked, then the algorithm proceeds to the next column without marking any elements in that column. If k elements are marked, then the algorithm is done; otherwise, Step 2 must be performed once for each column with no marked elements.

Step 2: If the number of marked elements in Step 1 is x, then the number of unmarked elements must be $k-x$. Mark a non-zero entry in a column with no marked elements, say $H_m[i,j]$. Unmark the other marked element in this row, $H_m[i,\acute{j}]$. Mark another non-zero element in this column, $H_m[\acute{i},\acute{j}]$, following the rule that once this stage marks an element in a row or column, no other element may be marked in that row or column until this iteration of Step 2 is completed. Continue to unmark and mark elements until no row or column has more than one marked element. This will result in a matrix with exactly one more marked element than before executing Step 2.

The k marked elements represent the setting for one of the m switches of stage 2. A marked element in a row i and column j represents that the input i of the switch is to be connected to output j of the same switch. Each marked element in H_m is then decremented by one to obtain H_{m-1}. Next the algorithm is applied to H_{m-1} to obtain the setting for another switch in stage 2, and this process is repeated until H_1 is obtained.

As an example, consider a Clos network with $m = n = 3$ and $k = 4$ with a permutation matrix

$$P = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 2 & 7 & 8 & 1 & 5 & 11 & 6 & 3 & 9 & 12 & 4 & 10 \end{pmatrix}$$

The corresponding H matrix is

$$H_3 = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 2 \end{bmatrix}$$

The first stage arbitrarily marks a non-zero element in the first column, $H_3[2,1]$. The next columns are also marked without any duplications of rows or columns which have been already marked. Here, $H_3[4,2]$ and $H_3[1,3]$ are marked arbitrarily. Next, we need to mark $H_3[3,4]$, but column 4 has no non-zero entry in a row with no marked elements, so no element is marked in this column. Since there is no marked element in the fourth column, the second step must be executed. The matrix, with asterisks representing marked elements, is

$$H_3 = \begin{bmatrix} 1 & 0 & 2* & 0 \\ 1* & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1* & 0 & 2 \end{bmatrix}$$

The second stage successively marks and unmarks elements of H_3 until it has four elements, no two of which reside in the same row or column. Arbitrarily mark a non-zero element in column 4 with no marked elements, $H_3[2,4]$. Unmark the other marked element in this row, $H_3[2,1]$, and mark a non-zero entry in the column of the unmarked element, column 1. One choice can be $H_3[1,1]$. This process is repeated until one more element is marked than was the case in Step 1. Continuing the process of unmarking and marking elements, $H_3[1,3]$ would be unmarked and $H_3[3,3]$ would be marked. Since four elements are now marked, and no two reside in the same row or column, the algorithm terminates. The matrix E_3 can be extracted from the marked elements of H_3 as shown below

$$H_3 = \begin{bmatrix} 1* & 0 & 2 & 0 \\ 1 & 1 & 0 & 1* \\ 1 & 1 & 1* & 0 \\ 0 & 1* & 0 & 2 \end{bmatrix} \text{ and } E_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

The H_2 matrix is obtained by subtracting E_3 from H_3 matrix, and H_2 can also be decomposed using the same method described above, which would leave another two solutions, E_2 and E_1. The time complexity of Neiman's algorithm is known to be $O(nk^2)$ for pass 1 and $O(n^2k^2)$ for all. For large k, Neiman's algorithm displays high time complexity, although his method holds for every possible permutation.

4.2.2 Ramanujam's Algorithm

Ramanujam [36] uses a different matrix than the other algorithms in this class, but it is related to the H matrix. He uses the *allocator matrix* M which has dimension $k \times k$, and $M[i,j]$ is the set of all destinations of inputs to first-stage switch j which are output at third-stage switch i. It is actually the transpose of H_m, with the entries listed rather than counted. The phase of the algorithm which extracts the desired matrix operates as follows. Set up a $k \times k$ matrix T, where $T[i, j]$ is the maximum element of $M[i, j]$, or 0 of $M[i, j]$ is empty. The largest element of T is marked, and its row and column are crossed off. This is repeated on the submatrix left in T until T is null or contains all zeros. If T is null, the marked elements define a matrix for extraction. These elements are deleted from M, and the process is repeated until M is null. If T is not null, reform T, replacing the largest value with a zero, and repeat this stage, choosing the largest element of T. The marked elements form the E matrix. As an example, consider the Clos network with $m = n = 3$, and $k = 4$. The permutation to be realized is given as

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 3 & 8 & 7 & 9 & 5 & 11 & 6 & 1 & 10 & 0 & 4 \end{pmatrix} .$$

The allocator matrix M and the T matrix are

$$M = \begin{bmatrix} \{2\} & \Phi & \{1\} & \{0\} \\ \{3\} & \{5\} & \Phi & \{4\} \\ \{8\} & \{7\} & \{6\} & \Phi \\ \Phi & \{9\} & \{11\} & \{10\} \end{bmatrix} \quad \text{and} \quad T = \begin{bmatrix} 2 & \Phi & 1 & 0 \\ 3 & 5 & \Phi & 4 \\ 8 & 7 & 6 & \Phi \\ \Phi & 9 & 11 & 10 \end{bmatrix}$$

From the T matrix, it can be seen that the largest element is $T[3, 2]$. Mark this element, and then delete the row 3 and column 2. Since the largest remaining element is 8, mark $T[2, 0]$ and then delete row 2 and column 0. Continuing the same procedure, $T[1, 1]$ and $T[0, 3]$ can be chosen. Marking each of the chosen elements with asterisk, the resulting M is

$$M = \begin{bmatrix} \{2\} & \Phi & \{1\} & \{0\}^* \\ \{3\} & \{5\}^* & \Phi & \{4\} \\ \{8\}^* & \{7\} & \{6\} & \Phi \\ \Phi & \{9\} & \{11\}^* & \{10\} \end{bmatrix}$$

From the marked M matrix, extract one of the solution matrix E_3

$$E_3 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The other two solution matrices E_1 and E_2 can be obtained in the same manner.

$$E_1 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad E_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

4.2.3 Kubale's Counterexample

Ramanujam's algorithm decomposes a $k \times k$ matrix M of sets of integers, called the allocator matrix, into n matrices having exactly one nonzero integer in each row and column. Kubale [37], however, noticed that the algorithm is incorrect for $k \geq 4$ because it may run into an endless loop in Step 3, although it works well for $k < 4$. For example,

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 4 & 0 & 3 & 1 & 5 & 6 & 7 \end{pmatrix}$$

with $n = 2$ and $k = 4$. From the step described above, the allocator matrix M becomes as follows:

$$M = \begin{bmatrix} \Phi & \{0\} & \{1\} & \Phi \\ \{2\} & \{3\} & \Phi & \Phi \\ \{4\} & \Phi & \{5\} & \Phi \\ \Phi & \Phi & \Phi & \{6,7\} \end{bmatrix}$$

By choosing the maximum integer in each of the sets $m_{i,j}$ we obtain an integer matrix

$$T = \begin{bmatrix} \Phi & \{0\} & \{1\} & \Phi \\ \{2\} & \{3\} & \Phi & \Phi \\ \{4\} & \Phi & \{5\} & \Phi \\ \Phi & \Phi & \Phi & \{7\} \end{bmatrix}$$

Since 7 is the largest element, $T[3,3]$ is marked, and row 3 and column 3 are removed, leaving

$$T = \begin{bmatrix} \Phi & \{0\} & \{1\} & . \\ \{2\} & \{3\} & \Phi & . \\ \{4\} & \Phi & \{5\} & . \\ . & . & . & . \end{bmatrix}$$

The next largest element is 5, and $T[2,2]$ is marked. T then becomes

$$T = \begin{bmatrix} \Phi & \{0\} & . & . \\ \{2\} & \{3\} & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix}$$

From the above matrix, it is obvious that the invalid choice is made by selecting $T[1,1]$ and then, $T[0,0]$, which has no elements. Since we are unsuccessful in selecting four nonzero integers, we must set the largest element of the original T matrix to zero and go back to previous steps. Then the T matrix becomes

$$T = \begin{bmatrix} \Phi & \{0\} & \{1\} & \Phi \\ \{2\} & \{3\} & \Phi & \Phi \\ \{4\} & \Phi & \{5\} & \Phi \\ \Phi & \Phi & \Phi & \Phi \end{bmatrix}$$

However, going back to previous steps is of no effect here because constructing the new T is based on same matrix M, and the algorithm loops indefinitely, thus showing that the Ramanujam's algorithm does not work in all cases.

4.2.4 Jajszczyk's Algorithm

Neiman has shown that the control of the rearrangeable switching network can be interpreted as a procedure of finding a set of E matrices which can be subtracted, one at a time, from some given H_m, and the E matrices are permutations to be realized by the middle-stage switches, which a one denoting a crosspoint to be closed and a zero to be open. Jajszczyk [38] used another approach to find a set of E matrices, which is illustrated as follows.

Step 1: For each row and column of the matrix H_m, find the number of zeros.

Step 2: Find the row or column with the maximum number of zeros and mark an arbitrarily chosen nonzero element in this row or column.

Step 3: Cross out the row and the column containing the marked element. The size of the matrix is essentially reduced by one, although the indices of the elements remain unchanged.

Step 4: Repeat the procedure $m-1$ times, starting from step 1, for the reduced matrix. The last element is always a nonzero element and is marked after $m-1$ repetitions of the procedure.

Step 5: Form an elementary permutation matrix E with the elements $E[i,j]$ given by

$$E[i,j] = \begin{cases} 0, & \text{if } h_{i,j} \text{ is not marked} \\ 1, & \text{if } h_{i,j} \text{ is marked} \end{cases}$$

The obtained E matrix is then subtracted from the H_m matrix, and the procedure is repeated for the resultant matrix H_{m-i} $(0 \leq i \leq m-1)$, until the matrix H_1 is obtained. Notice that the matrix H_1 is equal to matrix E_1. Jajszczyk's algorithm is simple and the time complexity of the algorithm is $O(nk^2)$, which is fast among the matrix decomposition algorithms.

4.2.5 Cardot's Counterexample

Jajszczyk's algorithm is very efficient and works pretty well. However, Cardot [39] has found some errors in this algorithm. For example, the H_4 matrix with $k = 10$ is given below.

$$H_4 = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 2 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 2 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

According to Steps 1 to 3 of Jajszczyk's algorithm, the elements $H[3,6]$, $H[5,7]$, $H[6,1]$ and $H[8,2]$ can be marked, and all the rows and columns containing the marked elements are crossed out, leaving

$$H_4 = \begin{bmatrix} . & . & 1 & 0 & 1 & . & 0 & . & 0 & 2 \\ . & . & 0 & 0 & 1 & . & 1 & . & 0 & 0 \\ . & . & . & . & . & . & . & . & . & . \\ . & . & 0 & 0 & 1 & . & 0 & . & 0 & 1 \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & 1 & 1 & 0 & . & 0 & . & 2 & 0 \\ . & . & . & . & . & . & . & . & . & . \\ . & . & 0 & 0 & 0 & . & 2 & . & 0 & 1 \\ . & . & 0 & 0 & 1 & . & 1 & . & 0 & 0 \end{bmatrix}$$

Suppose we choose the element $H[3, 7]$ which has maximum five zeros in its column. At the next step, column 9 will be empty, so the algorithm is blocked, which means there is a flaw in Jajszczyk's algorithm.

4.3 Parallel Decomposition

4.3.1 Carpinelli's Algorithm

Ramanujam's algorithm and Jajszczyk's algorithm fail because they could not predict the partitionability of the given permutation matrix in advance. Any algorithm to perform a matrix extraction must have the capability to determine whether a partitioning exists. Neiman's algorithm achieves partitioning by convolving the marked elements until the partitions are accounted for although he never explicitly checked for them. Carpinelli's algorithm [50] introduces a concept of partitioning which accounts for the failure these matrix decomposition algorithms. An algorithm to recognize this partitioning is given below.

```
partition($H_m E_m$)
{
    int    $H'_m$, $partition\_exists$, $M_1$, $M_2$;
    $H'_m = H_m$;    $E_m = 0$;
    while ($H'_m$ != 0) {
        $partition\_exists$=NO;
```

```
            generate_partition($H'_m$, $partition\_exists$, $M_1$, $M_2$);
            if ($partition\_exists$==NO) {
                pick $i$, $j$ such that $H_m[i,j] \neq 0$;
                $H'_m = H'_m \backslash i, j$;    $E_m[i,j] = 1$;
            }
            else {
                partition($M_1$, $E_1$);
                partition($M_2$, $E_2$);
                $E_m = E_m + E_1 + E_2$;    $H'_m = 0$;
            }
        }
    }
```

First, the algorithm initializes the variables H'_m and E_m. The while loop adds elements to E_m until it becomes a permutation matrix. The subroutine *generate_partition()* is to check if the partition exists. If a partition exists, the subroutine forms the submatrices and returns them in M_1 and M_2. If a partition does not exist, i and j are chosen to mark an arbitrary non-zero element. If a partition exists, two partition submatrices are processed recursively. The subroutine *generate_partition()* is shown below, which is a heart of the algorithm.

```
generate_partition($H'_m$, $partition\_exists$, $M_1$, $M_2$)
{
    int    $R$, $C$;

    $M_1 = 0$; $M_2 = 0$;
    parfor (each possible set of rows of $H'_m$) {
        $R$ =set of rows of $H'_m$;
        $C$ =set of all columns of $H'_m$ that have at least one non-zero in a row of $R$;
        if ($|R| = |C|$) {
            $M_1$ = rows and columns of $H'_m$ in $R$ and $C$;
            $M_2$ = rows and columns of $H'_m$ not in $R$ and $C$;
            *partition_exists*=YES;
        }
    }
}
```

This subroutine generates all the possible sets R in parallel, and checks all possible partitions. First, the subroutine initializes the variables, and checks the partitionability in parallel. The condition of the partitionability can be checked by extracting the sets R and C and checking the number of elements in the two sets. If the number of elements in the two sets is the same, this means that a partition does exist, and partition submatrices M_1 and M_2 are formed and the flag is set. Once a partition is found, parallel executions are terminated, and the subroutine exits, returning the values obtained. Returning to the subroutine *partition()*, two partition submatrices are recursively processed and partial E matrices are created. A partial matrix is a matrix with one or more rows of all zero elements. Then, these partial E matrices are combined together to form E_m.

For example, consider the matrix

$$H_m = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 1 \\ 2 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 \end{bmatrix}$$

First, variables are initialized by setting $H'_m = H_m$ and $E_m = 0$, and starts partitioning. Then *partition_exists* is set to NO and calls subroutine *generate_partition()*. One of these execution has $R = \{1,3\}$ and $C = \{1,3\}$. Since $|R| = |C| = 2$.

partition_exists is set to YES, and M_1 and M_2 become

$$M_1 = \begin{bmatrix} 1 & 2 \\ & \\ 2 & 1 \\ & \end{bmatrix} \text{and } M_2 = \begin{bmatrix} & \\ 2 & 1 \\ & \\ 1 & 2 \end{bmatrix}$$

Since a partition does exist, Step 3b recursively processes M_1 and M_2, resulting in two E_1, E_2 matrices

$$E_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \text{and } E_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

These two matrices are added, resulting in the final matrix E_m

$$E_m = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

4.4 Edge coloring and Matching

So far, matrices are used to represent the Clos network and the decomposition has taken place on that basis. Another approach to represent a permutation network is by using the bipartite multigraph. The bipartite multigraph G can be expressed as a triplet $\{V_1, V_2, E\}$, where V_1 and V_2 are sets of vertices and E is the multiset of all edges of the multigraph. The coloring is the process of assigning tags, or colors to each edge such that no vertex has more than one edge of a given color incident to it. This is actually a means of minimizing the number of colors used. The matching is the process of creating a set of edges such that no two are incident to a common vertex. The following algorithms make use of coloring and matching to effectively decompose the permutation.

4.4.1 Introduction

The graph theoretic approach to finding the setting of the switches of stage 1 starts by treating each switch in stages 0 and 2 as a vertex in a multigraph G. Let the set of switches of stage 0 be denoted as $V0$ and the set of switches of stage 2 be denoted as $V2$. Then, given a permutation P, an edge is added between vertex i and vertex j if an inlet attached to switch i of stage 0 is to be routed to an outlet attached to switch j of stage 2. The result of this is the bipartite multigraph $G = (V0, V2, E)$, where E is the set of edges between $V0$ and $V2$. G is a multigraph since multiple edges between vertices are allowed, and is bipartite since each edge in G is incident to two vertices, one in $V0$ and the other in $V2$. The degree of G, which is the number of edges incident on any vertex, is clearly m. The graph theoretic approach then decomposes G into m subgraphs, each of degree 1. Each such subgraph will represent the setting of one of the m switches of stage 1. An edge in a subgraph between vertex i, $i \in V0$, and vertex j, $j \in V2$, indicates that an input to switch i is to be connected to an output of switch j. These settings insure that no conflict will occur in stage 1 and all required paths specified by the permutation will be accommodated.

Many algorithms have been proposed to decompose G in the general case [40]. Hwang's algorithm runs in $O(k^{5/2})$ time. Other algorithms also exist where techniques such as *edge coloring* and Euler partitioning are used. The graph based algorithms are outside the scope of this thesis. The two routing approaches mentioned above have been discussed extensively in the literature and the graph theoretic techniques have always been described as more efficient. However, it has been found that both edge coloring and direct matrix decomposition approaches are equivalent [51]. This finding may well lead to a new, unified routing algorithm that makes Clos network particularly suitable for processor interconnection in large-scale multiprocessor systems.

4.4.2 Vizing's Method

Vizing's method [48] of coloring a bipartite multigraph uses the method of alternating path. The multigraph is initially uncolored, and each iteration adds one more colored edge to the multigraph. Assume that edge (i, j) which is incident to vertices i and j is uncolored. In the multigraph for Clos networks, each vertex has degree m. Since this edge is uncolored, vertices i and j are each missing at least one color. Assume that vertex i is missing color a and vertex j is missing color b. If they both miss the same color, that edge can be colored by the missing color. Color edge (i, j) with a. This now leaves two edges incident to vertex j with color a and none with the color b, so change the color of the other edge from a to b. If this causes another vertex to have two edges colored b, change the color or the other edge from b to a, and continue until the coloring is valid. Since the multigraph is bipartite, and both vertex sets have the same cardinality, there must be at least one other vertex which needs color a. The alternating path, the path of edge color changes, will eventually find this vertex. An algorithm based on Vizing's method which was formalized by Gabow and Kariv [46] is shown below.

```
augment()
{
    let vertex $i$ miss color $a$ and vertex $j$ miss color $b$;
    let $S$ be the subgraph of edges colored $a$ or $b$;
    let $P$ be a connected components of $S$ incident to $i$ or $j$;
    interchange color $a$ and $b$ on the edges of $P$;
    color edge $(i, j)$;
}
```

As an example, consider Figure 4.1. First, edge (x_1, y_1) is selected. Since vertex x_1 does not have color a and y_1 misses color b, an alternating path of colors a and b will be formed. Then color edge (x_1, y_1) with a. The time complexity

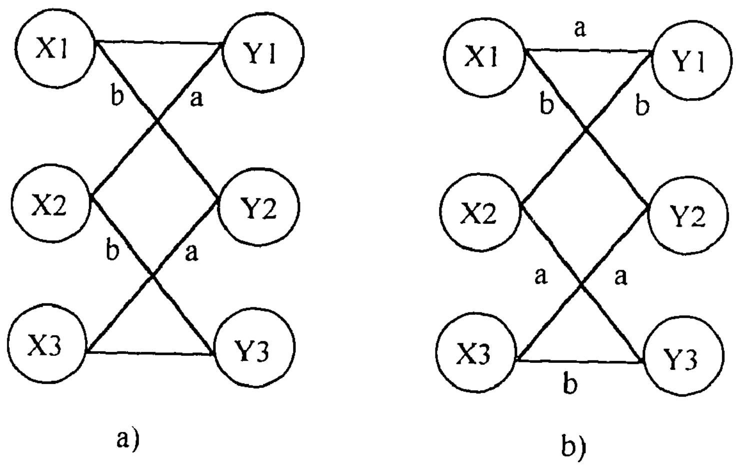

Figure 4.1 Augmenting bipartite multigraphs: (a) before, (b) after

of this algorithm for the complete coloring is $O(|V| \cdot |E|)$ where $|V|$ is the number of vertices in the multigraph, and $|E|$ is the number of edges. Since $|V| = 2k$, and $|E| = nk$ for the Clos network, the time complexity is again $O(nk^2)$. Likewise, the space complexity is $O(|V| + |E|)$, which reduces to $O(nk)$.

4.4.3 Euler Partitions

The Euler partition uses a divide-and-conquer technique. This partitions the edges of G into open and closed paths, so that each vertex of odd/even degree is the end of exactly one/zero open paths. Figure 4.2 shows the Euler partitioning of a graph. The partition enables the division of G into two edge-disjoint subgraphs G_1 and G_2. A path can be found by starting at a vertex of odd or even degree and selecting an edge. Add it to the path, traverse the edge from the original vertex to the other vertex it is incident to, and remove it from G. Repeat the process until a vertex of zero degree is reached. If $E \neq \Phi$ then repeat the entire process. Once the multigraph is reduced to a set of paths, the subgraphs can be determined. This procedure can be formalized as the following recursive algorithm.

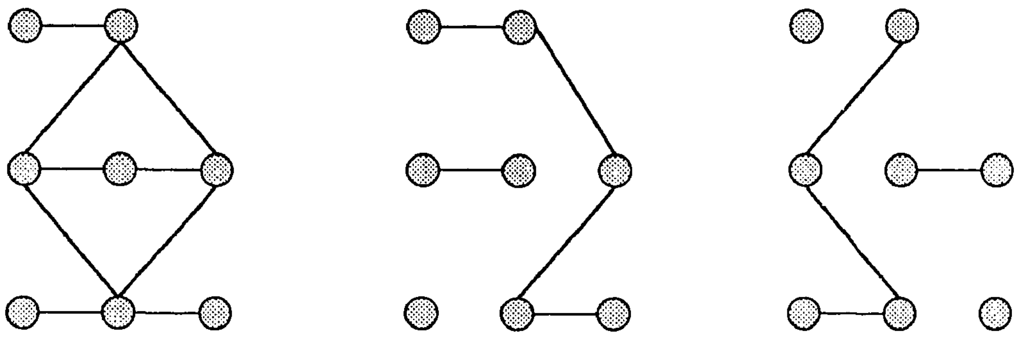

Figure 4.2 Euler partitioning

```
BEGIN
1.    Let $\delta$ be the maximum degree in $G$;
2.    If $\delta = 1$ THEN color all edges in $G$ using a new color
      ELSE
      BEGIN
3.        Form $G_1$ and $G_2$ using an Euler partition such that neither
          subgraph has degree $> \lceil \delta/2 \rceil$;
4.    Euler-color($G_1$);
5.    Euler-color($G_2$);
      END;
END
```

4.4.4 Gabow's Modified Algorithm

Gabow [45] presents a modified version of the previous algorithm which always determines a minimal edge coloring. If the degree of the vertex is odd, the algorithm finds a matching of all vertices having maximum degree. The edges in this matching are colored and removed from the multigraph. This reduces the degree of the multigraph by one, and the degree now becomes even. The rest of the algorithm follows the same procedure as the previous one, as illustrated below.

```
PROCEDURE EC($G, \delta$);
BEGIN
PROCEDURE REC(G,$\delta$);
BEGIN
1.   IF $\delta$ is odd THEN
     BEGIN
2.       IF $\delta = 1$ THEN $M := G$ ELSE MD($G.M$);
3.       Let $c$ be a new color;
4.       FOR each edge $e \in M$ DO
         BEGIN
5.       color($e$) := $c$;
6.       Delete $e$ from $G$;
         END;
     END;
7.   EP($G, P$);
8.   IF $P$ is not empty THEN
9.   Make $L_1$ and $L_2$ empty lists;
10.  For each path $p$ in $P$ DO
     BEGIN
11.  Let $p$ be the sequence of edges $e_1, \cdots, e_r$;
12.  For $i$:= 1 to $r$ DO
13.  IF $i$ is odd THEN put $e_i$ in $L_1$ ELSE put $e_i$ in $L_2$;
```

```
          END;
14.       FOR $i$:= 1,2 DO
          BEGIN
15.             Let $G_i$ be the multigraph consisting of the edges in $L_i$ and the
                vertices incident to them;
16.             REC($G_i$, $\lfloor \delta/2 \rfloor$);
          END;
     END;
     END (REC);
17.  Delete all vertices of degree 0 from $G$;
18.  Let $\delta$ be the maximum degree of a vertex;
19   REC($G$, $\delta$);
     END(EC);
```

MD is a procedure which finds M which is a matching of all vertices of maximum degree. EP forms P, the set of paths needed to derive the Euler partition. Gabow's algorithm runs in time $O(nk^{3/2} \lg\ k)$ for the Clos network where $m = n$.

4.5 Gordon's Algorithm

Unlike the above algorithm, Gordon [43] uses a unique method to decompose the matrix, although the nature of his algorithm is the same as the coloring decomposition. He defined two $k \times n$ matrices S and C, called the specification and count matrices, respectively. The relations between the H, S, and C matrices can be seen in Figure 4.3. If we use the notation proposed by Neiman in reference to the Clos network, then the necessary connections are assumed and expressed as a permutation

0	1	2	2
1	3	2	0
0	4	4	3
3	3	0	4
2	4	1	1

S matrix

1	1	2	0	0
1	1	1	1	0
1	0	0	1	2
1	0	0	2	1
0	2	1	0	1

H matrix

2	0	1	1
1	1	1	1
1	0	2	1
1	2	0	1
0	2	1	1

C matrix

Figure 4.3 Relations between the H, S, and C matrices

$$P = \begin{pmatrix} 0 & 1 & \dots & i & \dots & N-1 \\ \pi(0) & \pi(1) & \dots & \pi(i) & \dots & \pi(N-1) \end{pmatrix}$$

where inlet i is to be connected to outlet $\pi(i)$, $0 \le i \le N-1$, and $N = nk$. Initially, S is set to represent the specification in the following way. All elements of S are unassigned. Then for each signal $i, 0 \le i \le N-1$, calculate x and t where $x = \lfloor i/n \rfloor$ is the first-stage input switch at which signal arrives, and $t = \lfloor \pi(i)/n \rfloor$ is the last-stage output switch to which it should be routed, and set the next unassigned element in the xth row of S to t. On the other hand, each element of C, $c[x, y]$, $0 \le x \le k-1$, $0 \le y \le n-1$, is initialized to the number of occurrences of the integer x in column y of S. The pointers cx and sx represent rows of C and S matrices respectively, and y and z represent columns of the S or C matrix. As an example, a sample P matrix and resulting S and C matrices when $k = 4$ and $n = 3$ are

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 10 & 3 & 5 & 6 & 11 & 7 & 1 & 9 & 4 & 0 & 8 \end{pmatrix}$$

$$S = \begin{bmatrix} 0 & 3 & 1 \\ 1 & 2 & 3 \\ 2 & 0 & 3 \\ 1 & 0 & 2 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

Algorithm: Initially, sx is set to zero.

Step 1: Find a row cx, in column y of C such that $c[cx, y] = 0$. If no such element can be found then increment y until either such an element is found or all columns are satisfied, in which case the algorithm halts with a solution.

Step 2: If we have not halted we must have found $c[cx, y] = 0$. There must therefore be another column z (greater than y since we are leaving only satisfied columns to the left), such that $c[cx, z] > 1$. This follows since there are exactly n copies of each element (0 to $n-1$) in each row, so a missing element in one column implies a repeated element in another. We increment z, from the initial value y, until $c[cx, z] > 1$.

Step 3: We now have a column z of S that contains more than one copy of the missing element cx. Repeatedly increment sx (mod k) until $s[sx, z] = cx$. As explained later, this way of setting sx prevents the algorithm from entering a loop in which the same elements are swapped repeatedly on successive passes.

Step 4: Swap the elements $s[sx, y]$ and $s[sx, z]$, thus inserting the missing element $s[sx, z]$ into column z of S. This will as a side effect reduce the number of elements $s[sx, z]$.

Step 5: Increment $c[cx, y]$ and $c\{s[sx, y], z\}$ and decrement $c[cx, z]$ and $c\{s[sx, y], y\}$.

Step 6: Increment cx (mod k) and go to Step 1.

Example: The application of Gordon's algorithm is illustrated by the following sequences of matrices for the example. The two elements of the scheduling matrix that have been swapped are marked by *; the incremented and decremented elements in C are marked by $+$ and $-$. The P matrix is given as

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 4 & 1 & 2 & 11 & 6 & 8 & 9 & 7 & 10 & 0 & 3 & 5 \end{pmatrix}$$

The S and C matrices are

$$S = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 1 & 2 & 2 & 1 \\ 2 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 0 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

In the first iteration, $cx = 0$, $y = 0$, $z = 1$, and $sx = 2$. The resulting matrices are as follows with the swapped elements marked with asterisks.

$$S = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 1 & 2 & 2 & 1 \\ 0^* & 2^* & 0 & 1 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1^+ & 1^- & 2 & 0 \\ 2 & 0 & 0 & 2 \\ 0^- & 2^+ & 1 & 1 \end{bmatrix}.$$

In the second iteration, $cx = 2$, $y = 0$, $z = 1$. and $sx = 1$.

$$S = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 2^* & 1^* & 2 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 1^- & 1^+ & 0 & 2 \\ 1^+ & 1^- & 1 & 1 \end{bmatrix}$$

In the third iteration, $cx = 1$, $y = 2$, $z = 3$, and $sx = 2$.

$$S = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 2 & 1 & 2 & 1 \\ 0 & 2 & 1^* & 0^* \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 1 & 1^- & 1^+ \\ 1 & 1 & 1^+ & 1^- \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

In this example, S becomes the solution matrix after the third step of the algorithm. The run time is dominated by the number of swaps, which has time complexity $O(nk^{3/2})$. Gordon's algorithm is basically a special kind of edge coloring algorithm. Each column of the decomposed S matrix determines the switch setting of a second stage switch whose destination is given by elements in that column. Since the Clos network has connections from each center-stage switch to each of the last-stage switch, elements in each column of the S matrix are not identical. Gordon's algorithm, however, has been found to display errors which will be discussed in chapter 6.

4.6 Discussion

Neiman's algorithm, which consists of two stages, works for all permutation. However, the matrix algorithms of Jajszczyk and Ramanujam are faster, but do not work for all permutations. This is due to the improper choice of elements in the H matrix which leads to errors in the algorithms. This can be prevented using the partitioning, which works for all permutation and does not require backtracking. Gordon's algorithm uses two matrices for the decomposition. However, his algorithm is closer to the coloring algorithms in nature because elements in each column of the decomposed S matrix can be considered as edges colored with one of n different colors. The H, S, and C matrices are closely related, and each matrix has its own characteristics. Although the H matrix and bipartite multigraphs are basically the same, edge coloring algorithms usually work faster than matrix decomposition algorithms without any errors. Gordon's algorithm does not work for all cases; this will be discussed in chapter 6. Also, a new routing algorithm is introduced based on Gordon's algorithm. The routing algorithms for fault tolerant Clos networks are discussed in chapter 7.

CHAPTER 5

FAULT TOLERANT MINS

5.1 Introduction

In chapter 3, we reviewed the interconnection networks that can be applied for parallel/distributed computer systems and switching networks. However, these interconnection networks provide only one path from a given network input to a given output. Hence, if there is a single hardware fault, fault-free communication will not be possible between some network input/output pairs. Different approaches to fault tolerant multistage interconnection networks have been studied. In general, MINs can be made fault tolerant by adding extra hardware such as switches, interstage links and multiplexers/demultiplexers. Adding extensive hardware usually decreases performance degradation under faulty condition, but increases the cost and size. Adding little hardware, on the other hand, increases performance degradation under faulty conditions but keeps the cost and size down. As a consequence, a compromise must be made where the trade-offs are weighed carefully and the best design is reached. A good fault tolerance technique is one that needs minimal hardware and causes minimal performance degradation under faulty conditions. Any fault tolerance technique should cause no performance degradation under normal conditions. As the extreme case, the duplication provides two networks in parallel, with one being active and the other being standby. If a fault occurs, the standby network is switched in and the faulty network is switched out, and normal operation resumes. This approach provides the same performance in faulty conditions as in normal conditions, but increases the cost and size of the system.

A number of fault tolerant MINs have recently been reported for multiprocessor systems. The details of these techniques depend mainly on the type of network and the fault tolerance model used. Fault tolerance has also been provided for some other network architectures through various approaches. In this chapter, some of the fault-tolerant MINS are discussed including the Extra-Stage Cube (ESC) and fault-tolerant Clos network (FTC). The advantages and disadvantages of each network will be discussed. This will help explain the problem of fault tolerance, and thus will facilitate its solution. The reconfiguration of the fault tolerant networks when faults occur is considered.

5.2 Extra Stage Cube (ESC) Network

The ESC network is formed from the generalized cube (GC) network by adding one extra stage and multiplexers/demultiplexers to activate the bypass extra stage (stage 3) or the output stage (stage 0) [56]. An ESC network for $N = 8$ inputs is shown in Figure 5.1. The stages are numbered in decreasing order from 3 to 0 starting from the extra stage. Stage 3 offers two types of paths depending on the states of the multiplexers. This results in an additional path being available from each source to each destination. A stage is enabled when its interchange switches provide paths to the next stage. It is disabled when its interchange switches are bypassed. Enabling and disabling of stages 3 and 0 is accomplished by having dual input/output ports, and multiplexers and demultiplexers to select between the input/output lines. Figure 5.2 details interchange switches for stages 3 and 0. At stage 3, a multiplexer selects between two sets of identical input lines, one of which bypasses the stage 3 switch and the other of which routes through the switch. At stage 0, a demultiplexer provides the option of bypassing the switch or routing data through it. Failures may occur in network interchange switches, links between interchange switches, and network input/output lines. Failed components of the network are considered unusable until replaced or repaired.

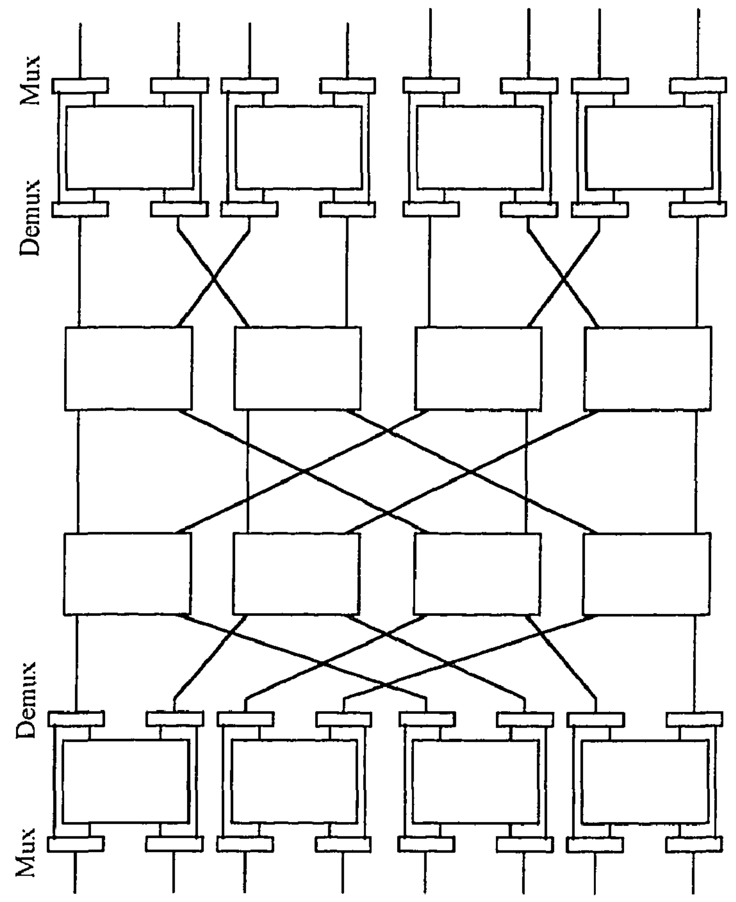

Figure 5.1 The Extra Stage Cube (ESC) network

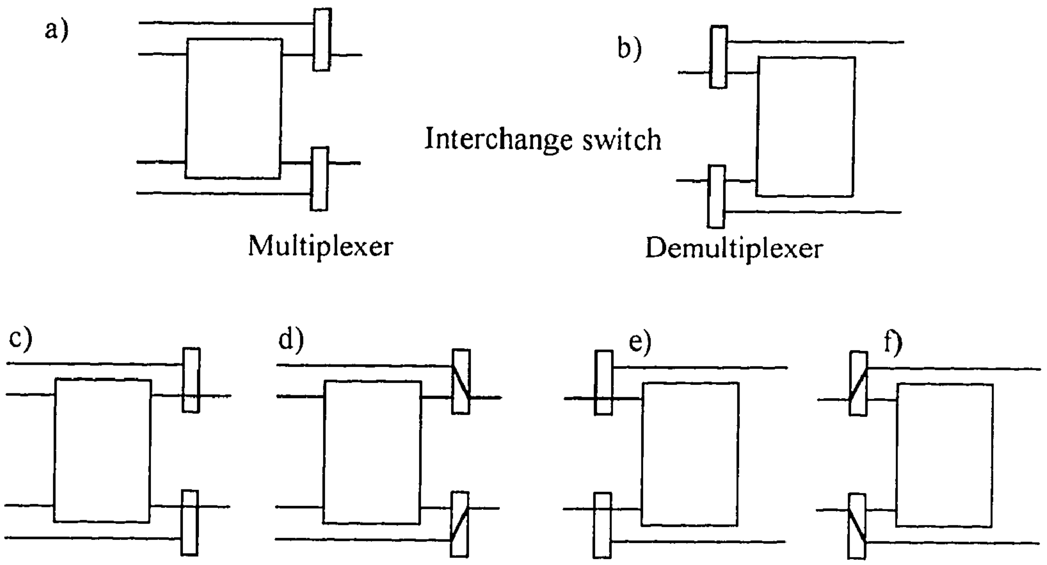

Figure 5.2 The Extra Stage Cube Network: (a) Stage 0 interchange switch (b) Stage 3 interchange switch (c) Stage 0 enabled (d) Stage 0 disabled (e) Stage 3 enabled (f) Stage 3 disabled

Once a fault occurs in the network, the network is recovered in the following ways. It is assumed that the ESC network can be tested to determine the existence and location of faults. If an input line connected to a stage 3 multiplexer fails, stage 3 is enabled and the nonfaulty input line is used instead. If the fault is on an input line to a stage 3 interchange switch, that line is unused and the system continues to ignore the faulty line. If an output line from a stage 0 switch to a PE is faulty, the network is reconfigured as if stage 0 is faulty. If the fault is on an output line from a demultiplexer, that line is unused and the system continues to ignore the faulty line. Stage 3 and 0 enabling and disabling may be performed by a system control unit. In normal operation, stage 3 is disabled and stage 0 is enabled. This fault-free ESC is topologically identical to a GC. If after running fault detection and location tests a fault is found, the network is reconfigured. If the fault is in stage 0, stage 3 is enabled and stage 0 is disabled. For faults in a link or switch in stages 2 or 1, both stages 3 and 0 will be enabled. Stage 3 of the ESC network allows access to two distinct stage 2 inputs. Stages 2 to 0 of the ESC network form a GC topology, so each of the two stage 2 inputs has a single path to the destination, and these paths are distinct except for the stage 3 and 0 switches, which are fault-free in this case. Thus, at least one fault-free path must exist.

The ESC uses a routing tag scheme for the control of the network, which is similar to the exclusive-or tag scheme for the GC network. The ESC network uses 4-bit routing tag $T = t_3t_2t_1t_0$ for the one-to-one source to destination connection. The tag values depend on whether the ESC network has a fault, as well as the source and destination addresses, which need to be computed. If the network is fault free, stage 3 is disabled and the routing tag is $T = t_3^*t_2t_1t_0$, where t_3 is ignored and can take any value. If there is a fault in a network link or switch in stages 2 to 1, stage 3 is enabled, and bit 3 of the tags can be used to control stage 3 and select between the one of two paths. The primary path is used if it is not faulty. However, if it is faulty, the secondary path is used. For routing tags, $T = 0\ t_2t_1t_0$ yields the primary path and $T = 1\ t_2t_1\overline{t_0}$ the secondary path. Stage 0 uses $\overline{t_0}$ instead of t_0 to compensate for the swap already performed by stage 3. If the fault is in stage 0, stage 3 is enabled and stage 0 disabled. A routing can be accomplished by substituting stage 3 for stage 0, because both stage 0 and stage 3 perform same functions. In this case, the tag is $T = t_0t_2t_1t_0^*$, where t_0^* is ignored because stage 0 is disabled. The t_3 is now set as t_0, and stage 3 performs the function of stage 0.

The fault size of the ESC is 1, and any inputs must remain capable of accessing any outputs after the ESC recovers from a fault. The ESC is robust in the presence of multiple faults. The ESC offers a straightforward routing method. In addition, the multiplexers and demultiplexers need to be set only after a fault occurs. Also, the ESC does not need specially designed switches. Simple binary switches and 1×2 multiplexers/demultiplexers are used in order to form the ESC along with interstage links. On the other hand, the ESC requires $N/2$ extra switches in addition to N multiplexers and N demultiplexer to achieve fault tolerance for a MIN of size N. Also, there must be an external hardware unit to set all the multiplexers and demultiplexers so that data is routed through stage 3 rather than being bypassed when a fault occurs. Furthermore, after recovering from a fault, additional time is needed to find if the fault lies on the primary path or on the secondary path before generating a new routing tag. This time constitutes performance degradation, as it slows down the system. Although the ESC has many advantages and drawbacks, this network is considered one of the best fault tolerant MINs reported.

5.3 Fault tolerant Clos Networks (FTC)

The fault tolerant Clos network adds fault tolerance to the ordinary Clos network by using extra switches and multiplexers/demultiplexers [60]. Recall that the Clos network of size N must have $k = N/m$ switches of size $m \times n$ in stage 0, and k switches of size $n \times m$ in stage 2. The n switches of stage 1 must be of size $k \times k$. An ordinary Clos network has $n = m$. However, when $n > m$, some degree of fault tolerance is obtained since extra paths exist in the network.

The FTC achieves fault tolerance in the following ways. To make the outer stages fault-tolerant, k_{sp} extra switches are added to each of these two stages. Also, n_{sp} extra switches are added to the middle stage in order to make it fault tolerant. In the FTC, each inlet is connected by a demultiplexer to $1 + n_{sp}$ distinct switches in stage 0. Also, each outlet is connected by a multiplexer to $1 + n_{sp}$ distinct switches in stage 2. These multiplexers and demultiplexers serve as a fault recovery mechanism in case of a fault in either of the two outer stages. Figure 5.3 shows the FTC with $n = k = m = 3$, and $k_{sp} = n_{sp} = 1$.

An FTC with $N = mk$ is formed from an ordinary Clos of size N as follows. First, use $k + k_{sp}$ switches with size $m \times (n + n_{sp})$ in each of the outer stage. Then the original center stage switches must be enlarged from $k \times k$ to $(k + k_{sp}) \times (k + k_{sp})$.

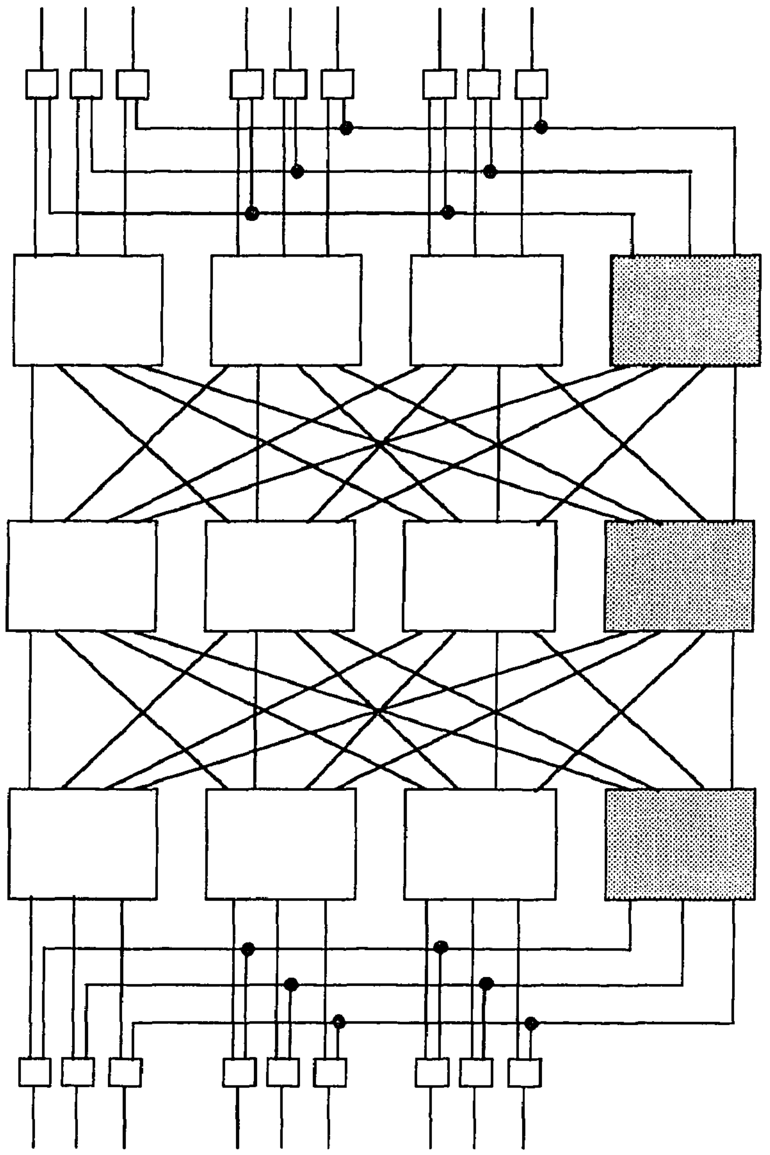

Figure 5.3 The FTC with $m = k = 3$, and one extra switch in each stage

Connect the network inlets to the inputs of the first stage switches via $1 \times (n_{sp} + 1)$ demultiplexers, and the network outlets to the outputs of the third stage switches via $1 \times (n_{sp} + 1)$ multiplexers.

For the FTC, the fault model is defined as follows.

1. Any switch can fail.

2. Any interstage link can fail.

3. External links and multiplexers/demultiplexers cannot fail.

It should be mentioned that the faults are assumed to occur independently, and that faulty components are unusable. The fault tolerance criterion of the FTC is complete recovery, that is, regaining pre-fault connectivity after a fault occurs.

5.3.1 Reconfiguration of the FTC

It is important for the FTC to be reconfigured in case of faults in order to regain its pre-fault connectivity. Consider an FTC network with $n_{sp} = k_{sp} = 1$. Let three switches be $X(f_i, i)$, $0 \leq i \leq 2$, where f_0, f_1, and f_2 are unused switches of the first, second, and third stage, respectively. The configuration of the FTC at any time is a function of the present values of f_0, f_1, and f_2. In general, the reconfiguration of the FTC can be performed through one or more of the following operations:

- Setting the multiplexers and demultiplexers

- Terminal relabelling

- Permutation translation

As will be seen below, the value of f_1 affects the terminal relabelling, while the values of f_0 and f_2 affect the settings of multiplexers/demultiplexers and permutation translation. The multiplexer/demultiplexer setting is performed if an outer stage switch fails.

When the FTC is not faulty, one switch in each stage will be unused. This unused switch can be any switch, but for convenience it will be assumed to be the last switch in each stage, i.e., $X(k, 0)$, $X(n-1, 1)$, and $X(k, 2)$. This choice is convenient because it makes the multiplexers and demultiplexers remain in state 0 under normal conditions. When a fault occurs, they can switch to state 1, thereby avoiding the defective switch. Permutation translation is also performed if an outer stage switch fails. Let $P = \{P_0, P_1, ..., P_{N-1}\}$ be an arbitrary permutation of $\{0, 1, ..., N-1\}$. In the actual network, P_i is the outlet to which inlet i is to be connected. In an ordinary Clos network, P goes directly to the central routing unit where the settings of the individual switches are extracted and delivered to the switches for implementation. In the FTC, the same steps are to be taken with the exception that permutation P is translated before it goes to the central routing unit. Terminal relabelling is performed if a middle-stage switch fails.

As mentioned above, f_1 affects the labelling of the outputs of switches $X(z, 2)$, $0 \leq z < k+1$. Let these outputs and inputs be referred to as the inward terminals of the outer stages or just the inward terminals. In each of these switches, only m out of the n inward terminals will be used, and will be referred to as the active terminals. Each active terminal will have two labels: a local one, to be used by the switch's control unit, and a global one, to be used by the central routing unit. The local label is an integer z, $0 \leq x < m$, and the global label is also an integer Z, $0 \leq Z < m(k+1)$. The active terminals will be labeled from top to bottom locally, with respect to the switch, as the sequence $0, 1, ..., m-1$. Globally, the active terminals that were labelled from top to bottom locally will be labelled from top to bottom, with respect to the stage, as $0, 1, .., m(k+1)-1$. The labels are updated always after a fault occurs, and the current labels are used to implement the routing information received from the control unit. More details about the terminal relabelling can be found in [60]. The reconfiguration of the FTC network can be illustrated more straightforward using the S and C matrices, as can be seen in following examples.

5.3.2 Examples

To illustrate this, consider a permutation P of the FTC with $n = k = 3$, and one spare switch in each stage.

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 4 & 5 & 0 & 8 & 7 & 3 & 2 & 6 \end{pmatrix}$$

Initially, let the unused switches be $X(3,0)$, $X(3,1)$, and $X(3,2)$ in each of the three stages of the FTC. Recall that this is the configuration suggested to be used under normal conditions. Then, permutation Q, according to the rules set forth above, will be

$$Q = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 3 & 4 & 8 & 7 & 6 & 1 & 2 & 5 & 0 & x & x & x \end{pmatrix}$$

The H and S matrix representations of P are

$$H_3 = \begin{bmatrix} 0 & 2 & 1 \\ 1 & 0 & 2 \\ 2 & 1 & 0 \end{bmatrix} \qquad S = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

On the other hand, the H and S matrix representations of Q are

$$H_3 = \begin{bmatrix} 0 & 2 & 1 & 0 \\ 1 & 0 & 2 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \qquad S = \begin{bmatrix} 1 & 1 & 2 & * \\ 2 & 2 & 0 & * \\ 0 & 1 & 0 & * \\ \# & \# & \# & \end{bmatrix}$$

The size of the matrix H increases by exactly one row and one column, and the S matrix also has an additional row and column. The additional paths due to extra switches in the outer stages are represented as pound characters, and asterisks for the extra switches in the middle stage, which are explained in greater detail in chapter 7.

As another example, again consider a permutation of P of the FTC with $n = k = 3$, and one spare switch in each stage. Assume that switches $X(1,0)$, $X(2,1)$ and

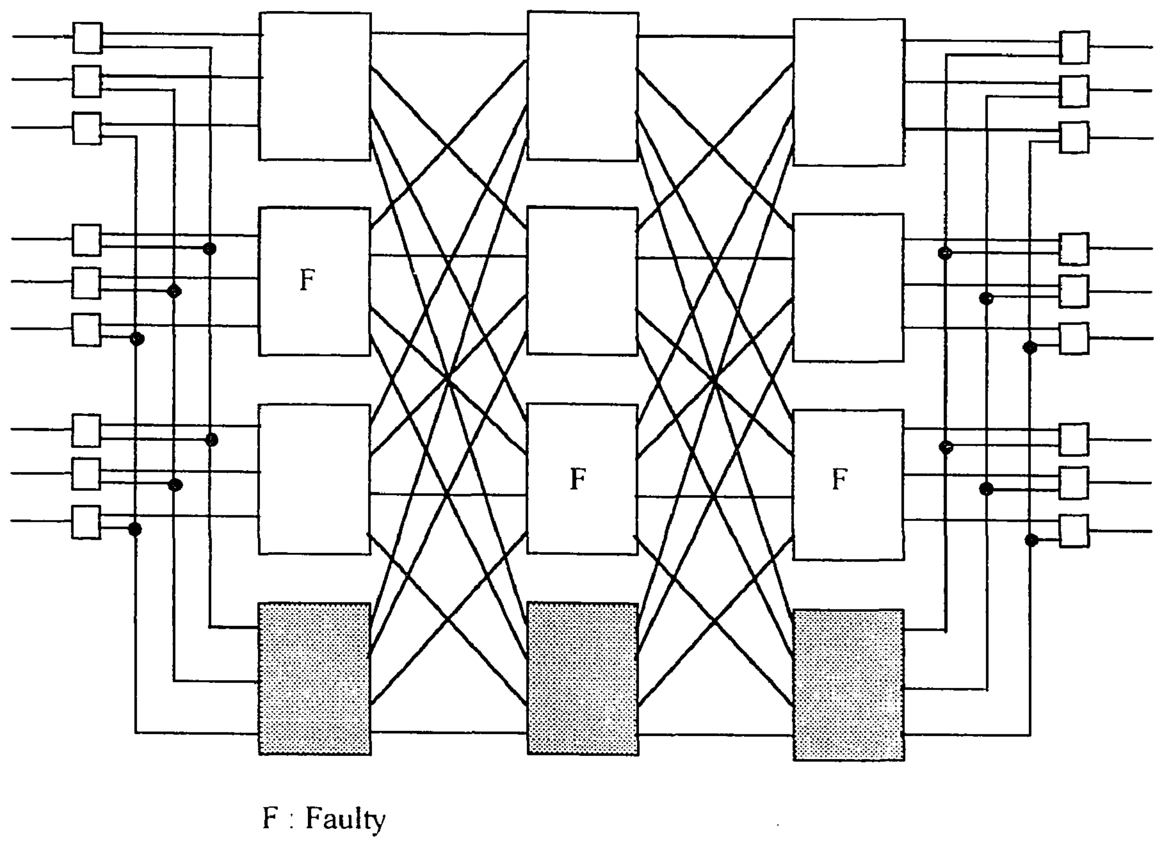

Figure 5.4 The faulty FTC with $X(1,0)$, $X(2,1)$, and $X(2,2)$ faulty switches

$X(2,2)$ suddenly failed, as shown in Figure 5.4. Due to the failure of $X(2,1)$, the inward terminals of stages 0 and 2 should be relabelled. Specifically, inward terminal number 2 of each switch should be left out in assigning the numbers. The failure of $X(1,0)$, and $X(2,2)$ affects the permutation translation. Permutation P, given before, is translated according to the rules laid down above to

$$Q = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 3 & 4 & 11 & x & x & x & 2 & 5 & 0 & 10 & 9 & 1 \end{pmatrix}$$

The routing result will be implemented by all the switches except those that are defective, namely, $X(1,0), X(2,1)$ and $X(2,2)$. The matrix representation of permutation Q above is

$$H_3 = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 0 & 0 & 3 & 0 \\ 2 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \end{bmatrix}$$

In the previous example, the S matrix was be given by

$$S = \begin{bmatrix} 1 & 1 & 2 & * \\ 2 & 2 & 0 & * \\ 0 & 1 & 0 & * \\ \# & \# & \# & \end{bmatrix}$$

Since $X(1,0)$ is defective and $X(3,0)$ is a spare switch, all input signals are moved to the extra switch, and $X(1,0)$ becomes unusable, which is denoted as dots.

$$S = \begin{bmatrix} 1 & 1 & 2 & * \\ . & . & . & . \\ 0 & 1 & 0 & * \\ 2 & 2 & 0 & * \end{bmatrix}$$

Also, the faulty condition of $X(2,1)$ forces the elements in column 2 to be bypassed to the spare switch $X(3,1)$ which is represented as column 3, resulting in

$$S = \begin{bmatrix} 1 & 1 & . & 2 \\ . & . & . & . \\ 0 & 1 & . & 0 \\ 2 & 2 & . & 0 \end{bmatrix}$$

Finally, the faulty condition of $X(2,2)$ prevents the use of the second switch of in the third stage. Instead, the signals assigned to this switch must now use the spare switch which will be denoted as 3. Thus the resulting matrix is

$$S = \begin{bmatrix} 1 & 1 & . & 3 \\ . & . & . & . \\ 0 & 1 & . & 0 \\ 3 & 3 & . & 0 \end{bmatrix}$$

Representing the reconfiguration of the network using the S matrix shown above presents complications because of the introduction of dots in the rows and columns of the matrix. In chapter 7, the reconfiguration matrix is introduced which retains all the information of each switch's use without swapping the rows or columns.

CHAPTER 6

NOVEL ALGORITHM FOR CLOS MINS

6.1 Introduction

Although Gordon's algorithm is simple and fast, as discussed in chapter 4, his algorithm does not work for all permutations. His algorithm has two special features. First, the use of two matrices in the algorithm contributes to the improvement of the time complexity since it helps to find the number of occurrences of each element directly. The next is the use of $sx \pmod k$, which is the heart of the algorithm and makes the algorithm very effective. In this chapter, it will be shown that Gordon's algorithm does not work for all cases, and a counterexample will be given in section 6.2. Next, a new simple algorithm will be introduced. This algorithm is based on the Gordon's algorithm. Three kinds of swaps by which this algorithm realizes the desired mapping are discussed: 1) simple swap, 2) next simple swap, and 3) successive swap. Also, we are going to prove that the new algorithm works for all permutations. In section 6.4, the worst case and the average behavior of the algorithm are discussed in detail.

6.2 Failure of Gordon's Algorithm

The algorithm given by Gordon is very simple, fast, and works well when the matrix size is moderate. Although Chiu and Siu [44] claimed the incorrectness of the algorithm, it stemmed mainly from a typographical subscript reversal, which led to a misunderstanding about the algorithm. Gordon reaffirmed in his reply that the algorithm is still valid. However, our research found that his algorithm may run into an infinite loop for $k \geq 5$. The heart of his algorithm lies in the repeated increment of $sx \pmod k$ until $s[sx, z] = cx$ as shown in step 3 of his algorithm. Recall that

cx represents a row of C which satisfies $c[cx, y] = 0$. This way of setting sx is intended to prevent the algorithm from entering a loop in which the same elements are swapped repeatedly on successive passes. The setting of sx, on the other hand, is influenced by the choice of cx. However, Gordon did not mention anything specific about the way of setting next cx after two elements in row sx of S are swapped. This is especially true if row cx of C reaches $k-1$ while column y of C still contains zeros. It is quite possible that the increment of $cx \pmod k$ until $c[cx, y] = 0$ must have been used in the algorithm because this is the most easy and efficient way to choose the next value of cx. It is not likely that Gordon chose cx after some calculations because, if he had done that, he certainly would have made it clear in the paper. We have tested this algorithm on several possible cases. These include 1) increment of $cx \pmod k$ after the swap, 2) decrement of cx, and reset to $k-1$ when $cx < 0$, and 3) random choice of cx, until $c[cx, y] = 0$ for all three cases. An example for the first case is given below. The two elements of the C matrix that have been swapped are marked by *; the incremented and decremented elements in S are marked by $+$ and $-$. Suppose that currently, $cx = 0$, $y = 1, sx = 4$, and

$$S = \begin{bmatrix} 0 & 0 & 2 & 4 & 1 \\ 3 & 1 & 3 & 2 & 0 \\ 4 & 0 & 1 & 3 & 2 \\ 1 & 0 & 4 & 1 & 4 \\ 2 & 2 & 3 & 4 & 3 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 3 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 1 & 1 \\ 1 & 0 & 1 & 2 & 1 \end{bmatrix}$$

After the first repetition, $cx = 1$, $y = 1$, $sx = 1$, $z = 2$, and

$$S = \begin{bmatrix} 0 & 0 & 2 & 4 & 1 \\ 3 & 3^* & 1^* & 2 & 0 \\ 4 & 0 & 1 & 3 & 2 \\ 1 & 0 & 4 & 1 & 4 \\ 2 & 2 & 3 & 4 & 3 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 3 & 0 & 0 & 1 \\ 1 & 0^- & 2^+ & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1^+ & 1^- & 1 & 1 \\ 1 & 0 & 1 & 2 & 1 \end{bmatrix}$$

The second repetition yields $cx = 4$, $y = 1$, $sx = 4$, $z = 3$, and

$$S = \begin{bmatrix} 0 & 0 & 2 & 4 & 1 \\ 3 & 3 & 1 & 2 & 0 \\ 4 & 0 & 1 & 3 & 2 \\ 1 & 0 & 4 & 1 & 4 \\ 2 & 4^* & 3 & 2^* & 3 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 3 & 0 & 0 & 1 \\ 1 & 0 & 2 & 1 & 1 \\ 1 & 0^- & 1 & 2^+ & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1^+ & 1 & 1^- & 1 \end{bmatrix}$$

After the third repetition, $cx = 1$, $y = 1$, $sx = 1$, $z = 2$, and

$$S = \begin{bmatrix} 0 & 0 & 2 & 4 & 1 \\ 3 & 1^* & 3^* & 2 & 0 \\ 4 & 0 & 1 & 3 & 2 \\ 1 & 0 & 4 & 1 & 4 \\ 2 & 4 & 3 & 2 & 3 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 3 & 0 & 0 & 1 \\ 1 & 1^+ & 1^- & 1 & 1 \\ 1 & 0 & 1 & 2 & 1 \\ 1 & 0^- & 2^+ & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The fourth repetition yields $cx = 2$, $y = 1$, $sx = 4$, $z = 3$, which reduces to the first matrix and enters into an infinite loop. When examining the above example, it can be clearly seen that the use of (mod k) incrementing of sx does not always effectively prevent the process from repeatedly finding the same element in following passes. In most cases, this does not happen and the algorithm behaves well, especially when $k < 5$. However, as k increases, the algorithm has more chances to enter a loop independent of the ways of setting cx as described above. Chiu and Siu [44] reported a new algorithm by modifying Gordon's algorithm without giving the time complexity and proof that it works for all permutations. Also, their algorithm is trivial, so it will not be covered in this thesis. In the next section, a new algorithm is introduced for decomposing Clos networks which is based on Gordon's algorithm. This can be done by scanning the C matrix row-by-row, and by a class of swaps, which will be explained later.

6.3 New Algorithm for Clos Networks

Although Gordon's algorithm is simple and fast, his algorithm has been demonstrated to have errors in some permutations as shown in the previous section. In this section, a new algorithm will be discussed which is based on Gordon's algorithm, but uses a different approach. In order to describe the algorithm, we shall use the notation proposed by Neiman in reference to the Clos network. The necessary connections are assumed and expressed as a permutation:

$$P = \begin{pmatrix} 0 & 1 & \dots & i & \dots & N-1 \\ \pi(0) & \pi(1) & \dots & \pi(i) & \dots & \pi(N-1) \end{pmatrix}$$

where inlet i is to be connected to outlet $\pi(i)$, $0 \le i \le N-1$, and $N = mk$. This algorithm uses two $k \times n$ matrices S and C, called the specification and count matrices, which were described in chapter 2. In order to obtain the S matrix from the permutation matrix, the following step must be taken. Initially, all elements of S are unassigned. Then for each signal $i, 0 \le i \le N-1$, calculate x and t where $x = \lfloor i/n \rfloor$ is the first-stage input switch at which signal arrives, and $t = \lfloor \pi(i)/n \rfloor$ is the last-stage output switch to which it should be routed, and set the next unassigned element in the xth row of S to t. The first stage switches are denoted by x, and the second stages are represented by y. Each element of $s[x,y]$ is the destination switch in the third stage. Each element of C, $c[x,y]$, $0 \le x \le k-1$, $0 \le y \le n-1$, is initialized to the number of occurrences of the integer x in column y of S.

As an example, a permutation P and the S and C matrices when $k = 4$ and $n = 3$ is as follows.

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 10 & 3 & 5 & 6 & 11 & 7 & 1 & 9 & 4 & 0 & 8 \end{pmatrix}$$

$$S = \begin{bmatrix} 0 & 3 & 1 \\ 1 & 2 & 3 \\ 2 & 0 & 3 \\ 1 & 0 & 2 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

In order to explain the algorithm, it is necessary to define some of the terms that are going to be used.

Definition 1: A column of C is *d-missing* if that column does not contain any d. On the other hand, a column of C is *d-excessive* if there are more than one d in that column.

Definition 2: When a column y in the C matrix is *d-excessive* and a column z is *d-missing*, an element which satisfies $s[sx, y] = d$ in the S matrix for $0 \leq sx \leq k-1$, is called a *swapping element* and $s[sx, z]$ is called the *swapped element.*

Definition 3: When $s[sx, y]$ is a swapping element and $s[sx, z]$ is a swapped element, then two elements $s[sx, y]$ and $s[sx, z]$ are *simply swappable* if $s[sx, y] < s[sx, z]$ and $c[cx, y] = c[cx, z] = 1$ for $0 \leq cx < s[sx, y]$.

Definition 4: When $s[sx, z]$ is a swapped element and $s[sx, y]$ is a swapping element, then two elements $c[cx, y]$ and $c[cx, z]$ are *successively swappable* if $s[sx, y] > s[sx, z]$ and $c[cx, y] = c[cx, z] = 1$ for $0 \leq cx < s[sx, y]$.

Definition 5: When two elements $s[sx, y]$, and $s[sx, z]$ are swapped because of being successively swappable, an element $s[sx_1, y]$ which satisfies $s[sx_1, y] = s[sx, y]$ is called $s[sx, y]$-*alternative.*

The new algorithm is illustrated as follows.

Algorithm: Initially sx is set to zero.

Step 1: Find a column cx, in a row y of C such that $c[cx, y] > 1$. If no such element can be found then increment cx until either such an element can be found or all rows are satisfied, in which case the algorithm stops with a solution. If the algorithm has not stopped, it must have found $c[cx, y] > 1$. Set $z = 0$.

Step 2: Increment z until $c[cx, z] = 0$. This follows since there are exactly n copies of each element $(0$ to $n-1)$ in each row, so a repeated element in one column implies a missing element in another. We now have a column z of S that contains no element cx.

Step 3: (Simple Swap) Repeatedly increment $sx (\mathrm{mod}\ k)$ until $s[sx, y] = cx$. If $s[sx, z] < cx$, go to Step 2. Otherwise, swap the elements $s[sx, y]$ and $s[sx, z]$ thus removing the repeated element $cx = s[sx, y]$ in column y of S. This will, as a side effect, increase the number of occurrences of element cx in column z of S. Increment

$c[cx, z]$ and $c\{s[sx, z], y\}$ and decrement $c[cx, y]$ and $c[s[sx, z], z]$. It is easily seen that these four simple changes restore the count property. If swapped, go to Step 1.

Step 4: (Next simple swap) Repeat Step 3, thus providing one more chance to simply swap two elements in another row. If swapped, go to Step 1. This step is done only once before $c[cx, y]$ becomes 1.

Step 5: (Successive Swap) Swap $s[sx, y]$, $s[sx, z]$, and update C as in Step 3. If $s[sx, y] > cx$, go to Step 1. Otherwise, increase $sx (\text{mod } k)$ for another $s[sx, y]$ and repeat Step 5.

This algorithm works for all permutations, which can be proved using the following three theorems.

Theorem 1: Given two sets $S\epsilon$ and Sm which are Y-excessive and Y-missing, respectively, let $X\epsilon(i)$, and $Xm(i)$ be numbers with the value i in the sets $S\epsilon$ and Sm, where $0 \leq i < Y$. If the number of Y's in $S\epsilon$ is two, it is always possible to reduce the number of Y in $S\epsilon$ to one without any change in the occurrence of $X\epsilon$ and Xm.

Proof: Arranging the elements of the set $S\epsilon$ and Sm,

$$\begin{array}{ll}
X\epsilon(0) & Xm(0) \\
X\epsilon(1) & Xm(1) \\
X\epsilon(2) & Xm(2) \\
\vdots & \vdots \\
Y\epsilon & \\
Y\epsilon & \\
Z\epsilon & Zm \\
\vdots & \vdots \\
Z\epsilon & Zm \\
Zm &
\end{array}$$

There are two possible cases for $Y\epsilon$ to be swapped with an element in the set Sm.

First, if Ye and Zm are in the same row, then two elements can be swapped, resulting the reduction of number of Ye in the set Se to be one without any change in the number of occurrences in Xe or Xm. However, if Ye and Xm are on the same row, Ye and any one of $Xm(i)$, $0 \leq i < Y$ should be swapped. The index i is used in order to distinguish the elements of Xe and Xm which have the same value i. As a result, two identical numbers $Xe(Y-1)$ and $Xm(Y-1)$ are on the same Ye-excessive column. Now take $Xe(Y-1)$, which is an $Xm(Y-1)$-alternative. Again, there are two possibilities. If $Xe(Y-1)$ is in the same row with Zm, the number of Ye in the Ye-excessive column can be reduced to one without any change in the number of occurrences in X. However, if $Xe(Y-1)$ is in the same row with $Xm(Y-2)$, we need to swap $Xe(Y-1)$ and $Xm(Y-2)$, and then find the $Xm(Y-2)$-alternative which is $Xe(Y-2)$. In worst case, this process continues until $Xm(1)$ finds its alternative $Xe(0)$. Since other Xms are not in the same row with $Xe(0)$, $Xe(0)$ must select Zm, which leads to the proof of the theorem. □

Theorem 2: Given two sets of Se and Sm which are Y-excessive and Y-missing, respectively, let $Xe(i)$, and $Xm(i)$ be numbers with the value i in sets Se and Sm, where $0 \leq i < Y$. If the number of Ys in Se is three, it is always possible to reduce the number of Y in Se to one by applying simple and successive swaps.

Proof : Any Ye in the Ye-excessive column can be swapped into the Ye-missing column without any change of occurrences of Xs, which can be proved using the same procedure as in Theorem 1. Once the number of Ye's is reduced to two, Theorem 1 can be applied, so the number of Y's can be reduced to one.□

Theorem 2 can be generalized to the case when the number of Ye's is arbitrary.

Theorem 3: Given an arbitrary permutation of the S matrix, it is always possible to decompose the matrix if the C matrix is scanned row-by-row from top-to-bottom.

Proof : For an arbitrary $c[cx, y]$ in a row cx being scanned which satisfies $c[cx, y] > 1$, it is always possible to make $c[cx, y] = 1$ by applying Theorems 1 and 2. Thus, all elements in the C matrix which are greater than one can be reduced to one. □

Lemma 1: The maximum number of swaps in the successive-swap is $k - 1$.

Proof: All k elements in the $Y\epsilon$-excessive and $Y\epsilon$-missing columns can be swapped except the remaining $Y\epsilon$, which is at least one.□ .

6.4 Example

To illustrate the algorithm clearly, consider a three-stage Clos network having $n = 3$ and $k = 5$ with an H matrix as shown below.

$$H_3 = \begin{bmatrix} 2 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

The S and C matrices derived from the H matrix are shown below.

$$S = \begin{bmatrix} 0 & 0 & 2 \\ 1 & 4 & 4 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \\ 3 & 4 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 2 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

Now check the C matrix for an element that is greater than 1, which implies that more than two edges incident to the corresponding output node are colored identically. Since $C[2,2] \geq 2$ and $C[2,1] = 0$, the C matrix is 2-excessive in column 2 and 2-missing in column 1. Since $cx = 2$, we find cx in the S matrix at $sx = 0$ because sx was first set to zero. Thus, we find that $S[sx, y] = 2$, and $S[sx, z] = 0$. These two elements are not simply-swappable because $S[sx, y] > S[sx, z]$, so we move to the next row, 2, in the S matrix. Since $S[2,1] < S[2,2]$ in this case, they too are not simply-swappable, thus a forced-swap must be applied. This is done by swapping the first two elements $S[0,1]$ and $S[0,2]$, and then updating the C matrix by incrementing $C[0,2]$ and $C[2,1]$ and decrementing $C[0,1]$ and $C[2,2]$.

$$S = \begin{bmatrix} 0 & 2 & 0 \\ 1 & 4 & 4 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \\ 3 & 4 & 0 \end{bmatrix} C = \begin{bmatrix} 1 & 0 & 2 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

Since the swapped element $S[0, 2]$ in column y is 0 which is less than cx, next find the 0-alternative in column 2, which is $S[4.2]$. Now, $S[sx.y] = S[4.2]$ and $S[sx.z] = S[4.1]$. These two elements are simply-swappable since $S[4.2] < S[4.1]$ and thus can be swapped. This finishes the successive swap for $C[2.2]$ and results in

$$S = \begin{bmatrix} 0 & 2 & 0 \\ 1 & 4 & 4 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \\ 3 & 0 & 4 \end{bmatrix} C = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 2 & 1 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

Next, we proceed to $C[3.0]$, which is greater than 1. From the S matrix, we find that $S[2.0]$ is not simply-swappable with $S[2.2]$, so we move to the next 3 in the 4th row. For $S[4.0] < S[4.2]$, we can now swap two elements and the two matrices are shown below.

$$S = \begin{bmatrix} 0 & 2 & 0 \\ 1 & 4 & 4 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \\ 4 & 0 & 3 \end{bmatrix} C = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Finally, the program terminates since all the elements in the C matrix are 1. The resulting three columns of the S matrix denote the completely decomposed switch settings of the second-stage switches, and first and third stage switch settings can be derived from this. The basic idea of the algorithm is to make the C matrix all 1's by using three kinds of swaps. This means that there are no identical elements in each column of S when completely decomposed. Steps 1 and 2 find the two columns z_0 and y_0 which are cx-missing and cx-excessive from the C matrix. The cx is, on the other hand, the element in S which is missing or excessive in the same two columns of S. Then, swaps are performed from Steps 3 to 5 until all $c[cx, y]$ become 1.

6.5 Worst-case Behavior

This algorithm is simple, but deriving the exact time complexity of the algorithm is very complicated. Gordon reported the time complexity of his algorithm in his paper without giving any proof. He just mentioned that the time complexity is roughly proportional to the number of swaps. The basic difficulty in deriving the time complexity of the algorithm is as follows. First, the runtime is proportional to the number of swaps. However, it is difficult to calculate the number of swaps for a given permutation. For a given $c[cx, y] > 1$, the number of swaps to be performed must be $c[cx, y] - 1$. But, $\sum(c[cx, y] - 1)$ does not necessarily represent the total number of swaps, because one swap results in the change of four elements of $c[cx, y]$, two of them increase, and two of others decrease. Secondly, for an element $c[cx, y] > 1$, it is difficult to know analytically what kind of swaps must be performed in the worst case for a given permutation.

Considering the difficulty of analytic approaches, the next possible method is simulation. The computer simulation usually cannot prove all the possible cases as the problem becomes complex. However, it helps to narrow the bound of time complexities. For that reason, the new algorithm has been programmed and simulated for various values of n and k. Figure 6.1 shows the worst case runtime vs. k with respect to various values of n. The graph shows that the runtime of setting the Clos network increases as k increases. For a fixed k, the runtime also increases as n increases. A closer look at the graph shows that the runtime is roughly proportional to n, but in case of k, the runtime is proportional to k^x for some values of x. In order to exactly obtain the time complexity of the algorithm, these curves were fitted to the arbitrary non-polynomial function. The result of the curve fitting

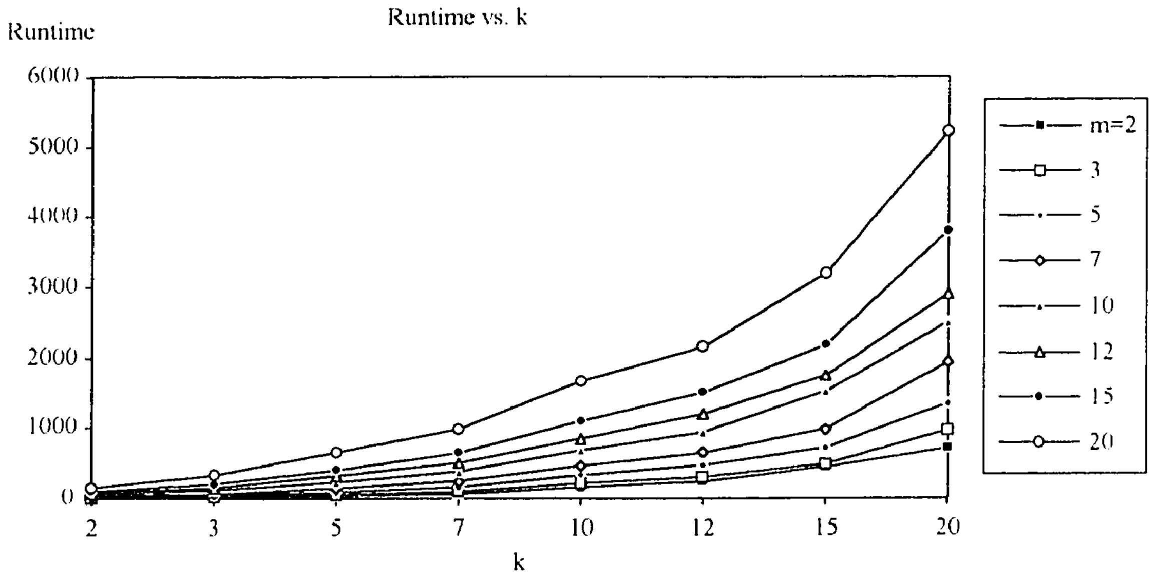

Figure 6.1 Worst case runtime vs. k

shows that the time complexity of the algorithm is proportional to $nk^{3/2}$, that is $O(nk^{3/2})$.

The simple swap dominates the other two kinds of swaps, and the next simple swap also dominates successive swaps. Simple swaps do not require much time to swap two elements. The successive swaps, on the other hand, are not frequent, but take relatively long since up to $k-1$ swaps must be made in order to reduce $c[c.x,y]$ by one. As a result, successive swaps still have considerable effects on the overall runtime although they are less frequent. Another thing to mention here is that the runtime is linearly proportional to just the number of columns n, but not to the number of rows k. This is mainly due to the use of $sx(\text{mod } k)$ and the effect of successive swaps.

6.6 Discussion

In this chapter, Gordon's algorithm has been demonstrated to display some errors in some of the permutations. A new algorithm for decomposing the Clos network which is based on Gordon's algorithm has been introduced. In this algorithm, the same S and C matrices are used to represent the Clos network, which help to speed up routings by checking C in order to calculate the number of occurrences of each element in S in each column. The basic difference between Gordon's algorithm and the new algorithm lies in the scanning direction in the C matrix. In Gordon's algorithm, it is scanned column by column, removing columns once all elements are nonidentical in each column. Swapping elements can take place between a not-yet-decomposed leftmost column and the rest of the columns. However, the new algorithm scans the C matrix row-by-row, and swapping elements are restricted to two columns for the successive swap. This gives an obvious advantage in proving that it works for all permutations, but, in Gordon's algorithm, it is difficult to prove. Another advantage to the new algorithm is that it has the potential to be run in parallel since only two columns are involved in the successive swap and other pairs of two columns can be swapped at the same time.

CHAPTER 7

ROUTING FAULT TOLERANT CLOS NETWORKS

7.1 Introduction

The Clos network can not realize all possible permutations when a fault occurs in the system. Thus, extra switches are added to the ordinary Clos network in order to achieve fault tolerance. The algorithm for the ordinary Clos network needs to be extended to the fault-tolerant cases for following reasons. First, the structure of the FTC is basically same as the ordinary Clos network except for added switches. Because of this, the representation of the network does not become complicated. Second, the spare switches can greatly simply the routing process, which is an obvious advantage when there are few or no faults in the system. Third, the same routing algorithm for the FTC can be used for the ordinary Clos network, which is a special case of the fault tolerant network. A new routing algorithm for the FTC will be introduced in section 7.2, which utilizes extra switches in all stages. For clarity, the FTC is classified into three types of networks, and in each case, the representation of the network, and routing rules are considered. In the last section, the simulation of the runtime for the FTC is discussed.

7.2 Routing the FTC

In chapter 6, we introduced a new routing algorithm for the Clos network. The algorithm for the ordinary Clos network can be extended to the fault-tolerant Clos network discussed in chapter 5. Recall that the FTC has extra switches in all stages and they provide alternative paths when faults occur in the network. However, when there are no faults in the system, these extra switches can be utilized as additional routing paths, which simplify the requirements of the routing process and reduce the runtime. The outer stage spare switches generate additional rows in the S matrix and the second stage spare switches creates additional columns. Additional paths introduced by the two types of spare switches are very flexible during the routing process, but they have different characteristics. In order to develop the new routing algorithm for the FTC, it is required to know the properties of these two types of spares and, for that reason, the fault-tolerant Clos network will be classified into three possible configurations:

I. networks with spare switches in each of the outer stages only.

II. networks with spare switches in the second stage only.

III. networks with spare switches in all stages.

In the following subsections, the networks and representation of the extra switches are discussed for all three possible cases. The rules and conditions for swapping the elements are considered, which will be the basis of the new algorithm for the FTC.

7.2.1 Routing FTC with Spare Switches in Outer Stages (Type I)

The first type of FTC has extra spare switches in outer stages only along with multiplexers and demultiplexers. In this configuration, signals are bypassed to the spare switches through the multiplexer/demultiplexers in case of faults which occur in the outer stages. Figure 7.1 shows the Type I FTC network which has one extra spare switch in each outer stage.

The S and C matrices are the same as those of the ordinary Clos network except that extra rows are added which account for the extra outer stage switches and multiplexer/demultiplexers. The elements in the xth row of S represent the signals passing through the xth switch of the first stage whose destination switches are $s[x, y]$, where $0 \leq y \leq n - 1$. The elements in the yth column of S are the signals passing through the yth second stage switch whose destinations are $s[x, y]$, where

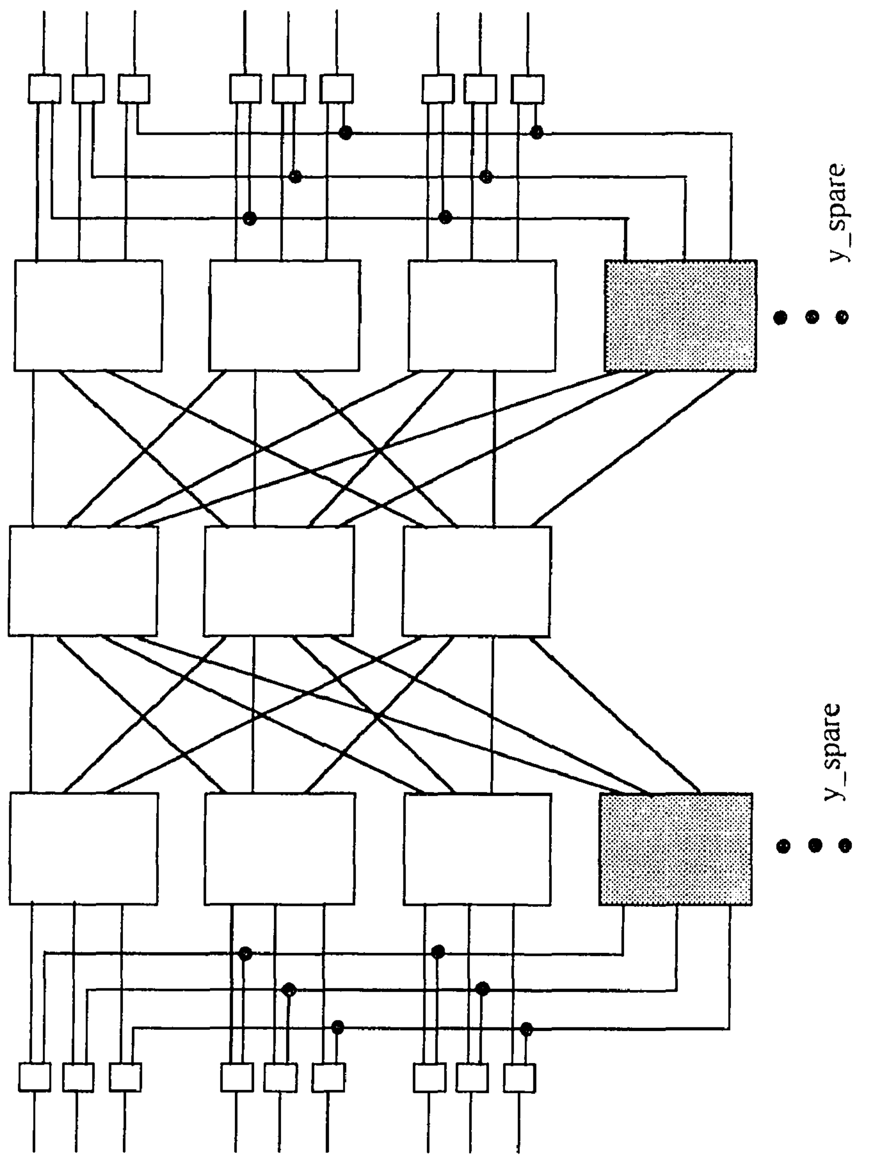

Figure 7.1 The FTC network with extra switches in the outer stages (Type I)

$0 \leq x \leq k-1$. Each element of the S matrix represents the signal directed to the last stage switch $s[x,y]$ through the yth second stage switch. Let k_{sp} be the number of spare switches in the first or third stage. If the number of spare switches are not equal in those stages, then the smaller number will be taken as k_{sp}. The k_{sp} spare switches at the outer stages create k_{sp} additional rows in the S matrix, and each element can serve as an alternative path during the routing process. The total number of extra paths is nk_{sp}. All the redundant paths due to the spare switches are denoted as $\#$ for convenience. Initially, the elements of S, $s[x,y]$, where $k \leq x \leq k + k_{sp} - 1$, $0 \leq y \leq n-1$, are initialized to spares $\#$. Also, each $c[x,y]$ of the C matrix, where $0 \leq x \leq k-1$, $0 \leq y \leq n-1$ is initialized to the number of occurrences of the integer x in column y of S. The number of spares in the S matrix is not considered in the C matrix. For example, the S and C matrices for a Type I Clos network with $n = k = 3$, and 2 spare switch in each of the outer stage can be given as,

$$S = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 2 & 0 \\ 0 & 2 & 2 \\ \# & \# & \# \\ \# & \# & \# \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & 1 \\ 0 & 2 & 1 \end{bmatrix}$$

To consider the reconfiguration of faulty switches in the outer stages, faulty switches and interstage connections must be taken into account. Recall that we have assumed no multiplexers/demultiplexers are defective. If the xth switch at the first stage is faulty, the xth multiplexer is set so that each signal in the xth switch is bypassed to the available spare switches. One of the spare switches, the rth spare switch, is assigned to these signals in order to provide alternative paths. The rth row of the S matrix is simply cleared which is denoted as dots. Now define a new matrix, the *reconfiguration matrix*, R. The R matrix is a $k \times 3$ matrix, where each row y represents the yth switch in one stage, and each column x denotes the xth stage. The element $R[y,x]$ shows that the $R[y,x]$th spare switch in the xth stage is assigned instead of the yth switch in the xth stage. For example, if the 0th switch in the first stage is defective in the above example, the S, C, and R matrices would be

$$S = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 2 & 0 \\ 0 & 2 & 2 \\ \# & \# & \# \\ \cdot & \cdot & \cdot \end{bmatrix} \qquad C = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & 1 \\ 0 & 2 & 1 \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} 3 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

Notice that the elements in the 0th row of S remains same, but the # spares in the last row are no longer available. These spares are now assigned to the signals in the 0th row, which can be seen in the R matrix where $r[0,0] = 3$. Note that other elements in R show that other switches are not reconfigured, and remain the same. Dots in the S matrix mean that there are no paths available in the 0th input switch, and they are simply ignored during the decomposition process. On the other hand, if the xth switch at the third stage is faulty, the xth demultiplexer is set so that rerouted signals from the third-stage spare switches can be bypassed to reach outlets of the xth switch. For example, if the 0th switch in the third stage is also defective, and the 4th spare switch is used instead for the above matrices, the resulting S, C, and R matrices would be

$$S = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 2 & 0 \\ 0 & 2 & 2 \\ \# & \# & \# \\ \cdot & \cdot & \cdot \end{bmatrix} \qquad C = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & 1 \\ 0 & 2 & 1 \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} 3 & 0 & 4 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

Permutation translation can also be used as was shown in chapter 5 for the reconfiguration due to the failure of outer stage switches. Faults due to the interstage links can be modeled as a switch failure and the network can be reconfigured in the same way described above. The rules and conditions for swapping elements in the ordinary Clos networks can be applied in the FTC, since the basic structures remain the same. Recall that, in the Clos network, any two elements except spares in a row x of S can be swapped. This is due to the fact that inlets input to each of the first-stage switches can be fully connected within the switch. Each first-stage switch is represented by a row of S, and each element in a row of S corresponds to the inlets to a switch which flows to the third stage switch $s[x, y]$. Each column must have no identical elements except spares when completely decomposed. This is because each second-stage switch has only one connection to each third-stage switch.

The introduction of # spares has the following features. First, # spares in a column y of S can be swapped with any elements in that column. This is due to the multiplexers and demultiplexers along with spare switches in outer stages which can bypass input signals to the spares switches. Second, spares in a column y can be swapped with any element in another column z as long as both columns maintain the same number of # spares since the number of outer spare switches is fixed in the network. When the matrix is fully decomposed, then all the elements in C matrix must be one. The zeros in the C matrix indicate that these elements are swapped with the # spares in the same column. The total number of spares in each column is restricted to k_{sp}, which does not change during the routing process.

7.2.2 Routing FTC with Spare Switches in the 2nd Stage (Type II)

In contrast to the first type, the second type of FTC has extra spare switches in the second stage. In this configuration, signals are bypassed to the extra switches in case of faults in the second stage switch y, where $0 \leq y \leq n-1$. Figure 7.2 shows the Type II FTC network with one extra spare switch in the second stage. In the Type II FTC, the S matrix is represented in a different way from the Type I FTC. Let n_{sp} be the number of spare switches in the second stage. The n_{sp} spare switches at the middle stages create n_{sp} additional columns in the S matrix, and each element in the additional column can serve as an alternative path during the routing process. The total number of extra paths is kn_{sp}. All the initial elements in the spare columns of S are denoted as asterisks (*) for convenience. These spares are wild cards, like the # spares, but different in characteristics. Also, the C matrix is defined as in the ordinary Clos network except that extra columns are added. The

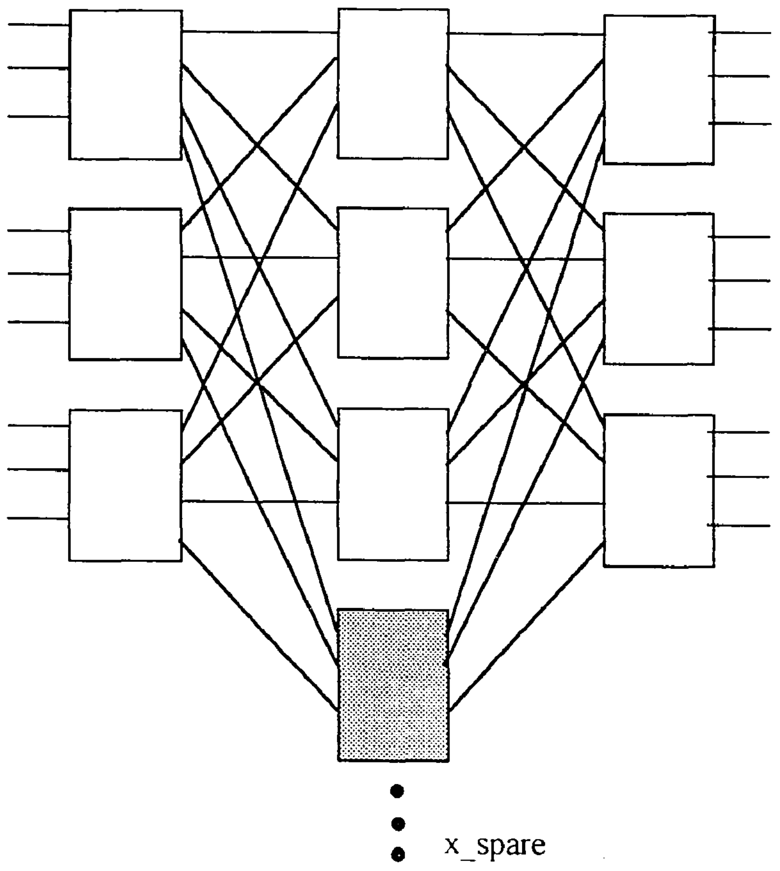

Figure 7.2 The FTC network with extra switches in the second stage (Type II)

$c[x, y]$ of the C matrix, where $0 \leq x \leq k-1$, $0 \leq y \leq n+n_{sp}-1$ are initialized to the number of occurrences of the integer x in column y of S. Each extra spare switch in the second stage generates one extra column in the S and C matrices. The elements in the yth column of S represent the signals moving to destination switches $s[x, y]$, $0 \leq x \leq k-1$ through the yth second stage switch. For example,

$$S = \begin{bmatrix} 1 & 0 & 1 & * & * \\ 1 & 2 & 0 & * & * \\ 0 & 2 & 2 & * & * \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 \end{bmatrix}$$

The right two columns of C are all zero because there are no elements between 0 to 2 in those columns, only * are in these columns. If the xth switch of the middle stage is faulty, the terminal relabelling described in chapter 5 must be performed. In the S matrix, the terminal relabelling can be achieved by clearing the xth column of S, and assigning these spares for the faulty xth second stage switch. The cleared xth column of S will be denoted as dots. The relationship between the faulty switch and spare switches will be noted in the reconfiguration matrix R as in the Type I FTC. The R matrix is used to perform the terminal relabelling of the inward terminals of the outer stages. For example, if the first switch in the middle stage in the above example is defective, and the 4th spare switch is used instead, the resulting S, C, and R matrices would be

$$S = \begin{bmatrix} 1 & 0 & 1 & * & . \\ 1 & 2 & 0 & * & . \\ 0 & 2 & 2 & * & . \end{bmatrix} \quad C = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 4 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

Notice that the elements in the first column of S remain the same, but the * spares in the last column are no longer available. These spares are assigned now to the signals in the first column, which can be seen in the R matrix where $r[1, 1] = 4$. Dots in the S matrix mean that there are no paths available in the first second-stage switch, and they are simply ignored during the decomposition process. The rules and conditions for swapping elements and * spares are as follows. First, any two elements including spares (*) in a row x of S can be swapped with any element. After the swap, the swapped elements are again free for any other swaps. This flexibility of * spares makes the routing processes very simple. However, spares in a column y can not be swapped with any elements in that column. Secondly, each column of S must have no identical elements except * spares when completely decomposed. This is because the second-stage switch has only one connection to each of the third-stage switches. Finally, the number of * spares in a column can take any value i, where $0 \leq i \leq k-1$.

7.2.3 Routing FTC with Spare Switches in All Stages (Type III)

The last type of FTC is the one with extra spare switches in outer stages, along with multiplexers and demultiplexers, as well as in the second stage. In this FTC, alternative paths are provided regardless of faults in any of the three stages. Figure 7.3 shows the type III of the FTC network with one extra spare switch in each stage.

In this type of network, there are k_{sp} spare switches in each outer stage and n_{sp} spare switches in the middle stage. The k_{sp} spare switches in the outer stages create k_{sp} additional rows in the S matrix, which has a total of nk_{sp} extra paths. Also, the n_{sp} spare switches in the middle stage create n_{sp} additional columns in the S matrix, and this can generate a total of kn_{sp} extra paths. Initially, the elements of S, $s[x,y]$, where $k \leq x \leq k + k_{sp} - 1$, $0 \leq y \leq n-1$, are initialized to # spares and $s[x,y]$, $0 \leq x \leq k-1$, $n \leq y \leq n + n_{sp} - 1$ are initialized to * spares. The elements of S, $s[x,y]$, where $k \leq x \leq k + k_{sp} - 1$, $n \leq y \leq n + n_{sp} - 1$, are denoted as blanks because spares in this area are not used as will be illustrated later in the new algorithm. Note that this area could have been initialized to * spares. Also, the C matrix has n_{sp} additional columns due to the second stage spare switches, but there are no additional rows in the matrix. The $c[x,y]$ of the C matrix, where $0 \leq x \leq k-1$, $0 \leq y \leq n + n_{sp} - 1$ are initialized to the number of occurrences of the integer x in column y of S. Each extra spare switch in the second stage generates

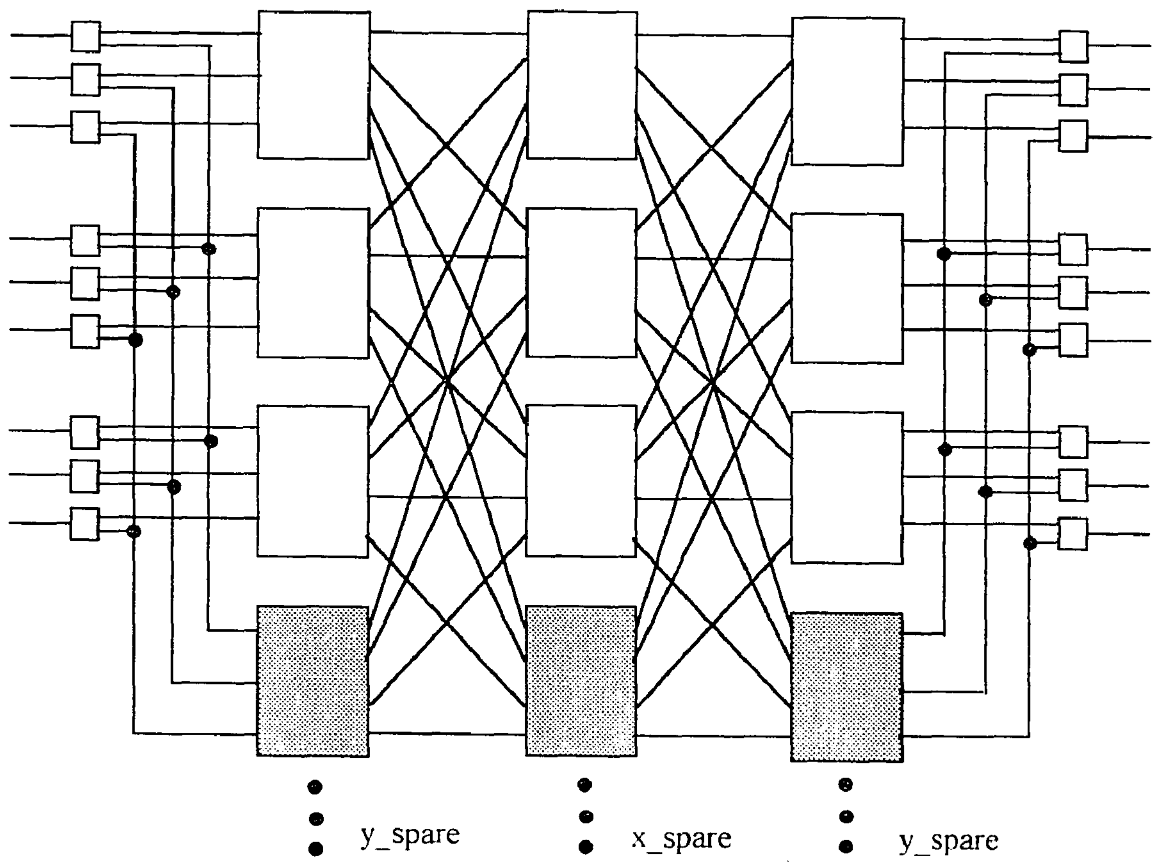

Figure 7.3 The FTC network with extra switches in all stages (Type III)

one extra column in the S and C matrices. For example, when $n_{sp} = k_{sp} = 2$,

$$S = \begin{bmatrix} 1 & 0 & 1 & * & * \\ 1 & 2 & 0 & * & * \\ 0 & 2 & 2 & * & * \\ \# & \# & \# & & \\ \# & \# & \# & & \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 \end{bmatrix}$$

Since this type of FTC has extra switches in all stages, all the defective switches need to be considered for the reconfiguration. If the xth switch in the first stage is faulty, the xth multiplexer is set so that each signal in the xth switch is bypassed to the available spare switches. One of the spare switches, the rth spare switch, is assigned to these signals in order to provide alternative paths. The rth row of the S matrix is simply cleared, which is denoted as dots. Set the R matrix with $x = 0$ as in the Type I FTC, where the element $r[y, x]$ represents that the $r[y, x]$th spare switch in the xth-stage is assigned instead of the yth switch in the xth-stage. If the xth switch in the middle stage is faulty, clear the rth column of S, and assign these spares for the faulty xth second stage switch. The cleared rth column of S will be denoted as dots. The R matrix is used to perform the terminal relabelling of the inward terminals of the outer stages. For example, if the 0th switch in the first stage and the 1st switch in the middle stage in the above example are defective, the resulting S, C, and R matrices would be

$$S = \begin{bmatrix} 1 & 0 & 1 & * & . \\ 1 & 2 & 0 & * & . \\ 0 & 2 & 2 & * & . \\ \# & \# & \# & & \\ . & . & . & & \end{bmatrix} \quad C = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} 4 & 0 & 0 \\ 1 & 4 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

The rules and conditions for swapping elements and * or # spares are as follows. First, # spares in a column y of S can be swapped with any elements in that column except * spares because the multiplexers and demultiplexer along with spare switches in outer stages can bypass signals. Also, spares in a column y can be swapped with any elements in another column z as long as both columns maintain the same number of # spares. Secondly, any two elements including * spares in a row x of S can be swapped with any element except # spares. After the swap, the swapped elements are again free to perform any other swaps. This flexibility of * spares makes the routing process very simple. However, * spares in a column y can not be swapped with any elements in that column. Third, each column of S must have no identical elements except # or * spares when completely decomposed. This is because the second-stage switch has only one connection to each third-stage switch. Finally, the number of * spares in a column is not restricted to n_{sp}, but can take any value i, $0 \leq i \leq k + n_{sp} - 1$. However, the number of # spares in a column must remain k_{sp}. Based on the above rules and conditions, the algorithm for the FTC network is introduced as follows. The S and C matrices in the FTC are the same as in the ordinary Clos network except that two kinds of spares are considered. First, the elements of S, $s[x,y]$, where $k \leq x \leq k + k_{sp} - 1$, $0 \leq y \leq n - 1$, are initialized to # spares. Also, $s[x,y]$ is initialized to * spares where $0 \leq x \leq k - 1$, $n \leq y \leq n + n_{sp} - 1$. The $c[x,y]$ of the C matrix, where $0 \leq x \leq k - 1$, $n \leq y \leq n + n_{sp} - 1$ are initialized to the number of occurrences of the integer x in column y of S.

Algorithm: Initially sx is set to zero.

Step 1: Find a column cx, in a row y of C such that $c[cx,y] > 1$. If no such element can be found then increment cx until either such an element can be found or all rows are satisfied, in which case the algorithm terminates with a solution. If the algorithm has found $c[cx,y] > 1$, set $z = 0$.

Step 2: (Wild Swap) Check whether # spares are available in column y. If available, increment $sx(\text{mod } k)$ until $s[sx,y] = cx$, then swap $s[sx,y]$ with a # spare in the column y, and go to Step 1. If not available, then check the * spare in the row sx. If the * spare is available, increment $sx(\text{mod } k)$ until $s[sx,y] = cx$, then swap $s[sx,y]$ with a spares in the row sx and go to Step 1.

Step 3: Increment $z(\text{mod } k)$ until $c[cx,z] = 0$.

Step 4. (Simple Swap) Repeatedly increment $sx(\text{mod } k)$ until $s[sx,y] = cx$. If

$s[sx,z] < s[sx,y]$, go to Step 3. If $s[sx,y] < s[sx,z]$, or $s[sx,z]$ is $*$, swap the elements $s[sx,y]$ and $s[sx,z]$ and update the C matrix. If swapped, go to Step 1.

Step 5: (Next simple swap) Repeat Step 4, thus providing one more chance to simply swap two elements in another row. If swapped, go to Step 1. This step is done only once before $c[cx,y]$ becomes 1.

Step 6: (Successive Swap) Swap $s[sx,y]$ and $s[sx,z]$, and update C as in Step 4. If $s[sx,y] > cx$ or $s[sx,y]$ is $*$, go to Step 1. Otherwise, increase $sx (\mathrm{mod}\ k)$ for another $s[sx,y]$ and repeat Step 6.

Example: For a given S and C matrix below when $n = k = 4$ and $k_{sp} = n_{sp} = 1$.

$$S = \begin{bmatrix} 1 & 1 & 2 & 3 & * \\ 1 & 3 & 2 & 0 & * \\ 2 & 0 & 2 & 3 & * \\ 1 & 0 & 0 & 3 & * \\ \# & \# & \# & \# & \end{bmatrix} \text{ and } C = \begin{bmatrix} 0 & 2 & 1 & 1 & 0 \\ 3 & 1 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & 2 & 0 & 3 & 0 \end{bmatrix}$$

First repetition: Wild swap continuously while scanning the C matrix row by row.

$$S = \begin{bmatrix} * & 1 & 2 & 3 & 1 \\ 1 & 3 & \# & 0 & * \\ 2 & \# & * & 3 & 2 \\ \# & 0 & 0 & 3 & * \\ 1 & 0 & 2 & \# & \end{bmatrix} \text{ and } C = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 3 & 0 \end{bmatrix}$$

Second repetition: Wild swap again for $cx = 3$, $sx = 3$, $y = 3$

$$S = \begin{bmatrix} * & 1 & 2 & 3 & 1 \\ 1 & 3 & \# & 0 & * \\ 2 & \# & * & 3 & 2 \\ \# & 0 & 0 & \# & * \\ 1 & 0 & 2 & 3 & \end{bmatrix} \text{ and } C = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 2 & 0 \end{bmatrix}$$

Third repetition: $cx = 3$, $y = 3$, $sx = 0$, and $z = 0$

$$S = \begin{bmatrix} 3 & 1 & 2 & * & 1 \\ 1 & 3 & \# & 0 & * \\ 2 & \# & * & 3 & 2 \\ \# & 0 & 0 & \# & * \\ 1 & 0 & 2 & 3 & \end{bmatrix} \text{ and } C = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

The wild swap greatly reduces the number of next swaps or successive swaps which take much time, since at most $k - 1$ swaps are needed in order to find the alternative paths. The number of simple swaps is also reduced. As the number of extra rows and columns increases, the algorithm has more chances to suppress the simple and successive swaps and thus improve the run time. The new algorithm works for all permutations, which can be proved using the following three theorems.

Theorem 4: Given two sets of Se and Sm which are Y-excessive and Y-missing, respectively, let $Xe(i)$, and $Xm(i)$ be numbers with the value i in the set Se and Sm, where $0 \leq i < Y$. Each set contains the same number of # wild cards, but the number of * wild cards may be different. If the number of Y's in Se is two, it is always possible to reduce the number of Y in Se to one without any change in the occurrence of Xe and Xm.

Proof : Arranging elements of the set Se and Sm,

$Xe(0)$	$Xm(0)$
$Xe(1)$	$Xm(1)$
$Xe(2)$	$Xm(2)$
$\vdots$	$\vdots$
Ye	
Ye	
Ze	Zm
$\vdots$	$\vdots$
Ze	Zm
Zm	
#	#
*	*

The proof is basically the same as Theorem 1. There are two possible cases for Ye to be swapped with an element in the set Sm. First, if Ye and Zm (or $*$) are in the same row, then two elements can be swapped, resulting in the reduction of the number of Ye in the set Se to one without any change in the number of occurrences in Xe or Xm. However, if Ye and Xm are on the same row, Ye and any one of $Xm(i)$, $0 \leq i < Y$ should be swapped. The index i is used in order to distinguish the elements of Xe and Xm which have the same value i. As a result, two identical numbers $Xe(Y-1)$ and $Xm(Y-1)$ are in the same Ye-excessive column. Now take $Xe(Y-1)$, which is an $Xm(Y-1)$-alternative. Again, there are two possibilities. If $Xe(Y-1)$ is in the same row with Zm (or $*$), the number of Ye in the Ye-excessive column can be reduced to one without any change in the number of occurrences in X. However, if $Xe(Y-1)$ is in the same row with $Xm(Y-2)$, we need to swap $Xe(Y-1)$ and $Xm(Y-2)$, and then find the $Xm(Y-2)$-alternative, which is $Xe(Y-2)$. In the worst case, this process continues until $Xm(1)$ finds its alternative $Xe(0)$. Since other Xms are not in the same row with $Xe(0)$, $Xe(0)$ must select Zm (or $*$), which leads to the proof of the theorem. □.

Theorems 2 and 3 in chapter 6 can be used similarly to prove that the algorithm for the FTC holds for all permutations.

7.3 Worst-case Behavior of the Algorithm

The new algorithm for the FTC network is similar to that of the ordinary Clos network, so deriving the exact time complexity of the algorithm with respect to the number of extra switches is a very complicated matter. In this case, the run time is dominated by the number of swaps, which consist of simple swaps, next simple swaps, successive swaps and wild swaps. Wild and simple swaps do not require much time to swap two elements. The successive swaps on the other hand, are not frequent, but take a relatively long time since they are continued until the alternative paths are found. For the FTC algorithm, the basic difficulty of deriving the time complexity of the algorithm remains the same as was explained in chapter 6. These are 1) $\sum(c[cx,y]-1)$ does not necessarily represent the total number of swaps, because one swap results in the change of four elements of $c[cx,y]$, two of which increase and two

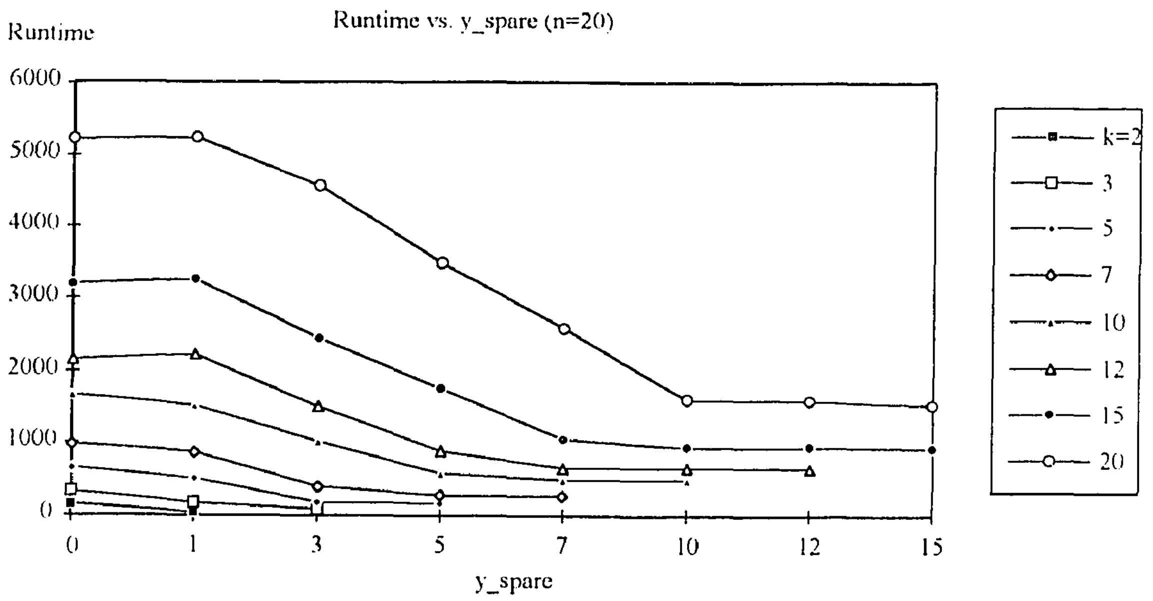

Figure 7.4 Worst case runtime vs. number of *y_spares* for various k

of which decrease. 2) For an element $c[cx, y] > 1$, it is difficult to predict analytically what kind of swaps must be performed in the worst case for a given permutation. For that reason, the new algorithm for the FTC network has been simulated to obtain the runtime of the algorithm with respect to various numbers of extra switches. Figure 7.4 shows the worst case runtime vs. *y_spare* for various values of k. The graph shows that the runtime of the algorithm for the FTC network decreases as *y_spare* increases. This is continued until *y_spare* reaches about $k/2$, where the runtime is saturated to a certain value. Runtime can be reduced to far less than half of that when there are no extra switches in the network.

Figure 7.5 shows the worst case runtime vs. *x_spare* for various values of n. As in the previous figure, the graph shows that the runtime of the algorithm for the FTC network decreases as *x_spare* increases. This is continued until *x_spare* reaches about $k/2$. But, the runtime decreases more slowly in this case, and it is reduced to slightly more than half of that when there are no extra switches in the network. Figures 7.6 and 7.7 show the average runtime versus the number of extra switches *y_spare*

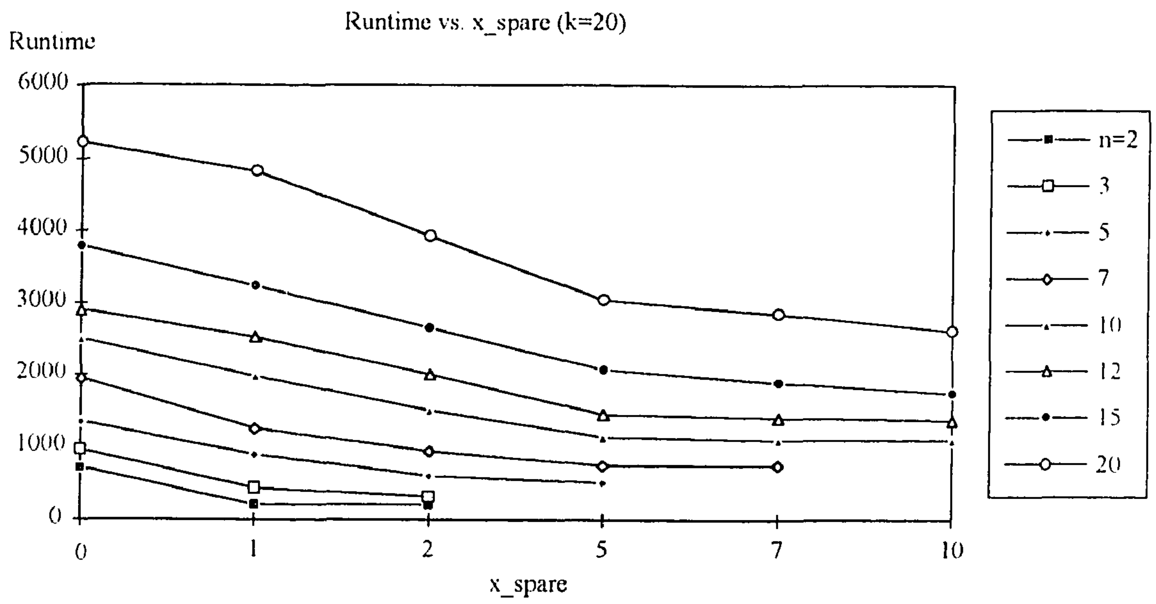

Figure 7.5 Worst case runtime vs. number of *x_spares* for various *n*

(*x_spare*) for the various $k(n)$. Figure 7.8 shows the number of each swap with respect to *x_spare*. As can be seen from the figure, the wild swap increases steadily with increases in *x_spare*, but other simple swaps and successive swaps decrease, which explains the improvement in runtime.

7.4 Discussion

In this chapter, a novel algorithm for routing in the fault tolerant Clos network has been introduced. Clos networks are used mainly to realize permutations. Without any fault tolerance, if a switch in the network fails, the network is rendered inoperative and the system has to be interrupted to put the network back to work. The FTC network can continue its work uninterrupted during the presence of a fault because the FTC network can reconfigure itself dynamically, by changing the settings of the multiplexers and demultiplexers and using the adaptive permutation translation scheme which can be facilitated by the use of the reconfiguration matrix R. The defective item can then be repaired during the time at which the system

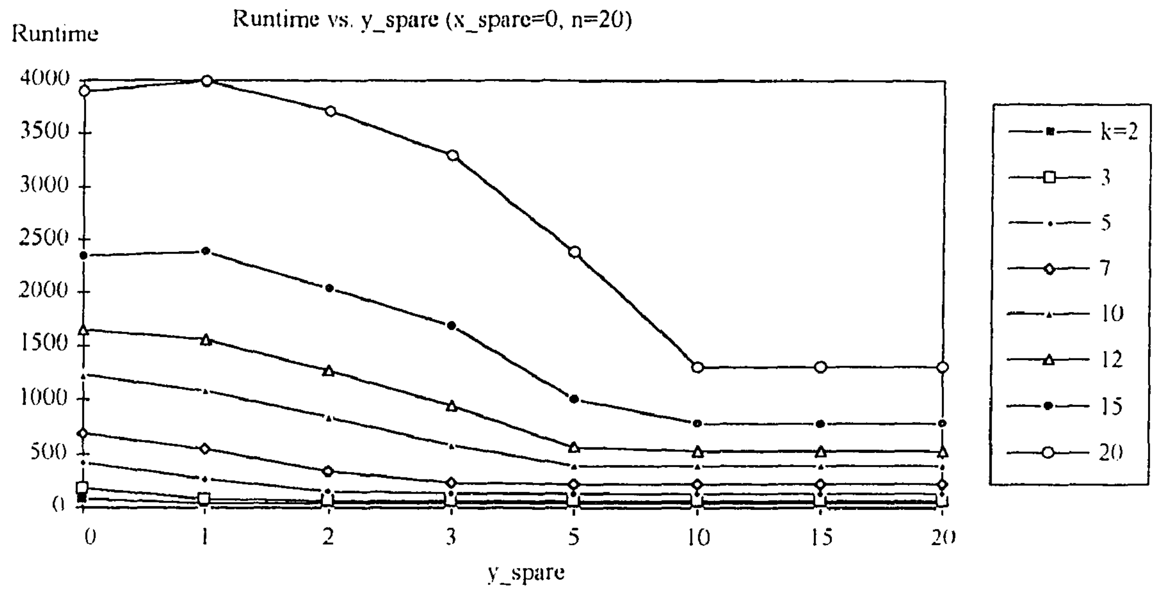

Figure 7.6 Average case runtime vs. number of *y_spares* for various *k*

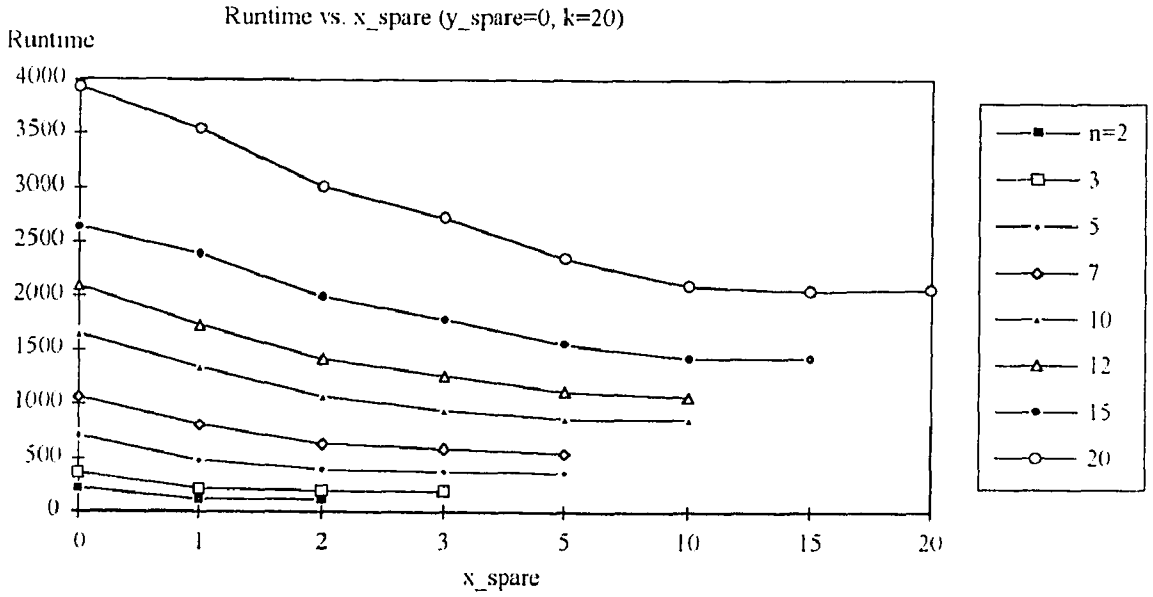

Figure 7.7 Average case runtime vs. number of *x_spares* for various *n*

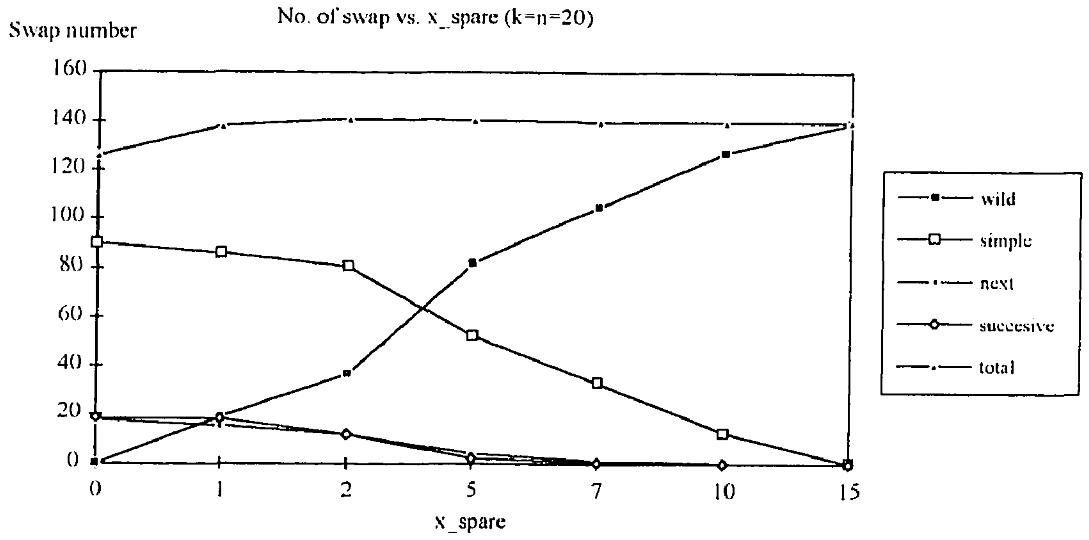

Figure 7.8 Number of simple, next simple, and successive swaps vs. *x_spares*

is unused. The spare switches introduce two types of wild cards depending on the location of spare switches in particular stages. In the Type I FTC network, two extra spares along with multiplexers/demultiplexers are required in order to create one additional row in the specification matrix. The Type II FTC network requires less hardware to create one additional column in the matrix, and the wild cards are much more flexible than in the Type I FTC network. In designing the routing algorithm, any wild cards can be used at any time during the decomposition. However, it is preferable to use the type I extra spares first and then type II spares next, since type II spares are more flexible. As in the previous algorithm, the new algorithm scans the C matrix row by row, and swapping elements are restricted to two columns for the successive swap, which gives the obvious advantage in proving that it works for all permutations. Another advantage to the new algorithm is that it has the potential to be run in parallel since only two columns are involved in the successive swap and other pairs of two columns can be swapped at the same time.

CHAPTER 8

RELIABILITY OF FAULT TOLERANT CLOS NETWORKS

8.1 Introduction

So far we have discussed the new routing algorithms in ordinary and fault tolerant Clos networks. Also, we considered the runtime with respect to the number of extra switches in the outer and middle stages. Another important factor in the FTC network is the reliability and space complexity with respect to the number of extra switches. The reliability and space complexity are dependant on the number of spare switches in the outer and middle stage switches, and these switches generate additional extra rows and columns in the specification matrix which contribute to the improvement in runtime. Thus, it is important to understand exactly how these factors are related, and design the FTC network accordingly. In section 8.2, the fault detection and location for the FTC network is discussed briefly. Next, the reliability and space complexity of the FTC network, which are important factors in designing the fault tolerant Clos networks, are considered. Finally, in the last section, the optimum number of extra switches for the fault tolerant Clos network is considered which will best balance the runtime, reliability and cost.

8.2 Fault Detection and Location of the FTC

The work of any fault tolerant MIN generally depends on two things: fault detection and fault location. Two techniques have been proposed in the literature for fault detection and location. First, fault detection and location can be performed off-line by applying prescribed test patterns to the inlets and comparing the output at the outlets with the expected values. Second, faults can be detected and located dynamically online through either parity checking or data bit checking. As good as the online techniques may sound, they require a special switch design with built-in hardware to carry out the dynamic checking. This online fault detection and location technique is the mechanism that can be applied to many MINs. However, the FTC network does not require any particular mechanism; rather it requires only that the processors be notified of the location of the fault, if any. For the work done in this thesis, it is assumed that there is some mechanism to detect and locate faults and notify the processors of the location of the fault.

8.3 Reliability of the FTC Network

The reliability of both the ordinary Clos network and the FTC network are dependent on the reliability of each switch and link of the networks. In chapter 5, multiplexers and demultiplexers are assumed to have high reliability when compared with switches and links in the FTC network. Rigorous reliability analysis is possible which considers the reliabilities of both multiplexers and demultiplexers. However, they are not considered in this thesis for analytical simplicity. First, define the reliability, r, of a single switch as the probability that the switch does not fail over a period of time τ. Then, $f = 1 - r$ is the probability that the switch fails in the same period τ. Similarly, define the reliability R of the network, ordinary or FTC, as the probability that the network does not fail over a period of time τ. Then $F = (1 - R)$ is the probability that the network fails in the same period τ. A switch fails if it cannot realize, partially or completely, a mapping of its inputs onto its outputs. Similarly, a network fails it cannot realize, partially or completely, a mapping of its inlets onto its outlets. For the ordinary Clos network to be fully operational over the period of time τ, all of its switches must be operational over the same period of time τ. For simplicity, assume that all the switches have the same reliability r. Therefore, the reliability of the ordinary network, assuming statistical independence (independent failure events), is

$$R_{ord} = r^{2k+m}$$

where $2k + m$ is the number of switches in the ordinary Clos network.

For the FTC with one extra switch in each stage, the network will remain fully operational if up to one switch in every stage fails. Let R_0, R_1 and R_2 be the reliabilities of stages 0, 1, and 2, respectively. Clearly, the three stages are statistically independent. Thus, the reliability of the network is

$$R_{FTC} = R_0 R_1 R_2$$

The reliability of the first stage, R_0 is the probability that at least k out of the $k + 1$ first stage switches, will be operational. Alternatively, if F_0 is the probability that the first stage fails, then

$$R_0 = 1 - F_0$$

For stage 0 to fail, given that there is one extra switch, at least two switches will have to fail, or less than k switches will have to function properly. This is a case of binomial distribution or Bernoulli trials, for which F_0 can be written as

$$F_0 = \sum_{i=0}^{k-1} \binom{k+1}{i} r^i f^{k+1-i} = \sum_{i=0}^{k-1} \binom{k+1}{i} r^i (1-r)^{k+1-i}$$

where $\binom{k+1}{i}$ is the combination of $k + 1$ taken i at a time. Substituting F_0 to $R_0 = 1 - F_0$ and realizing that $R_0 = R_2$ since the outer stages are the same, it follows that

$$R_0 = R_2 = 1 - \sum_{i=0}^{k-1} \binom{k+1}{i} r^i (1-r)^{k+1-i}$$

A similar analysis shows that the reliability of the middle stage is

$$R_1 = 1 - \sum_{i=0}^{m-1} \binom{m+1}{i} r^i (1-r)^{m+1-i}$$

Substituting these two equations yields,

$$R_{ftc} = \left(1 - \sum_{i=0}^{k-1} \binom{k+1}{i} r^i (1-r)^{k+1-i}\right)^2 \left(1 - \sum_{i=0}^{m-1} \binom{m+1}{i} r^i (1-r)^{m+1-i}\right)$$

When more than one switch is added to every stage, additional alternative paths are created and thus, greater reliability is expected. To verify that, the previous equation will be generalized to the case where x switches are added to each of stages 0 and 2. and y switches are added to stage 1. Using the same procedure as above. it can be shown that the reliability of the new network, R_{FTC} is

$$R_{FTC} = \left(1 - \sum_{i=0}^{k-1} \binom{k+x}{i} r^i (1-r)^{k+x-i}\right)^2 \left(1 - \sum_{i=0}^{m-1} \binom{m+y}{i} r^i (1-r)^{m+y-i}\right)$$

8.4 Effect of Spare Numbers on Reliability

The above equation can be used to show the reliability of a fault tolerant Clos network with respect to the number of spares switches x or y. Figure 8.1 shows the reliability of a fault tolerant Clos network with respect to the number of extra switches in the first or third stage in the Type I FTC network, when the reliability of the switch r is 0.9, 0.96, 0.98, and 0.99, respectively, and $n = k = 20$.

As shown in the figure, the reliability of the system depends on the number of extra switches. and just one or two extra switches are needed in each stage in order to improve the reliability of the system considerably especially when r is high. The high system reliability can be obtained as the reliability of the switch r increases. It can be seen that if r is large, the addition of more than one switch per stage is not needed and the reliability approaches 1. However, when the reliability of the

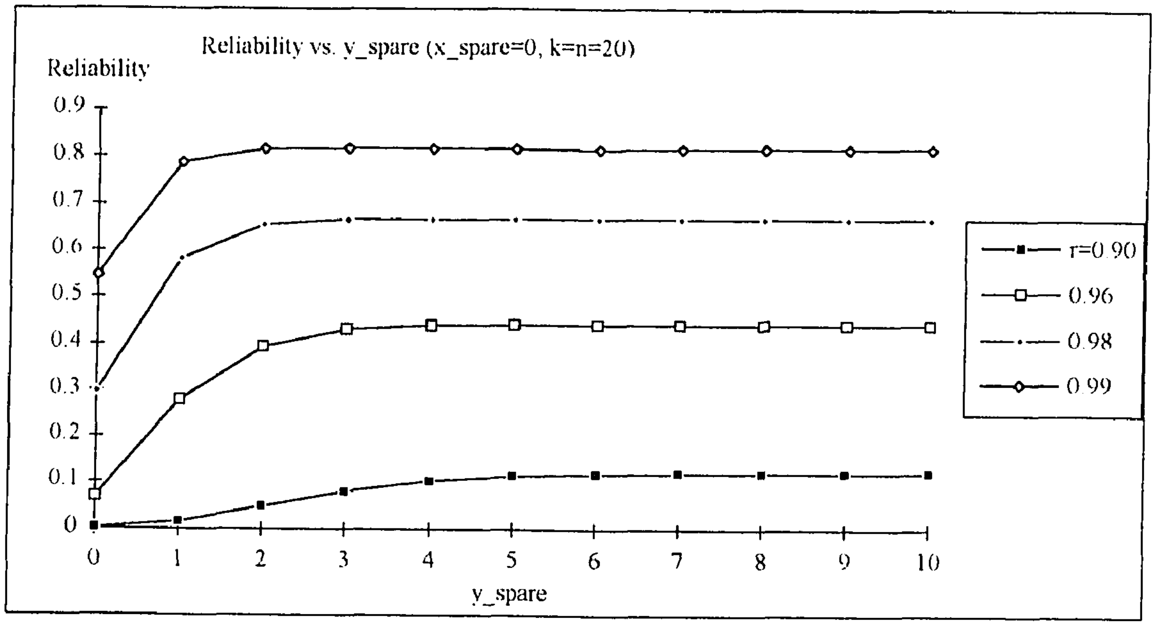

Figure 8.1 Reliability vs. number of *y_spare* switches in Type I networks when $k = n = 20$

switch r is low or when switches with high reliability are used for a long time, the system reliability increases slowly with respect to the number of spare switches. In this case, more switches are needed in order to obtain the better reliability of the system. Also, the relatively low system reliability is obtained when r is low.

Figure 8.2 shows the reliability of a fault tolerant Clos network with respect to the number of extra switches in the second stage in the Type II FTC network when the reliability of the switch r is 0.9, 0.96, 0.98, and 0.99, respectively, and $n = k = 20$. As in the Type I network, the reliability of the system depends on the number of extra switches, and just one or two extra switches are needed to improve the reliability of the system considerably when r is high. The high system reliability can be achieved as the reliability of the switch r increases, but it takes on a lower value than in Type I networks for the same reliability of the switch r. This is true when the reliability of the switch is low, where the system reliability increases slowly with respect to the number of spare switches, but with a much lower value. The

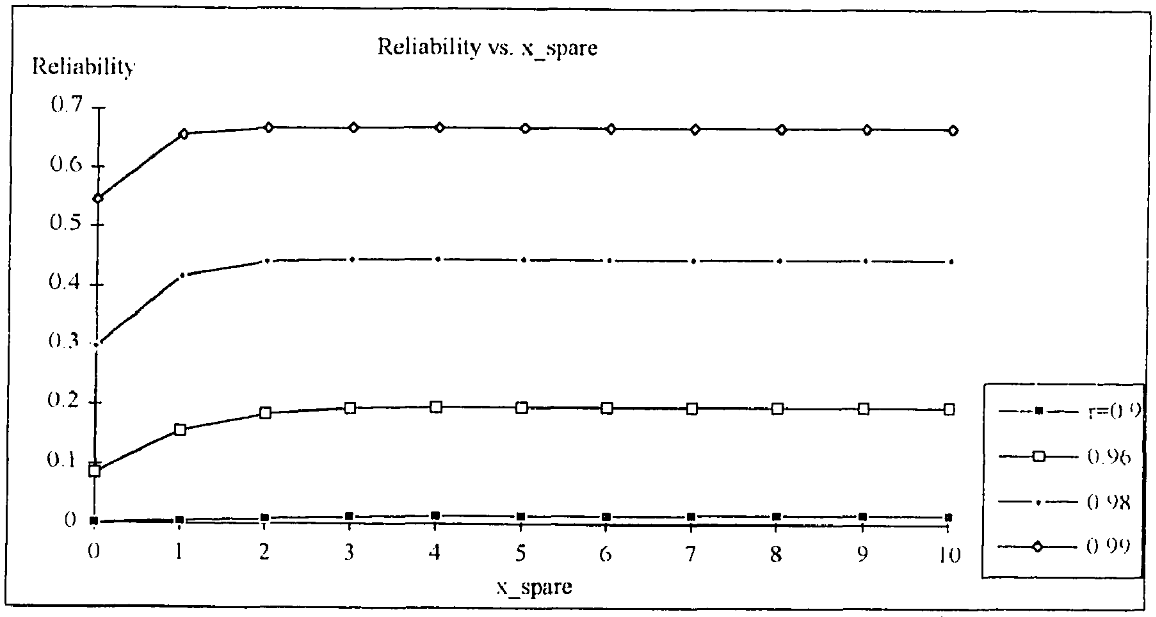

Figure 8.2 Reliability vs. number of *x_spare* switches in Type II networks when $k = n = 20$

main reason for this is that the Type I network has extra switches in both the first and third stages, while in Type II networks the extra switches are available only in the second stage, so the total number of extra switches is about the half that of the Type I network. Note, in the above two figures, that the network reduces to an ordinary Clos network and the reliability is same in both types of network when $x\_spare = y\_spare = 0$. Generally, the addition of extra switches increases the overall reliability of the network by orders of magnitude when the reliability r is low, while the addition of same number of switches increases the overall reliability of the network only slightly when r is high.

Therefore, it can be concluded that when the reliability r of the individual switches is high, there is no need for adding excessive hardware, especially when the total number of switches is small. That is because the higher the number of switches in the network, the higher its vulnerability to failure. The existence of small numbers of switches with a few extra switches in the FTC makes a failure in the network insignificant. Adding more switches per stage can be seen to increase the overall reliability of the network. However, reconfiguration of the network would be more difficult and time consuming. Moreover, the extra switches would increase the hardware of the network and complicate its design. The reliability of the FTC is generally greatly higher than that of the ordinary network and the FTC is more beneficial for networks with poor switch reliabilities. When $r = 1$, there is clearly no need for any fault tolerance.

8.5 Space Complexity of the FTC Network

We will now consider the space complexity of the FTC network. The addition of the extra switches in the first and third stages causes an increase in the number of inputs and outputs in each of the second stage switches. This is the same as when extra switches are added in the second stage, which results in the increase of the number of inputs and outputs in the first and third stage switches. Note that the addition of spare switches in the second stage results in the increase of switch areas in both the first and third stages, while the changes in the outer stages result in an increase only in the second stage. Since the switches are actually crossbar switches, the area of the switches, or the number of cross points is generally proportional to the product of the number of inputs and outputs of the switch. Here, we assume that the area of multiplexers/demultiplexers are not significant for the simplicity of analysis. Also, it is assumed here that the costs for the FTC networks are proportional to the area of the total number of switches.

Let x and y again be the number of extra switches in the second stage and first (or third) stage, respectively. Then the total number of switches in the second stage is $n + x$, while it is $k + y$ in the first (or third) stage. The number of inlets in the first stage switch is n, and the number of outlets is $n + x$. In the second stage switch, the number of inlets or outlets is $k + y$. Since the first and third

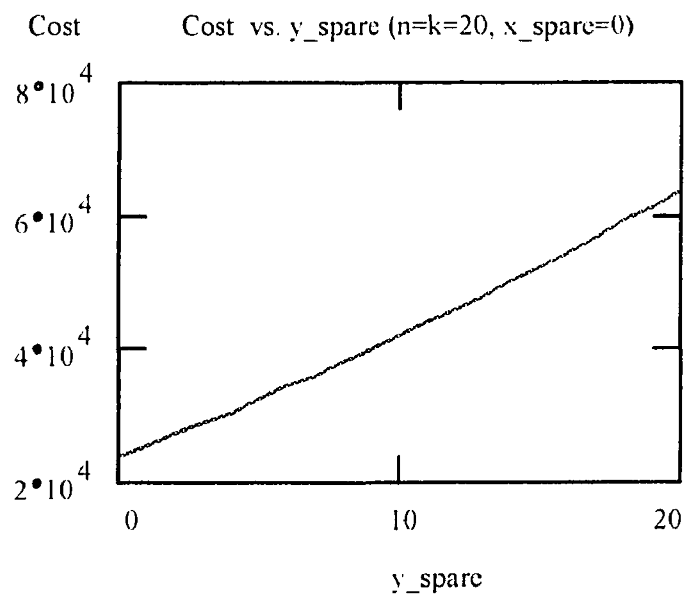

Figure 8.3 Cost vs. number of spare switches in Type I networks when $n = k = 20$ and $x = 0$

stages are identical, the total area of the outer stages is twice the area of either outer stage. The space complexity or the cost of the FTC network is proportional to $2(k+y)n(n+x)+(n+x)(k+y)^2$. Figure 8.3 shows the cost vs. the number of extra y_spares in Type I networks when $n = k = 20$ and $x\_spare = 0$. As can be seen in the figure, the cost increases monotonically as the number of y_spare increases.

Figure 8.4 shows the cost vs. number of x_spares in Type II networks when $n = k = 20$ and $y\_spare = 0$. As in Type I networks, the cost also increase steadily with the increase in x_spare. However, in this case, the cost is less than in Type I networks. Note that the increase in y_spare in Type I networks actually adds twice the number of extra switches to the network, although the number of extra switches is the number of x_spare in Type II networks. It can be seen from the figures that the Type I network is in general more expensive than Type II networks, but does not double the cost of the Type II networks for the same number of x_spare and

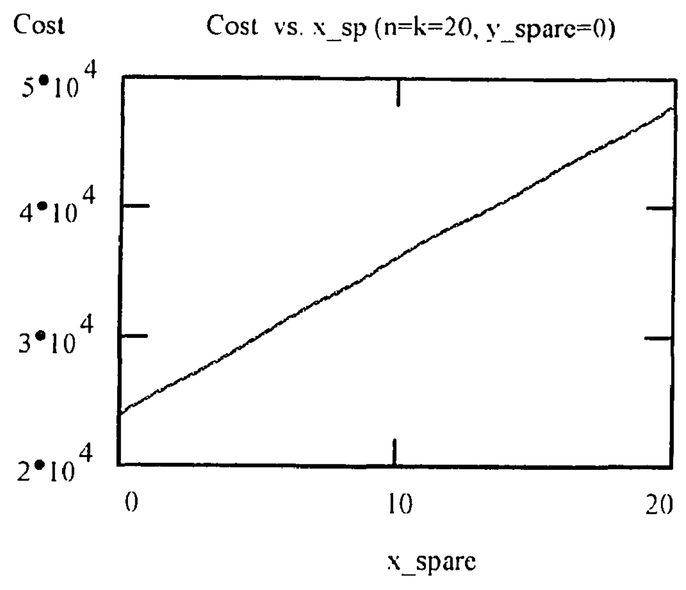

Figure 8.4 Cost vs. number of spare switches in Type II networks when $n = k = 20$ and $y = 0$

y_spare. However, it can achieve better reliability than the Type II network since there are more extra switches.

8.6 Optimum Number of Spare Switches in the FTC Network

So far we have examined the runtime, reliability, and cost with respect to the number of spare switches x, and y. As was seen in chapter 7, the runtime is roughly the same in both the Type I and Type II FTC networks. More specifically, the Type II network is faster when the number of spare switches is small. However, as the number of spares increases, the runtime is slower than the Type I network since it needs extra time to find the location of spares and to make sure that there are no identical elements in the specification matrix S. The number of spares needed in Type I networks for generating additional rows in the S matrix is twice the number of spares in the Type II network for creating the same number of additional columns. On the other hand, the Type I network can achieve better reliability than the Type II network, but it requires twice the number of extra switches. Because of this, the Type I network is more expensive than the Type II network, but it does not double the cost of the Type II network when *x_spare* and *y_spare* are the same. The optimum number of spare switches in each stage of the FTC network can not be determined exactly, rather it depends on the availability of the resources and requirements of the system. The general approach would be to decide the above factors first and then adjust the number of spare switches in the outer stages and in the middle stage.

The research so far has shown the following results for determining the number of spare switches in each stage. When the reliability of the switches is high, just one or two extra switches are needed in each stage in order to improve the reliability of the system. In this case, the fault tolerant routing algorithm is not efficient, and the runtime approaches the speed of an algorithm for the ordinary Clos network. No additional costs are required. However, when the reliability of the switches are not high, more than two extra switches are required in order to improve the reliability. High reliability can be achieved by adding more switches in outer stage, but with the increase in cost. Adding more extra switches in the middle stage is less costly in improving the run time than adding spares in the outer stages. However, better reliabilities are possible in the latter case. In both cases, the introduced fault tolerant routing algorithm utilizes extra switches to improve the runtime, which is roughly the same in both types of FTC.

8.7 Discussion

Besides the fault tolerance the FTC provides, the reliability of the network is greatly enhanced. High reliability means more system availability with uninterrupted operations. It is seen from the analysis that using this fault tolerance approach is most beneficial when the reliability of the individual switches is poor, or the number of switches in the network is large. As far as reliability is concerned, larger numbers of extra switches are needed in order to increase the reliability. This number depends on the number of switches in the network and the reliability of the individual switch. and can be determined for an optimum value. However, putting a large number of extra switches per stage adds significantly to the network hardware and routing complexity. High reliability can be achieved by adding more switches in any of the stages. But adding switches in the outer stage increases the cost and system hardware more rapidly. The same improvements in runtime can be obtained by adding more extra switches in the middle stage, which is less costly in improving the run time, but relatively low improvements can be achieved in reliability.

CHAPTER 9

CONCLUSION

9.1 Summary

This thesis has demonstrated the failure of Gordon's algorithm which uses two matrices for improving the runtime. A new simple algorithm for the control of rearrangeable Clos networks which runs in time $O(nk^{3/2})$ is proposed based on his algorithm. The new algorithm is extended to the fault tolerant Clos (FTC) network. which can further improve the run time when there are relatively few or no faults in the system. In order to achieve this. the FTC network has been classified into three types to find the swapping rules and conditions of extra elements. The optimum number of extra switches on the fault tolerant Clos network is considered which will best satisfy the run time, reliability and cost constraints. The result of each perspectives are summarized below.

9.1.1 Routing for Clos Networks

Although Gordon's algorithm is simple and fast. this research has shown that his algorithm displays errors in some of the permutations. especially when $k > 5$. The new algorithm is based on Gordon's algorithm where the Clos network is represented by the specification matrix and count matrix. As in Gordon's algorithm. the new algorithm has the advantage of speeding up routings by just checking the C matrix in order to calculate the number of occurrences of each element in each column of the S matrix. The new algorithm consists of three kinds of swaps: simple swap. next simple swap. and successive swap. The successive swap can be compared with the iteration phase of Neiman's algorithm. where the algorithm backtracks in order to select all elements which are not in the same rows and same columns. The time complexity of the new algorithm for the ordinary Clos network has been found to be $O(nk^{3/2})$.

The basic difference between Gordon's algorithm and new algorithm lies in the scanning direction in the C matrix. In Gordon's algorithm, it is scanned column by column, removing columns once all elements are nonidentical in each column. Swapping elements can take place between a not-yet-decomposed leftmost column and the rest of the columns. However, the new algorithm scans the C matrix row-by-row, and swapping elements are restricted to two columns for the successive swap. This gives an obvious advantage in proving that it works for all permutations, but, in Gordon's algorithm, this is hard to prove. Another advantage to the new algorithm is that it has the potential to be run in parallel since only two columns are involved in the successive swap, and other pairs of columns can be swapped at the same time. Also, the simple, next simple, and successive swaps can easily be extended to the fault tolerant Clos network, which is yet another advantage.

9.1.2 Routing for FTC Networks

The new algorithm for FTC networks shows that the previous algorithm for the ordinary Clos network can be easily extended to the fault-tolerant cases. It has been shown that the original matrices can be modified using extra rows and columns in the specification matrix so that they can represent the extra spare switches in the FTC network. Extra switches generates wild cards in the matrix, which provide flexibility during the decomposition process. The wild swaps employed in the algorithm, in addition to three kinds of swaps in the ordinary case, were found to be important since they can reduce the chances of entering into the time-consuming next simple swaps or successive swaps. The spare switches introduce two types of wild cards depending on the location of spare switches in the network. In Type I networks, two extra spares along with multiplexers/demultiplexers are required in order to create one additional row in the specification matrix. The Type II network requires less hardware to create one additional column in the matrix, and the wild cards are much more flexible than in the Type I network. It was shown that the addition of extra switches to the network considerably decreases the runtime of the algorithm. The failure in the switch is reflected in the reconfiguration matrix, which helps to reconfigure the network dynamically by changing the settings of the multiplexers and demultiplexers and using extra switches. As in the ordinary Clos network, the new algorithm realizes every permutation because of its scanning the C matrix row-by-row and restricting swapping elements to two columns in the successive swap.

9.1.3 Optimum Numbers of Spare Switches in FTC

Optimum numbers of extra switches in FTC networks can be determined with respect to the reliability, runtime and cost. The research so far has shown the following results for determining the number of spare switches in each stages. When the reliability of the switches is high, just one or two extra switches are needed in each stage in order to improve the reliability of the system. In this case, the fault tolerant routing algorithm is not efficient, and the runtime approaches the speed of the algorithm for the ordinary Clos network. No additional costs are required. However, when the reliability of the switches is not high, more than two extra switches are required in order to improve the reliability. High reliability can be achieved by adding more switches in outer stages, but with an increase in cost. Adding more extra switches in the middle stage is less costly in improving the run time than adding spares in the outer stages. However, better reliabilities are possible in the latter case. The runtime is improved by roughly the same amount in both types of FTC networks. The optimum number of spare switches in each stage of the FTC network can not be determined exactly, but rather it depends on the availability of the resources and requirements of the system. The general approach would be to decide the above factors first and then adjust the number of spare switches in the outer stages and in the middle stage.

9.2 Open Problems

This research has covered the routing issues in the ordinary as well as fault tolerant Clos networks in depth. In spite of the progress made in some areas, some problems have been observed and some encouraging ideas that need further research have been discovered. Those will altogether contribute to establish the sound bases of the research by continuing the study more deeply. First, the current algorithm for decomposing the Clos network requires that no identical elements be present in a column of S except spares in order to completely decompose the matrix. This condition is due to the structure of the Clos network in which each of the second stage switches is connected to every third-stage switch. Also, swaps are allowed only for the elements in the same row. This is also due to the first-stage switches' connection to each of the second-stage switches. These conditions look straightforward, but in fact, they requires extremely serial decomposition and frequent backtracking. However, by modifying the Clos network somehow, the current conditions might be alleviated in a way that could lead to a much faster, straightforward routing strategy. The question here is: Is there any modified structure of the Clos network which could lead to the much faster routing that can be performed in a serial as well as in a parallel method? And if so, how can we find that structure, and how much difference can we expect?

Meanwhile, this research has developed a new algorithm for decomposing the Clos interconnection network. This algorithm can be applied to Benes and other similar interconnection networks which are derived from the Clos network. Then, can we apply this algorithm to other multistage interconnection networks such as the shuffle-exchange, banyan, and omega networks? Also, can the algorithm for the

FTC network be applied to other fault tolerant interconnection networks such as the ESC?

Also, the newly introduced algorithm decomposes the specification matrix row by row, while Gordon's algorithm decomposes it column by column. The potential advantage of decomposing the matrix column by column is the reduction of the dimension of the specification matrix as the routing progresses, since each decomposed column can be removed from the matrix. Gordon's algorithm has been demonstrated to display errors for some permutations, but can we explain why his algorithm fails? Also, can we really find an algorithm which decomposes the matrix in column by column bases?

This thesis assumed that the FTC has an ability for the detection and location of faults. Further research is required in this area. In addition, another study needs to be performed on the reconfiguration problems due to the failure of interstage connections, and the analysis of the time complexity of new algorithms.

REFERENCES

1. J. Baer, "Multiprocessing Systems," *IEEE Transactions on Computers*, vol. C-25, no. 12, pp. 613-641, December 1976.
2. L. Bhuyan, "A Combinatorial Analysis of Multibus Multiprocessors," *Proceedings of 1984 International Conference on Parallel Processing*, pp. 225-227, August 1984.
3. H. Lorin, *Parallelism in Hardware and Software*, Prentice-Hall, Englewood Cliffs, NJ, 1972.
4. M. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54, pp. 1901-1909, December 1966.
5. M. Flynn, "Some Computer Organization and Their Effectiveness," *IEEE Transactions on Computers*, C-21, no. 9, pp. 948-960, September 1972.
6. W. Handler, "The Impact of Classification Schemes on Computer Architecture," *Proceedings of 1977 International Conference on Parallel Processing*, pp. 7-15, 1977.
7. W. Davis, *Operating Systems: A Systematic View*, 2nd Edition, AddisonWesley, Reading, MA, 1983.
8. M. Mano, *Computer System Architecture*, 2nd edition, Prentice-Hall, Englewood Cliffs, NJ, 1982.
9. T. Hallin and M. Flynn, "Pipelining of Arithmetic Functions," *IEEE Transactions on Computers*, vol. C-21, no. 8, pp. 880-886, August 1972.
10. T. Mudge et. al., "Analysis of Multiple Bus Interconnection Networks," *Proceedings of the 1984 International Conference on Parallel Processing*, pp. 228-232, August 1984.
11. T. Mudge, J. Hayes and D. Winsor, "Multiple Bus Architectures," *Computer*, pp. 42-48, June 1987.
12. T. Chen, "Parallelism, Pipelining, and Computer Efficiency," *Computer Design*, pp. 69-74, vol. 10, no. 1, January 1971.
13. P. Wayner, "Processor Pipelines," *Byte*, vol. 17, pp. 305-306, January 1992.
14. D. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers*, vol. 24, no. 12, pp. 1145-1155, December 1975.
15. W. Chu, "Advances in Computer Communications and Networking," Artech House, Dedham, MA, 1979.

16. Chuan-lin Wu and Tse-Yun Feng, "On a Class of Multistage Interconnection Networks," *IEEE Transactions on Computers*, vol. C-29, no. 8, pp. 694-702, August 1980.

17. T. Feng, "A Survey of Interconnection Networks," *Computer*, vol. 14, no. 12, pp. 12-27, December 1981.

18. H. Siegel, "Interconnection Networks for SIMD Machines," *Computer*, vol. 12, pp. 57-65, June 1979.

19. Yao-Ming Yeh and Tse-Yun Feng, "On a Class of Rearrangeable Networks," *IEEE Transactions on Computers*, vol. 41, no. 11, pp. 1361-1379, November 1992.

20. F. K. Hwang, "On the Rearrangeability of Some Multistage Connecting Networks," *Bell Systems Technical Journal*, vol. 55, No. 9, pp. 1411-1422, November 1976.

21. F. K. Hwang and A. Jajszczyk, "On Nonblocking Multiconnection Network," *IEEE Transactions on Communications*, vol. COM-34, no. 10, pp. 1038-1041, October 1986.

22. B. Douglass, "Rearrangeable Three-Stage Interconnection Networks and Their Routing Properties," *IEEE Transactions on Computers*, vol. 42, no. 5, pp. 559-567, May 1993.

23. G. Goke and G. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," *First Annual Symposium on Computer Architecture*, pp. 21-28, December 1973.

24. M. Leland, "On the Power of the Augmented Data Manupulator Network," *1985 International Conference on Parallel Processing*, pp. 74-78, August 1985.

25. K. Batcher, "The Flip Network in STARAN," *Proceedings of the 1976 International Conference on Parallel Processing*, pp. 65-71, 1976.

26. H. Siegel and R. McMillen, "The Multistage Cube: A Versatile Interconnection Network," *Computer*, pp. 65-76, December, 1981.

27. F. Lombardi and C. Feng, "Detection and Location of Multiple Faults in Baseline Interconnection Networks," *IEEE Transactions on Computers*, vol. 41, pp. 1340-1344, October 1992.

28. M. Kumar and J. R. Jump, "Generalized Delta Networks," *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 10-18, August 1983.

29. Z. Cvetanovic, "Best and Worst Mapping for the Omega Network," *IBM Journal of Research and Development*, vol. 31, pp. 452-463, July 1987.

30. D. Rau. J. Fortes and H. Siegel, "Destination Tag Routing Techniques Based on a State Model for the IADM Network." *IEEE Transactions on Computers*, vol. 41, no. 3, pp. 274-285. March 1992.

31. Charles Clos, "Study of Non-blocking Switching Networks." *Bell Systems Technical Journal*, vol. 32, no. 2, pp. 406-424, March 1953.

32. V. E. Benes, "On Rearrangeable Three-Stage Connecting Network." *Bell Systems Technical Journal*, vol. XLI. no. 5, pp. 117-125, September 1962.

33. V. I. Neiman. "Structure et commande optimales des réseaux de connexion sans blocage." *Annales des Telecommun.*, pp. 232-238, July/August 1969.

34. Nelson T. Tsao-Wu. "On Neiman's Algorithm for the Control of Rearrangeable Switching Networks," *IEEE Transactions on Communications*, vol. COM-22. no. 6. pp. 737-742, June 1974.

35. Abraham Waksman. "A Permutation Network." *Journal of the ACM.* vol. 15. no. 1. pp. 159-163. January 1968.

36. H. R. Ramanujam, "Decomposition of Permutation Networks." *IEEE Transactions on Computers*, vol. C-22. no. 7. pp. 639-643, July 1973.

37. Marek Kubale, "Comments on Decomposition of Permutation Networks." vol. C-31. no. 3, p. 265, March 1982.

38. Andrzej Jajszczyk, "A Simple Algorithm for the Control of Rearrangeable Switching Networks." *IEEE Transactions on Communications*. vol. COM-33, no. 2, pp. 169-171, February 1985.

39. Claude Cardot. "Comments on a Simple Algorithm for the Control of Rearrangeable Switching Networks." *IEEE Transactions on Communications*. vol. COM-34. no. 4, p. 395. April 1986.

40. Frank K. Hwang, "Control Algorithms for Rearrangeable Clos Networks," *IEEE Transactions on Communications*, vol. COM-31, no. 8. pp. 952-954. August 1983.

41. D. C. Opferman and N. T. Tsao-Wu, "On a Class of Rearrangeable Switching networks. Part I: Control Algorithm." *Bell Systems Technical Journal.* vol. 50. no. 5. pp. 1579-1600, May-June 1971.

42. Steinar Andresen. "The Looping Algorithm Extended to Base 2^t." *IEEE Transactions on Communications.* vol. COM-25. no. 10. pp. 197-203. October 1977.

43. J. Gordon and S. Srikanthan, "Novel Algorithm for Clos-Type Networks." *Electronic Letters*, vol. 26, no. 21, pp. 1772-1774. October 1990.

44. Y. K. Chiu and W. C. Siu, "Comment: Novel Algorithm for Clos-Type Networks," *Electronic Letters*, vol. 27, no. 6, pp. 524-526, March 1991.

45. Harold Gabow, "Using Euler Partitions to Edge Color Bipartite Graphs and Multigraphs," *International Journal of Computer and Information Sciences*, vol. 5, no. 4, pp. 345-355, 1976.

46. H. Gabow and Oded Kariv, "Algorithm for Edge Coloring Bipartite Graphs and Multigraphs," *SIAM Journal on Computing*, vol. 11, no. 1, pp. 117-129, February 1982.

47. Richard Cole and John Hopcroft, "On Edge Coloring Bipartite Graphs," *SIAM Journal on Computing*, vol. 11, no. 3, pp. 540-546, August 1982.

48. V. Vizing, "On an Estimate of the Chromatic Class of a p-graph," *Diskret. Analiz.*, no. 3, pp. 25-30, 1964.

49. D. Nassimi and S. Sahni, "A Self-routing Benes Network and Parallel Permutation Algorithms," *IEEE Transactions on Computers*, vol. C-30, no. 5, pp. 332-340, May 1981.

50. John D. Carpinelli and A. Yavuz Oruc, "A Non-backtracking Decomposition Algorithm for Routing on Clos Networks," *IEEE Transactions on Communications*, vol. 41, no. 8, pp. 1245-1251, August 1993.

51. J. Carpinelli, *Interconnection Networks: Improved Routing Methods for Clos and Benes Networks*, Ph.D. Thesis, Rensselaer Polytechnic Institute, Troy, NY, August 1987.

52. G. Lev, N. Pippenger and L. Valiant, "A Fast Parallel Algorithm for Routing in Permutation Networks," *IEEE Transactions on Computers*, vol. C-30, no. 2, pp. 93-100, February 1981

53. J. Lenfant, "Parallel Permutations of Data: A Benes Network Control Algorithm for Frequently Used Permutations," *IEEE Transactions on Computers*, vol. 27, no. 7, pp. 637-647, July 1978.

54. B. G. Douglass and A. Y. Oruc, "On Self-Routing in Clos Connection Networks," *IEEE Transactions on Communications*, vol. 41, no. 1, pp. 121-124, January 1993.

55. C. Raghavendra and R. Boppana, "On Self-Routing in Benes and Shuffle-Exchange Networks," *IEEE Transactions on Computers*, vol. 40, no. 9, pp. 1057-1064, September 1991.

56. G. Adams and H. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," *IEEE Transactions on Computers*, vol. C-31, no. 5, pp. 443-454, May 1982.

57. G. Adams, D. Agrawal and H. Siegel. "A Survey and Comparison of Fault-tolerant Multistage Interconnection Networks." *Computer*. pp. 14-27. June 1987.

58. K. Yoon and W. Hegazy, "The Extra Stage Gamma Network," *Proceedings of the 13th Annual Symposium on Computer Architecture*. pp. 175-182, 1986

59. K. Padmanabhan and D. Lawrie, "A Class of Redundant Path Multistage Interconnection Networks," *IEEE Transactions of Computers*. pp. 1099-1108. December 1983.

60. H. Nassar. *Fault-Tolerant Interconnection Networks for Multiprocessor Systems*. Ph.D. Thesis, New Jersey Institute of Technology, Newark, NJ, 1989.

61. C. Raghavendra and A. Varma, "INDRA: A Class of Interconnection Networks with Redundant Paths," *Proceedings of the 1984 Real Time Systems Symposium*, pp. 153-164. 1984.

62. T. Feng and C. Wu. "Fault-Diagnosis for a Class of Multistage Interconnection Networks," *IEEE Transactions on Computers* , vol. C-30. no. 10. pp. 743-758, October 1981.

63. D. Agrawal , "Testing and Fault Tolerance of Multistage Interconnection Networks," *Computer*, pp. 41-53, April 1982.

64. J. Lilienkamp, D. Lawrie and P. Yew, "A Fault Tolerant Interconnection Network Using Error Correcting Codes." *The Proceedings of the 1982 International Conference on Parallel Processing*. pp. 123-125. 1982.

65. D. Agrawal and D. Kaur. "Fault Tolerant Capabilities of Redundant Multistage Interconnection Networks." *The Proceedings of Real-time Systems Symposium*. pp. 119-127, December 1983.

66. J. P. Shen, "*Fault Tolerance of β-networks in Interconnected Multicomputer System*. Ph.D. Dissertation, Department of Electrical Engineering, University of Southern California, CA. August 1981.

67. W. Fuchs. J. Abraham and K. Huang, "Current Error Detection in VLSI Interconnection Networks," *The Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 309-315. 1983.