KING SAUD UNIVERSITY
College of Computer& Information Sciences
Department of Computer Engineering

INTERNAL BLOCKING IN FAULT TOLERANT MULTISTAGE INTERCONNECTION NETWORKS (MINs) IN ALL OPTICAL NETWORKS

PREPARED BY

Sultan Mohammed Marei

SUPERVISED BY

Dr. Abdulaziz Almazyad

Dr. Mohammed Amer Arafah

Thesis submitted in partial fulfillment of the requirements
for the degree of Master in the Department of Computer Engineering
At the College of Computer & Information Sciences,
King Saud University

Riyadh
Kingdom of Saudi Arabia,

3/1425 H
5/2004 G

Chapter 1

Introduction

Recently, significant developments have been made in photonic switching based on Wavelength Division Multiplexing (WDM). Therefore, researchers have used this technology to implement switches such as the Crossbar or Multistage Interconnection Networks (MINs) in order to upgrade the performance of the available systems that use these networks. Although the crossbar does provide a powerful communication capability, it is not economically feasible as the number of inputs/outputs becomes large. Moreover, due to the random nature of input arrivals, network utilization is much less than its capacity.

MINs are used because they are less expensive, easy to control, have low delay and support large scale of inputs/outputs. One of applications of MINs is in processor to memory communication in a parallel multiprocessing systems, in which, they allow a direct link between any processor to any memory module so the processor can access any memory module with a very small number of communications or accessing conflict.

1.1 Interconnection Networks

There are two famous classes of interconnection networks: Crossbar switch and Multistage Interconnection Networks.

1.1.1 Crossbar Switches

An N x N crossbar switch is a single-stage, strictly non-blocking network with N input ports and N output ports. It can realize N! permutations (any one-to-one mapping between the set of inputs and the set of outputs). Therefore it does not suffer from internal blocking. It can be implemented by photonic switching fabrics based on Guided-Wave Devices such as Directional Couplers, Semiconductor Optical Amplifiers (SOA), and Spatial Light Modulators (SLM).

The strength of Guided-Wave Fabrics is that they cannot sense the presence of packets that are passing through them. Therefore, once the control unit of a switch sets a path between an input port and an output port, high-speed data, multiplexed speech, or video can be transferred through them. However, the limitation of these switches is that since they cannot sense the presence of individual bits that are passing through them, they cannot read the packet header, and cannot configure their state accordingly. The first crossbar switch is based on a directional coupler [7] [6], which has two optical input ports and two optical output ports. In addition, it has an electronic control input, which is capable to configure the switch into two states. The first state is "straight" which is illustrated

in Fig. 1a where the upper input is directed to the upper and the lower input is directed to the lower output, and the second state is “Exchange” which is illustrated in Fig. 1b where the upper input is directed to the lower output and the lower input is directed to the upper output. The main advantage of this switch is its ability to deliver information in high bit-rate. However, the reconfiguration rate is limited by the electronic control input.

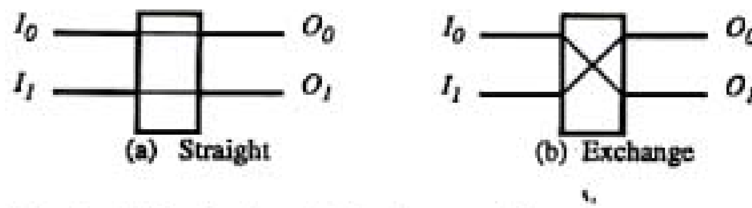


Fig. 1 A directional coupler in its two configurations

The implementation of a 4x4 crossbar switch based on 2x2 directional couplers is illustrated in Fig. 2.

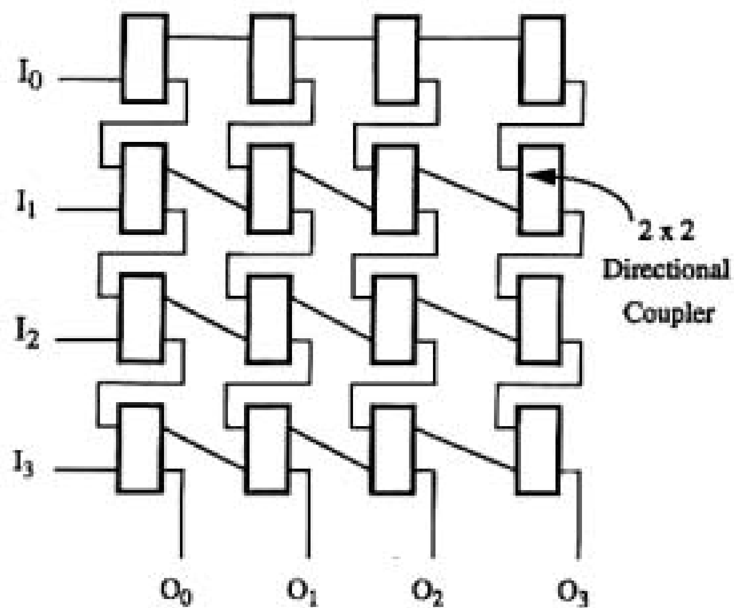


Fig. 2 A 4x4 crossbar switch based on directional couplers

However, the hardware complexity of an N x N crossbar is O (N^2) since it consists of N^2 cross points. Therefore, it is expensive and only appropriate for small switches, say, with N ≤16.

1.1.2 Multistage Interconnection Networks

The other class of interconnection networks is the Multistage Interconnection Networks (MINs), which consist of a few stages of a number of smaller switch elements as shown in Fig. 3.

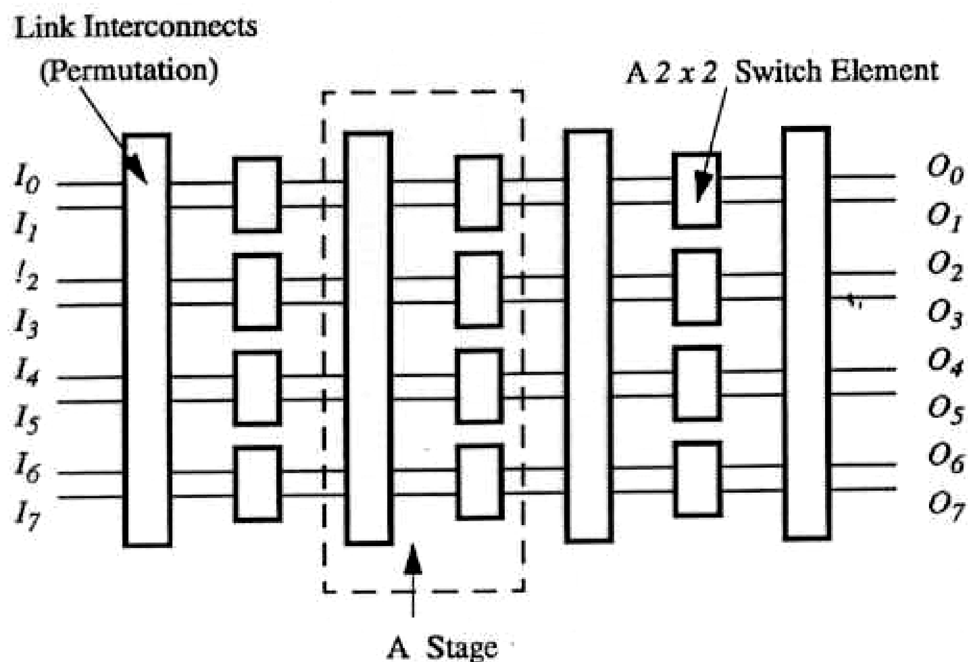


Fig. 3 A structure of multistage interconnection network.

The charachtersitics of MINs are :

a) Full Access vs. Dynamic Full Access Capability

In Full Access (FA), every input link can reach any output link in a single pass, and in Dynamic Full Access (DFA), every input can reach any output link in a finite number of passes through the network. In

multistage interconnection networks, by using 2 x 2 switch elements, only $\log_2 N$ stages are required to achieve full access capability. Fig. 4 illustrates a single stage shuffle-exchange network, which has only dynamic access capability. For example, input channel I_0 can reach to output channel O_7 in three passes. The full access networks are faster but more expensive. In this research, only full access multistage interconnection networks will be considered [1].

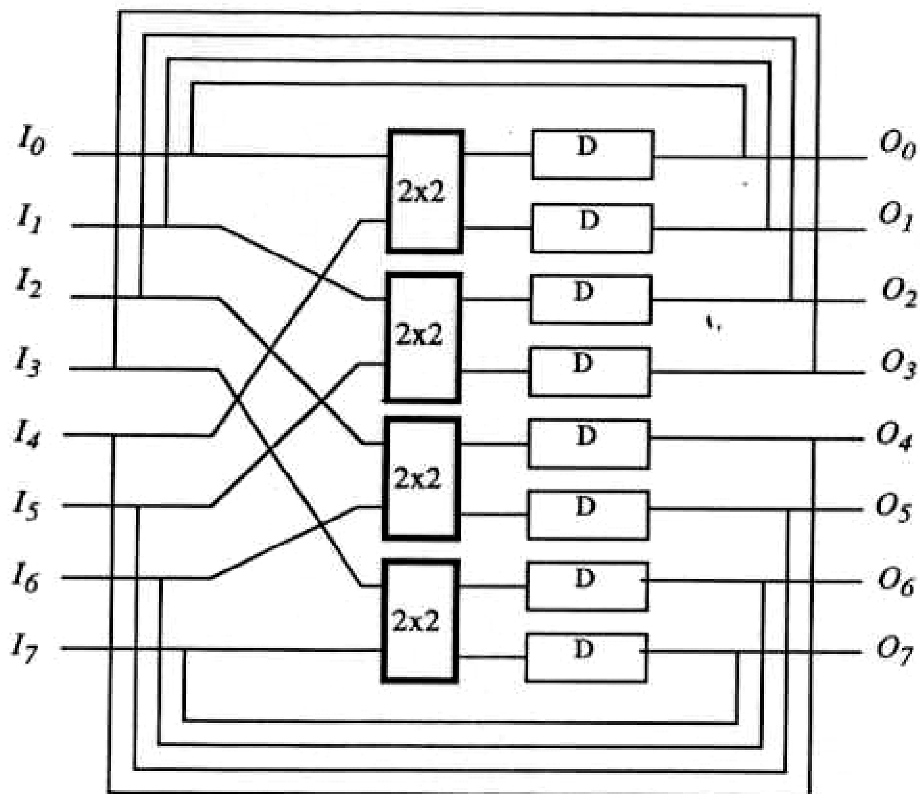


Fig. 4 A single stage shuffle-exchange network

b) Strictly Non-Blocking vs. Internal Blocking

Strictly non-blocking MINs can realize N! Permutations without any rearrangement of the existing connections; therefore, any input can

reach any output without collisions. On the other hand, MINs with only a few stages suffer the problem of internal blocking. Fig. 5 illustrates this problem where we have two connections; the first connection is from I_0 to 0_6, and the second connection is from I_6 to 0_7. It is clear that the configuration of the shaded 2 x 2 switch element is neither straight nor exchange [1].

Therefore, it has been considered that it is not permissible to perform these connections simultaneously. However, MINs are inexpensive and faster. For example, by using 2x 2 switch elements, only $\log_2 N$ stages are required to achieve full access capability in an N x N switch.

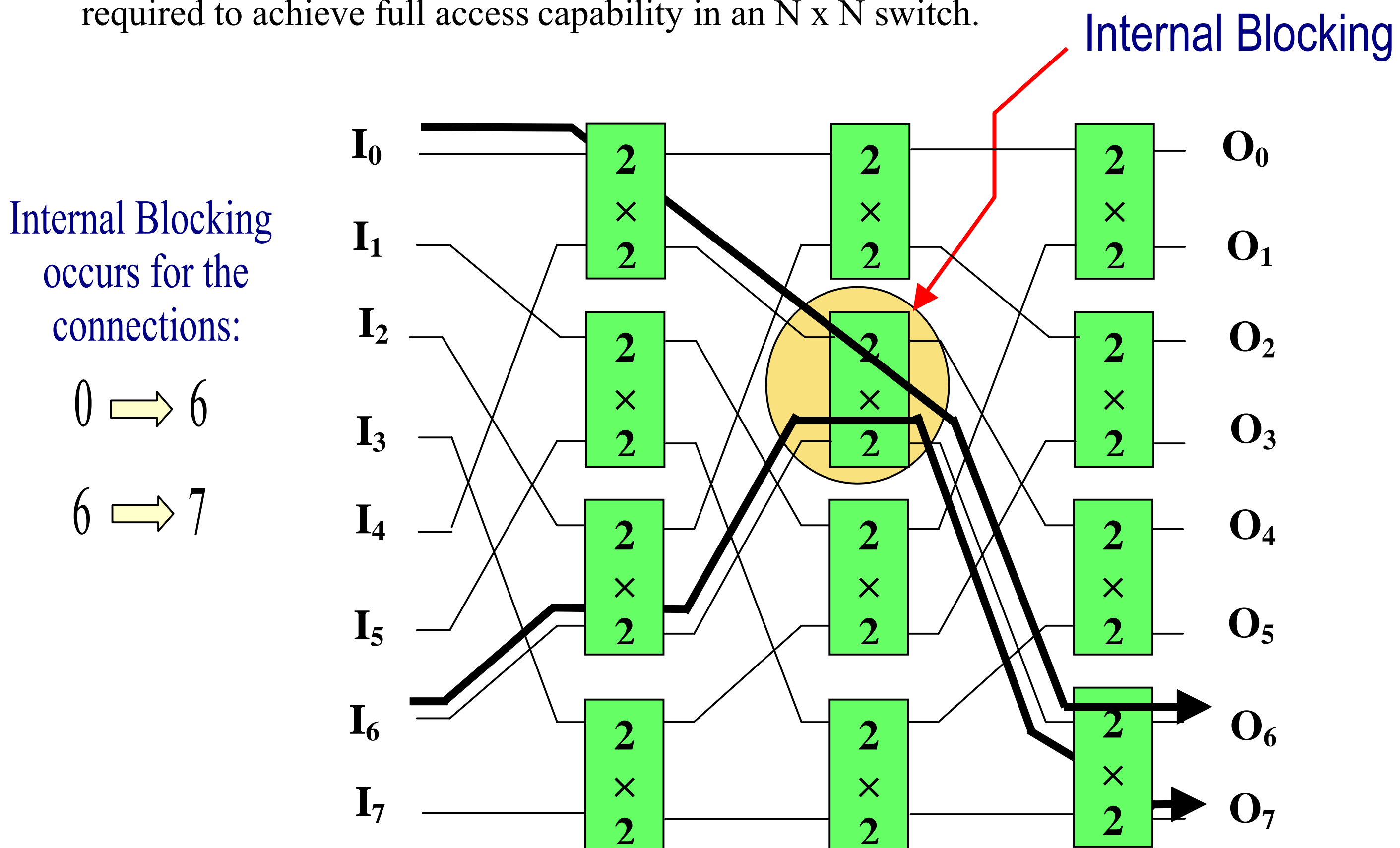


Fig. 5 Internal Blocking in Multistage Interconnection Networks.

c) Combinatorial Power

It is the ratio of the number of permutations realizable by the network to the total number of possible permutations (N!). MINs with internal blocking use only $\log_2 N$ stages, and in each stage, there are only N/2 of 2 x 2 switch elements. Since a switch element has two configurations in an ordinary multistage network, namely, straight and exchange; therefore, their combinatorial power is $\frac{2^{(N.\log N)/2}}{N!}$. For example, an ordinary 8 x 8 MIN has a combinatorial power of only 0.1016 which means that approximately 10% of permutations are realizable by these networks. With WDM, we can get a higher throughput by allowing different wavelengths using the same channel without any collision. However, if every link uses all available wavelengths, this improvement cannot be made.

These multistage interconnection networks are simple, inexpensive, and easy to build in a modular fashion. However, they have a problem of internal blocking. Fortunately, by using the WDM concept, this problem can be alleviated. In fact, it will occur only if two packets of the same wavelength are sent to the same output link of the same switch element. Therefore, if new algorithms are used to incorporate the advantages of WDM in MINs, their performance can be considerably improved.

1.2 Prior Research

The multistage interconnection networks have been studied intensively in the electrical domain. But due to the speed limitation of the electronic switching, many researches have been done over these networks in the optical domain. These researches investigate the implementation of MINs based on WDM by using either Guided-Wave Fabrics, which guide the propagation of the waves along a physically constructed path, or Free-Space Fabrics which utilize the spatial bandwidth without any predefined path by using mirrors, masks, polarized beam splitter (PBS), prism gratings, lenses, etc. The main objective of these researches is to implement such networks with very high bandwidth.

Solving the problem of internal blocking in MIN's has been considered in many previous researches. Dr. Arafah had developed new algorithms to resolve the internal blocking in MIN in Optical domain.

In this research, our goal is to implement a control unit for multistage interconnection network with *fault-tolerant* structure, by implementing *Extra Stage Omega Network* which can alleviate the problem of internal blocking by using buffering and wavelength conversion, and to improve the Omega network performance.

We redefine internal blocking as two or more packets with the same wavelength trying to access a channel simultaneously. This problem

can be eliminated by increasing the number of switch elements [5], the number of stages [5], or the size of the switch element [14]. However, all these techniques increase the cost and delay of such networks. Therefore, in this research, we attempt to use as few switch elements as possible, while maintaining the full accessibility by using *fault-tolerant* technique.

To alleviate the problem of internal blocking in MINs based on WDM, we can use buffers and/or wavelength converters. The advantage of wavelength conversion over buffering is the ability to utilize the available channel bandwidth and to send a packet to its destination without waiting for the next switching cycle.

1.3 Research contribution

In this research, we are going to apply Dr. Arafah's algorithms on Extra Stage Omega network [1], which define the behavior of the central controller, which acts as an interface in front of the MIN to resolve any internal blocking. Once the central controller resolves the internal blocking by buffering, wavelength conversion, or dropping of the packets, it directs the packets through the network without any collision.

Also, we consider additional configuration for a 2 x 2 switch element to take advantage of WDM. The additional configurations are: Upper Merger, Lower Merger, Upper Splitter, and Lower Splitter. Therefore, all permutations are permissible by MINs.

However, we develop the Extra Stage Omega Networks as shown in Fig. 6 and then we analyze the performance of them with modification to a 2x2 switch element to incorporate the new configurations, namely the splitters and the mergers.

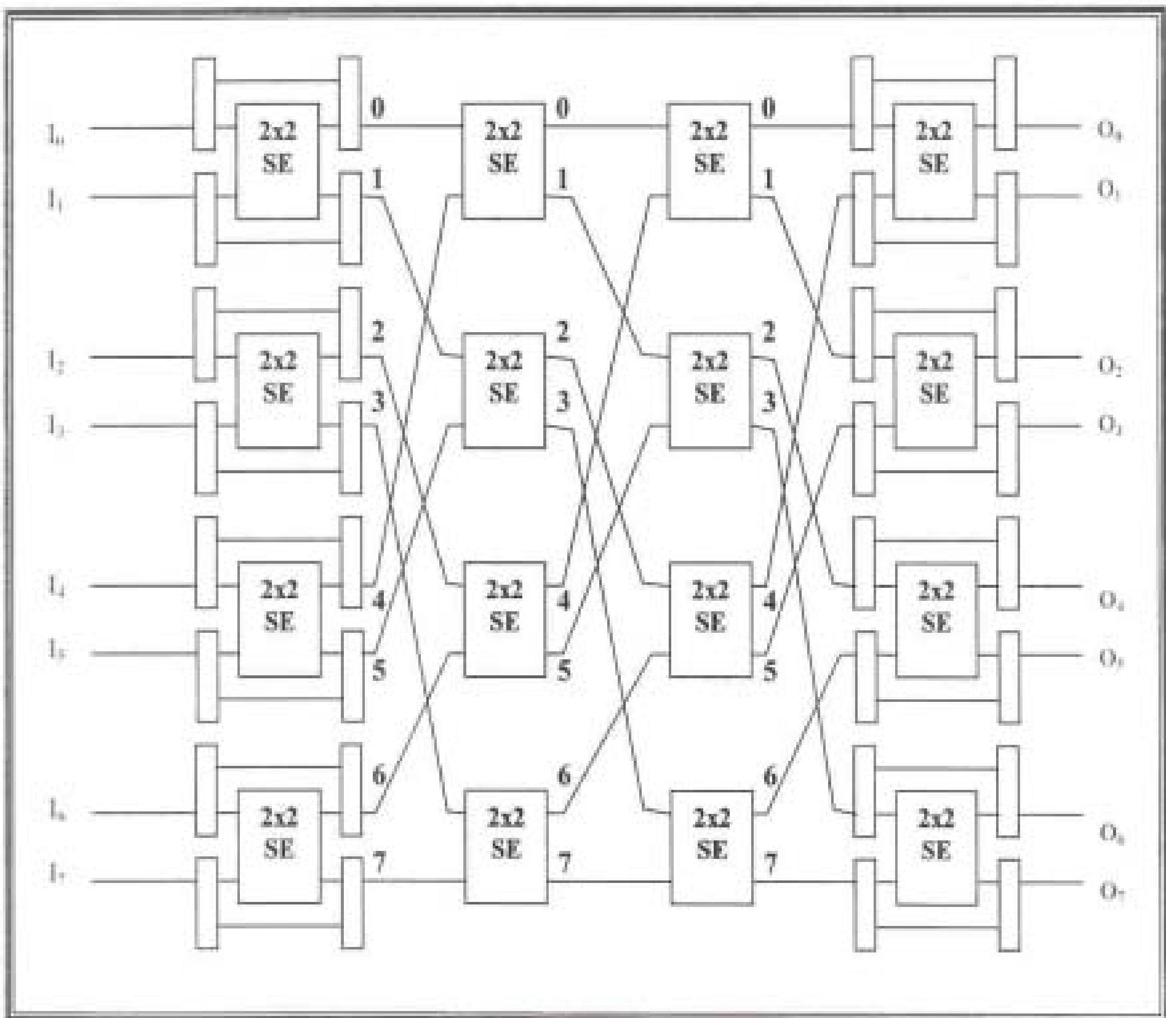


Fig. 6 Extra Stage Omega Network

We found that structure of the central unit is very simple to implement with minimal buffers available only at the central controller rather than distributed over all 2 x 2 switch elements. This will increase the utilization of the buffers. In case of distributed buffers, a switch may need more buffers than it has, while other switches may have idle buffers. Therefore, we consider centralized buffers as a basis of our research. In addition we add wavelength converters in the central controller. Rather than buffering a packet in the current switching cycle, we convert its wavelength to another wavelength to enable the ESO network to accommodate it. This has increased the rate of transmission and increased the utilization of the ESO network.

1.4 Research Motivation

Using the Extra Stage Omega Networks based on WDM has many advantages such as:

1- Redundancy: They have an additional path from each source to each destination.

2- Higher switching speed: We need to have switches, which can work beyond the capability of electronic switches.

3- Transparency: The current technology for transmission is fiber optics, therefore, by using electronic switches, we need to have Optical-Electrical Interface (O/E) and demultiplexing from high speed of a fiber channel to the speed of electronic switches. Therefore, we need to remove this bottleneck.

4- Performance upgrading: We need to develop new class of Omega networks based on WDM with a central controller to detect and resolve any internal blocking using buffering or wavelength conversion.

5- Efficient management of wavelengths: Due to the limited number of wavelengths, we have to handle these precious resources efficiently. We have to avoid any unnecessary wavelength conversion.

6- Cost Reduction: We have considered only multistage interconnection networks, which are cheaper than Crossbar Networks.

Chapter 2

Omega Networks with One-to-One Connections

2.1 Introduction

In this chapter, we introduce the architecture of Omega Network using WDM and two algorithms to deal with the problem of internal blocking in these networks. Also, we introduce four additional configurations for a 2x2 switch element to make any permutation permissible by a MIN.

2.2 Architecture of Omega Network

Based on the previously discussed characteristics of MINs, we have found that an NxN Omega Network, first developed by Lawrie (1975)[2], is best suited for WDM implementation. It consists of $n = \log_2 N$ identical stages, and each stage consists of a perfect shuffle connection followed by N/2 2 x 2 switch elements as shown in Fig. 7.

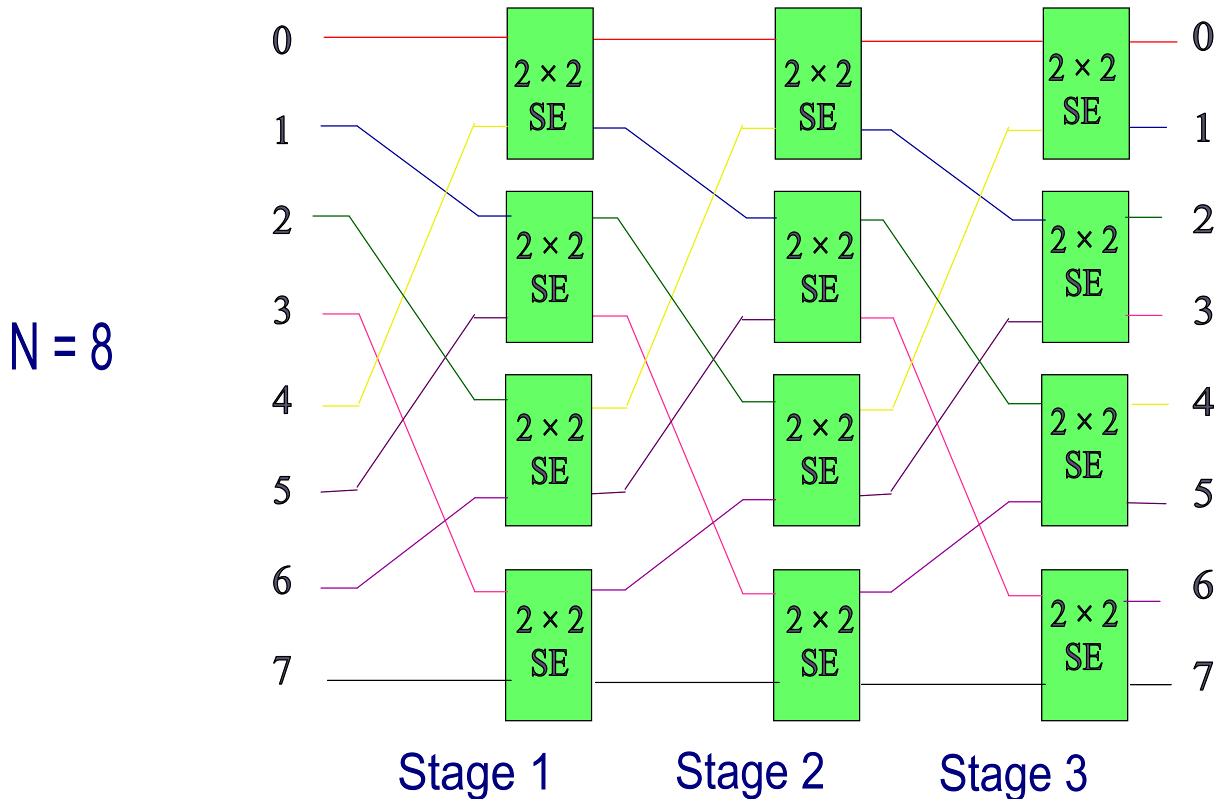


Fig. 7 8x8 Omega Network consisting of 3 stages, with each stage consisting of a Perfect-Shuffle connection followed by 4 2x2 switch elements.

The perfect shuffle connection of the Omega Network simply divides the N channels into two halves, which are then interleaved perfectly. It can be formally represented as follows: if the binary representation of an input link is $x_{n-1}\ x_{n-2}\ldots x_1 x_0$, and if it is perfectly shuffled, then the message will be directed to a destination with a binary representation $x_{n-2}\ldots x_1\ x_0\ x_{n-1}$.

The perfect shuffle connection is followed by a stage of N/2 2x2 switch elements. The general configuration of switch elements is shown in Fig. 8a. Each element can be configured independently based on the given permutation. Since we consider in this chapter only the one-to-one mapping between a set of input links and a set of output links, each switch can be configured into two states: either straight or interchange as

shown in Fig. 8b. Since this network is based on WDM, and each input link an carry at most ω packets, each with a different wavelength from the set of available wavelengths, $\Lambda = (\lambda_1, \lambda_2 \ldots, \lambda_{\omega-1}, \lambda_\omega)$, one may merge both inputs of a 2 x 2 switch element and forward them to either the upper or lower output link when the sets of wavelengths on both input links are disjoint. Therefore, two configurations have to be considered, namely, upper merger and lower merger as illustrated in Fig. 8c. Note that these two configurations would have been considered internal blocking in ordinary Omega Networks. Moreover, we need two additional configurations, namely, the upper splitter and lower splitter as illustrated in Fig. 8d to satisfy the requirements of one-to-one mapping [1].

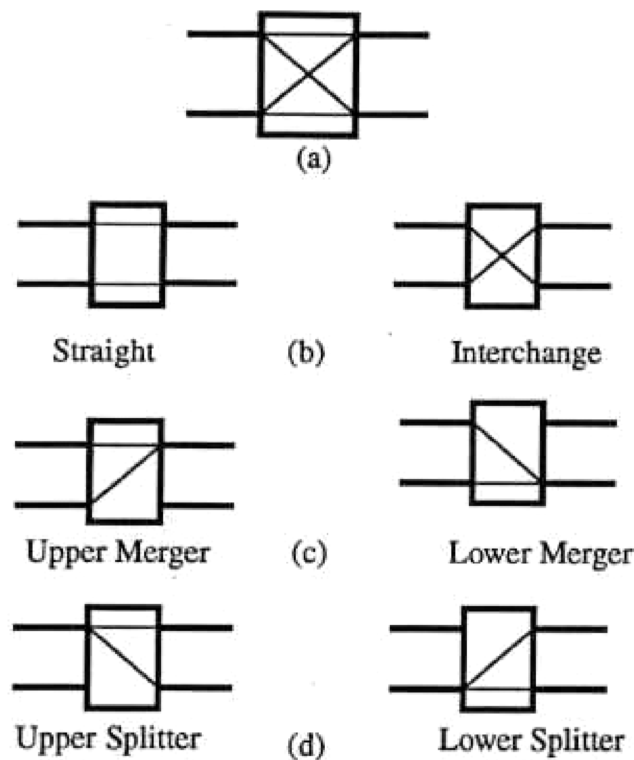


Fig. 8 The general switch is illustrated in (a), and the different configurations are illustrated in (b), (c) and (d).

This architecture has been selected as our target multistage network because of the following advantages:

- It has full access capability, since each input link can reach any output link in a single pass.
- It has a simple structure and a small number of stages, and therefore, is fast and inexpensive.
- It can realize N! permutations by considering the configurations of the 2 x 2 switch elements and the concept of WDM.
- It has the capability of self-routing since the destination address represents the routing tag. If the binary representation of an input link is: s_{n-1} s_{n-2}…$s_1 s_0$, and the binary representation of an output link is: d_{n-1} d_{n-1}…d_1 d_0, then the bit d_{n-1} sets the 2x2 switch element at the first stage, the bit d_{n-2} sets the switch at the second stage, and so on. If a bit is equal to zero, it sets the connection to the upper output link of the 2 x 2 switch element, and if a bit is equal to one, it sets the connection to the lower output link of that switch [2][3].
- Set of permutations realizable by an Omega Network is called an Omega class, which is characterized by having n-1 windows, where $n=\log_2 N$, and each of them is a permutation. Fig. 9 illustrates the definition of Windows (W_i).

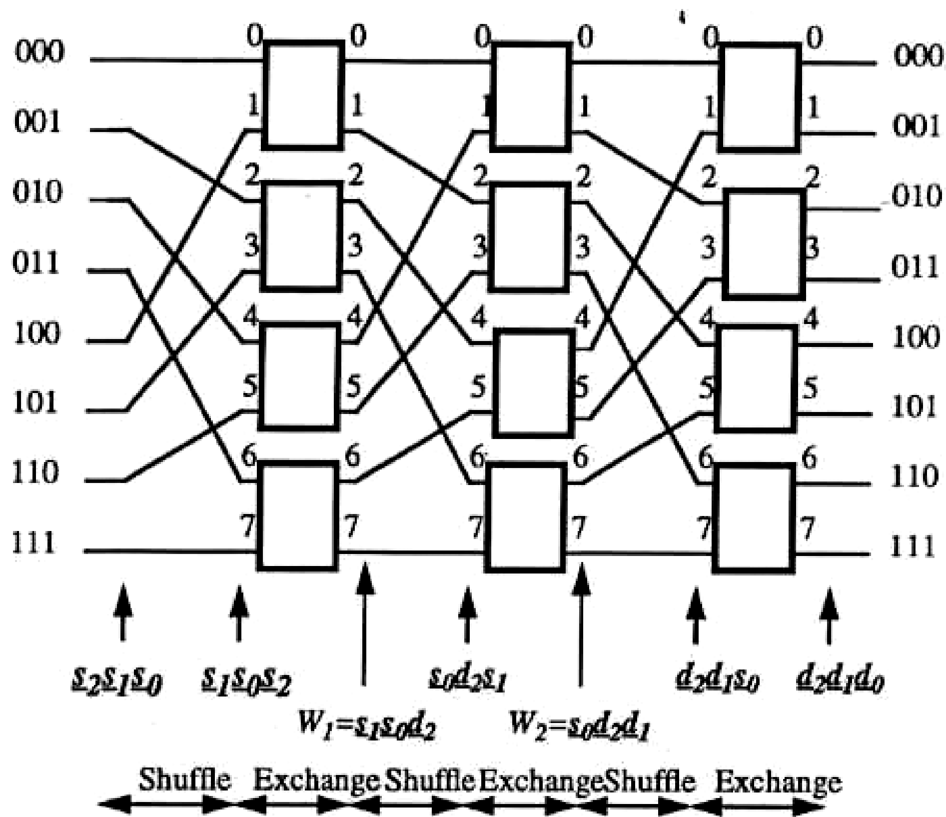


Fig. 9 Windows on an Omega Network

The main drawback of this architecture is that there is only one path between any source and any destination. Therefore, failure of any path causes a loss of full access capability. However, it still maintains the dynamic full access capability.

To increase the reliability of Omega Networks, we add some redundancy by using ESO networks and rerouting in ESO network under fault as shown in Fig. 10.

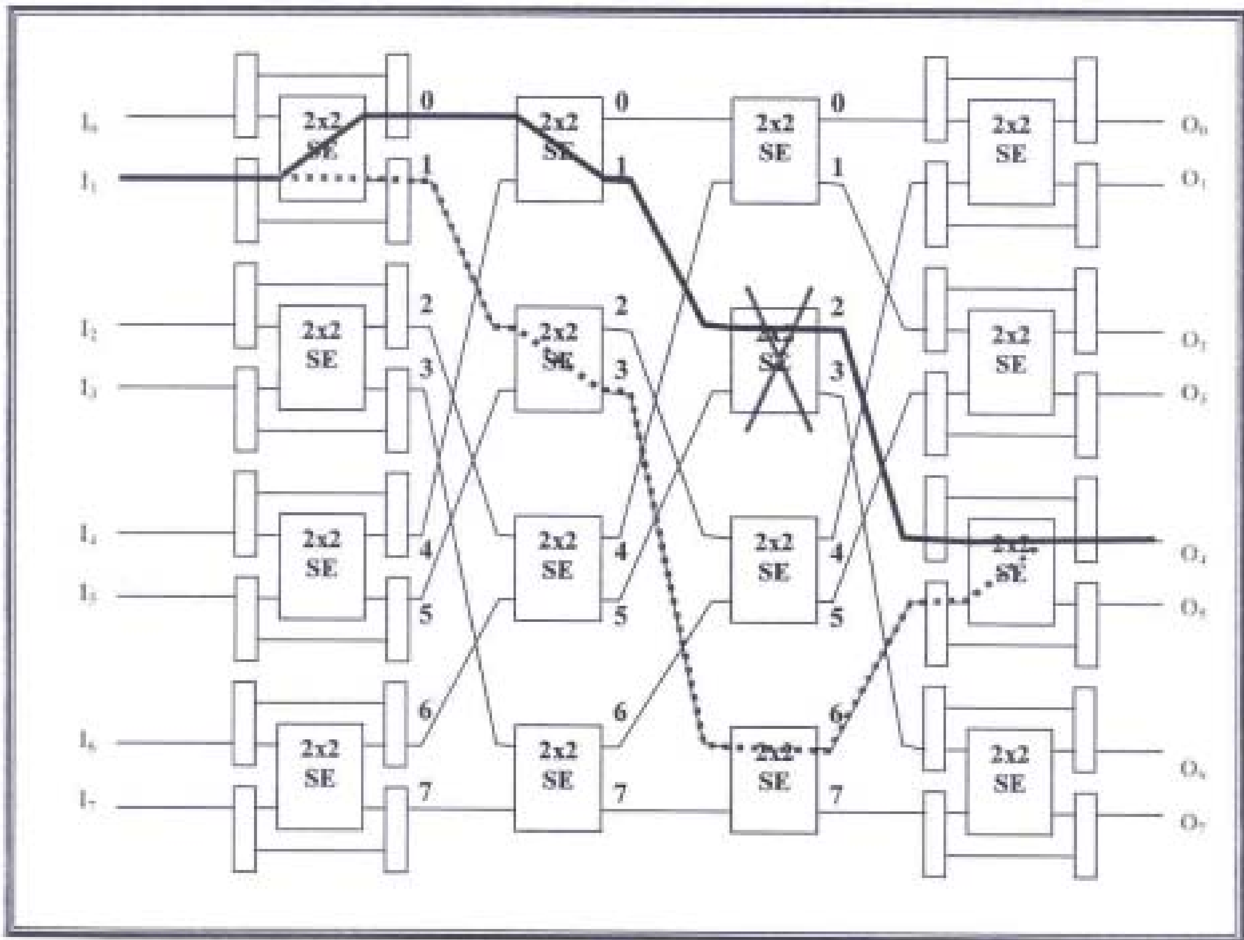


Fig. 10 Rerouting in Extra Stage Omega Network under fault.

2.3 Algorithms to Resolve Internal Blocking in Omega Networks

Considering the Omega Networks based on WDM which have been discussed in the previous section, internal blocking occurs only when two sets of packets, from two different input ports, are sent to the same output port of a 2 x 2 switch element, and at least one packet in the first set has the same wavelength as another packet in the second set. We will apply two algorithms to resolve internal blocking in Omega networks Buffering and Wavelength Conversion.

2.3.1 Omega Networks with Buffering

In this section, we use the same architecture of an Omega network, which has been discussed, in section 2.2. However, we include Buffers in the central controller of that network as illustrated in Fig. 11.

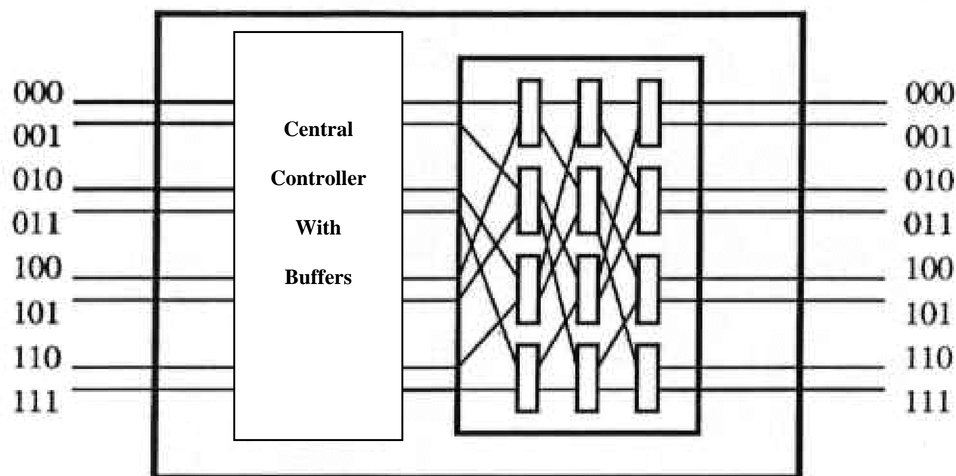
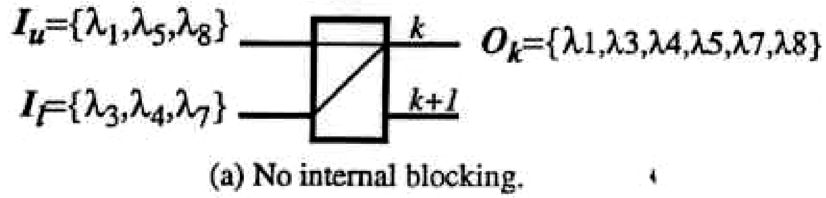


Fig. 11 The architecture of a collision-free Omega Network with a central controller with buffers.

For example, if the arriving set of packets to the first input port of a 2 x 2 switch element uses wavelengths (λ_1,λ_5,λ_8), and the arriving set of packets to the second input port uses (λ_3,λ_4,λ_7), and if both sets of packets are sent simultaneously to the same output port of that switch element, then there will be no internal blocking since these two sets are disjoint. This example is illustrated in Fig. 12a. In the ordinary Omega Network, this will cause internal blocking, and the permutation is not permissible. However, by using the concept of WDM, this permutation is allowed [1].



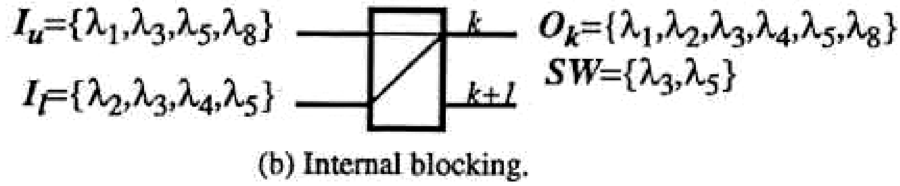


Fig. 12 Illustration of internal blocking in DWM-based Omega Network. SW is the set of wavelengths that causes the internal collision.

In Fig. 12b, packets using wavelengths (λ_1,λ_3,λ_5,λ_8) and (λ_2,λ_3, λ_4,λ_5), are sent simultaneously to the same output port of a switch element. There will be internal blocking due to two packets, which are using wavelengths, λ_3 and λ_5, respectively. The network controller can detect any internal blocking by the given permutation. Packet collisions are avoided by buffering the packets with wavelengths λ_3 and λ_5 at the controller. For simplicity, from here on, we assume that only packets using the lower input links can be buffered. Then, the set of packets forwarded to the output port use wavelengths (λ_1,λ_2,λ_3,λ_4,λ_5,λ_8). By using these concepts, we can develop an algorithm to resolve the problem of internal blocking and hence enhance the performance of Omega

Networks [1]. This algorithm is based on the definition of W_i and illustrated in Fig. 9. It is shown by an example in Fig. 13

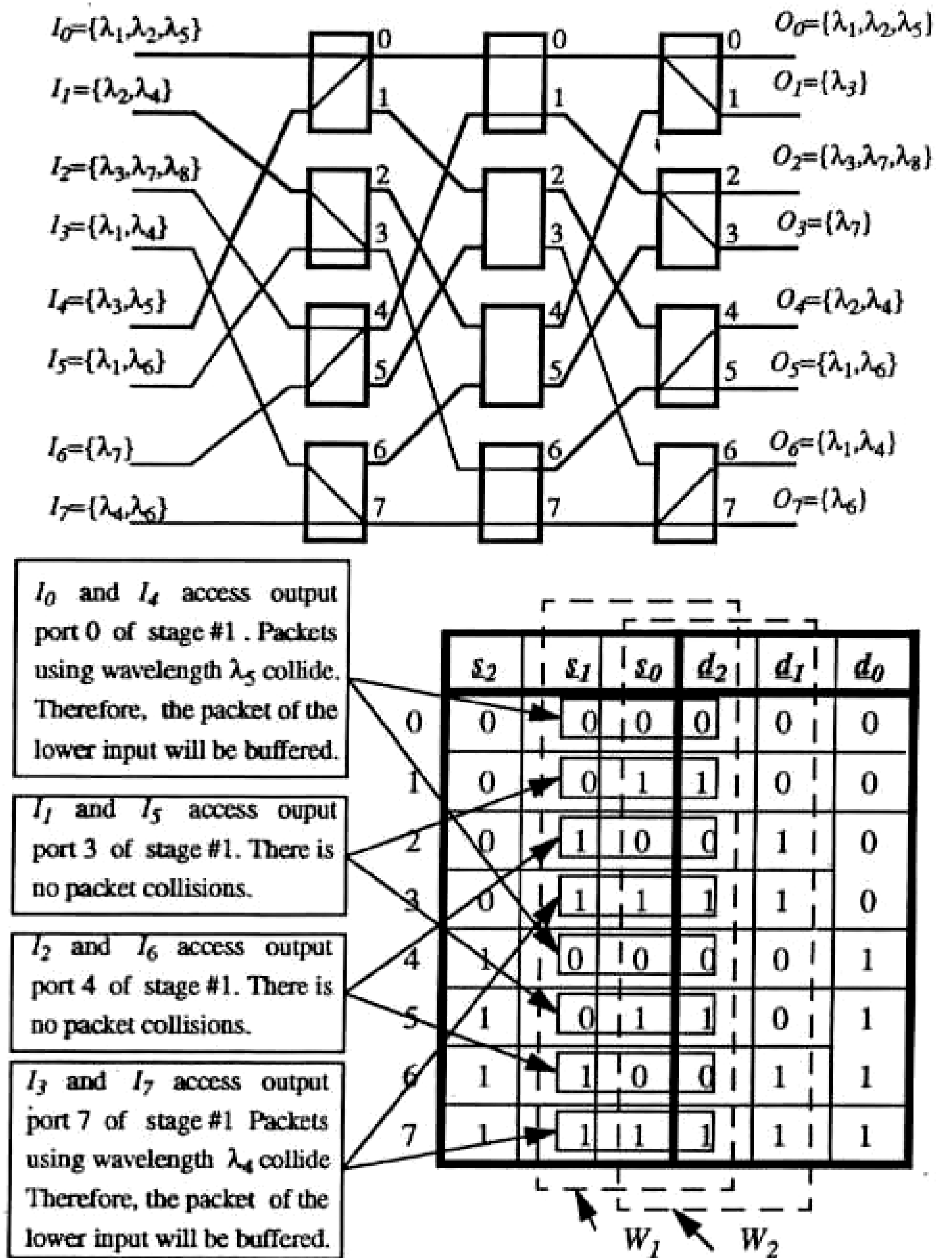


Fig. 13 Illustration of resolution of internal blocking in Omega Networks by buffering.

Although this algorithm will improve the performance of Omega Networks based on WDM, the improvement achieved is not as great as

that achieved by introducing wavelength converters in the network architecture.

2.3.2 Omega Networks with Wavelength Conversion

In this section, we use the same architecture of an Omega Network which has been discussed in section 2.2. However, we include Wavelength Converters in the central controller of that network as illustrated in Fig. 14. The purpose of a wavelength converter is to convert the wavelength λ_i of a packet to another wavelength λ_j, where $\lambda_j \neq \lambda_i$. Therefore, if we have internal blocking, and we manage to convert the wavelengths of packets that cause the internal blocking, we can reduce number of packets to be buffered by the central controller, and the performance of that network will be improved. Note that a packet can have at most one wavelength conversion and this happens inside the central controller. The number of packets which can have their wavelengths converted is limited by the number of available converters. Packets which will cause internal collision but which cannot be wavelength converted are buffered. If we run out of buffers, packets are dropped [1].

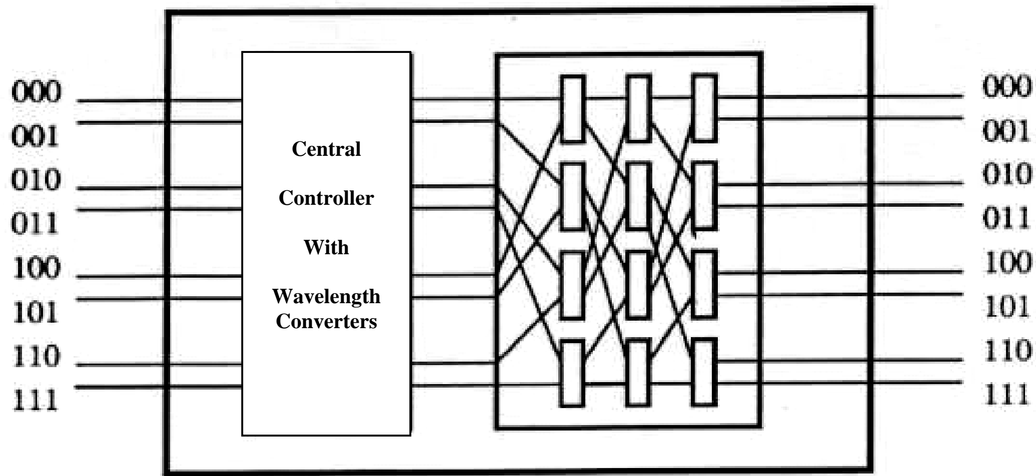
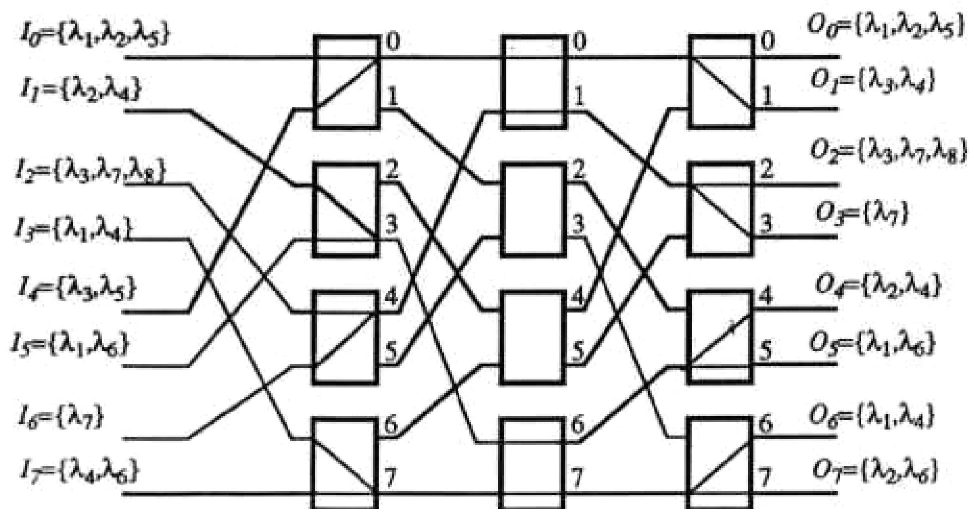


Fig. 14 The architecture of a collision-free Omega Network with a central controller with wavelength converters.

Considering the same example of the previous section, if the arriving set of packets to the first input port of a 2 x 2 switch element uses wavelengths (λ_1, λ_3, λ_5, λ_8), and the arriving set of packets to the second input port of that switch uses wavelengths (λ_2, λ_3, λ_4, λ_5), and if they are sent simultaneously to the same output port, then there will be internal blocking by two packets which use wavelengths λ_3 and λ_5, respectively. However, if the wavelengths λ_3 and λ_5 of the lower input link are converted to λ_6 and λ_7, respectively, then there will be no internal blocking at that switch element, and its output port can forward the set of packets with wavelengths (λ_1, λ_2, λ_3, λ_4, λ_5, λ_6 λ_7, λ_8) to the next stage. Fig. 15 illustrates the resolution of internal blocking in an 8 x8 omega network by wavelength conversion [1].



W_1 is not a permutation. Therefore, stage #1 may have internal blocking.

Packet using wavelength λ_5 of input port 4 is converted to wavelength λ_6.

No wavelength conversion.

No wavelength conversion.

Packet using wavelength λ_4 of input port 7 is converted to wavelength λ_2.

	s_2	s_1	s_0	d_2	d_1	d_0
0	0	0	0	0	0	0
1	0	0	1	1	0	0
2	0	1	0	0	1	0
3	0	1	1	1	1	0
4	1	0	0	0	0	1
5	1	0	1	1	0	1
6	1	1	0	0	1	1
7	1	1	1	1	1	1

W_1 W_2

Fig. 15 Illustration of resolution of internal blocking in Omega Network by Wavelength Conversion.

Chapter 3

Fault-Tolerant Structure

Introduction

In this chapter, we introduce a fault-tolerant structure based on Omega network using WDM. Then, we discuss a fault-tolerant ATM switch.

Traditionally, Fault-Tolerance has referred to building subsystems from redundant components that are placed in parallel. A prime example is the computer system for important government institution. The software will run on two pairs of primary computers, with one pair being in control as long as the simultaneous computations on both agree with each other, with control passing to the other pair in the case of mismatch. The term fault-tolerance in a software environment is used, if the software does the following:

- The program is able to compute an acceptable result even if the program itself suffers from incorrect logic.
- The program, whether correct or incorrect, is able to compute an acceptable result even if the program itself receives corrupted incoming data during execution.

A fault will be defined as the failure of:

- A component of the system.
- A subsystem of the system.
- Another system, which has interacted with the considered system.

Fault detection is usually the first step in fault tolerance. Even if other elements of a system prevent a failure by compensating for a fault, it is important to detect and remove faults to avoid the exhaustion of system fault tolerance resources.

3.1 Fault-Tolerance in MIN based

Multistage interconnection networks (MINs) are popular and efficient interconnection for large-scale multicomputers such as IBM SP1/SP2 [8] and NEC Cenju-3 [9]. Many of them are a class of networks which consist of $\log_2 N$ stages of 2 x 2 switching elements connecting N input ports to N output ports. These networks have the property of full access capability that any output can be reached from any input in a single pass through the network. In addition, there exit a unique path between any pair of input and output in these networks. The unique path property helps the use of a simple and efficient routing algorithm for setting up connections.

However, any single fault on a link or a switching element (SE) of these networks may cause to destroy the full access property. Interconnection networks have the feature of fault-tolerance if they can sustain to provide connection in spite of having faulty components.

To achieve fault-tolerance in MIN-based multicomputers, there are two alternative approaches. The first is to add SEs and/or links in the network, which provide multiple paths to detour faulty elements [10]. In this scheme, the failure of SE(s) and/or link(s) in the network causes reconfiguration of the network in order to preserve full access capability. The reconfiguration network by such scheme has the same communication capability as the original network.

The second scheme is to expense routing overhead in order to minimize the loss of resources [11]. Thus, the influence of the faulty SE(s) and/or link(s) can be decreased by allowing multiple passes through the network. The network is known to possess dynamic full access capability if every output can be reachable from every input in a finite number of passes, as routing the packet through intermediate outputs if necessary. A routing algorithm is known as recursive scheme, which allows routing through the network. Without loss of resources, in this scheme, a destination can be reachable from its source detouring faulty element(s).

In this research, we will focus on the Extra Stage Omega (ESO) interconnection network, as a type of a fault-tolerant structure, is proposed for use in large –scale parallel and distributed supercomputer systems. It has all of the interconnecting capabilities of the multistage networks that have been proposed for many supersystems. The ESO network is derived from the Omega network by the addition of one stage of interchange boxes and a bypass capability for two stages as shown in Fig. 16.

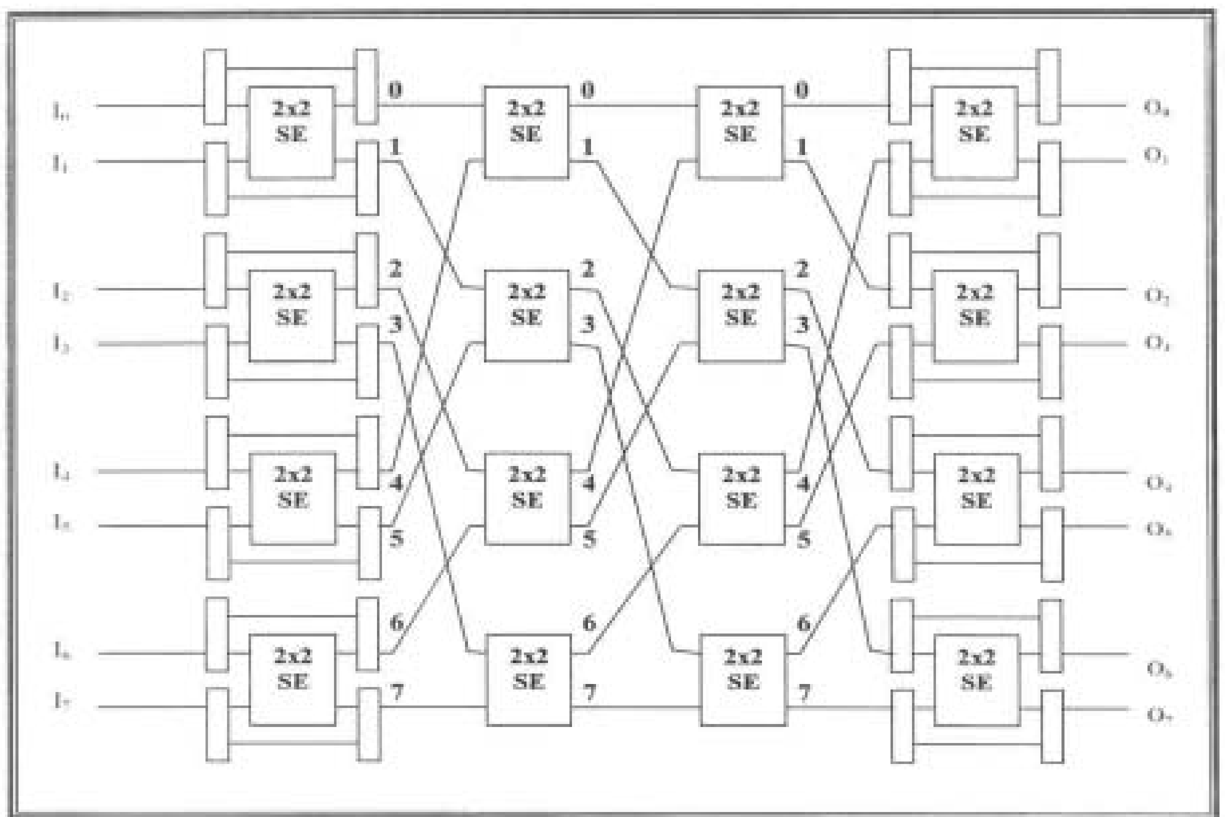


Fig. 16 Extra Stage Omega Network

It is shown that the ESO network provides fault-tolerance for any single failure as shown in Fig. 17. Further, the network can be controlled even when it has failure, using simple modification of a routing tag scheme proposed for Omega network. The ESO network consists of Omega network with one additional stage at the input and hardware to allow the bypass, when desired, of the extra stage or the output stage. Thus, the ESO network has a relatively low incremental cost over the Omega network. The extra stage provides an additional path from each source to each destination. Therefore, the ESO network is a practical answer to the need for reliable communications in parallel/distribution supersystems.

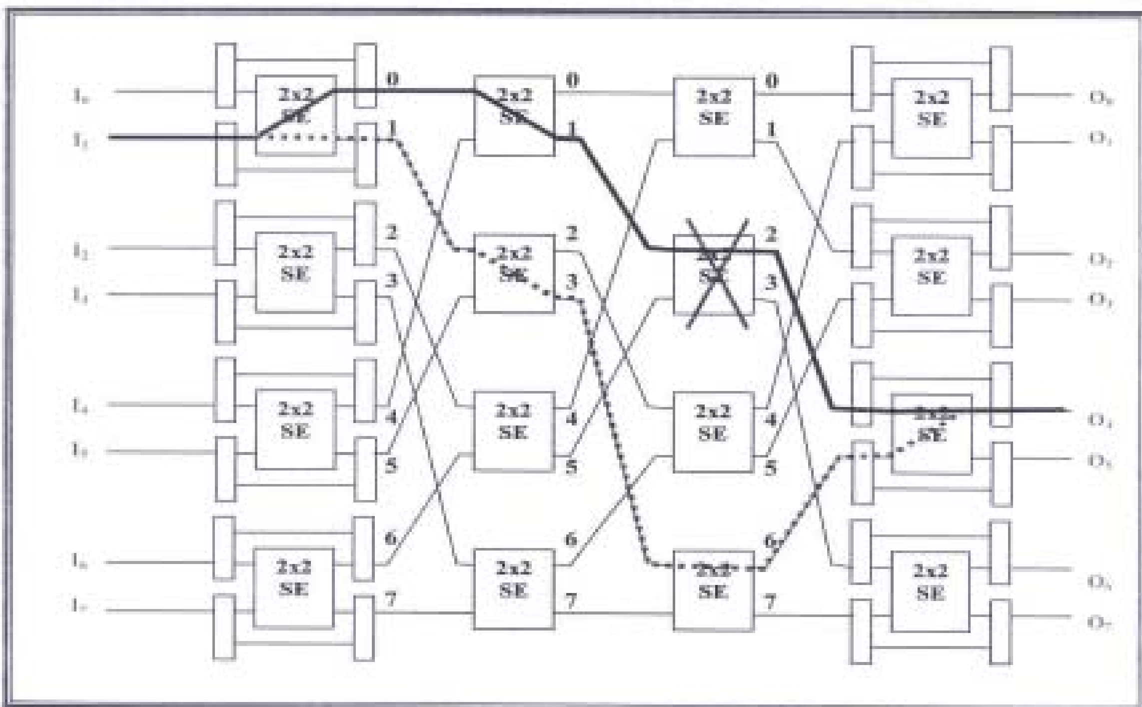


Fig. 17 ESO network with fault-tolerance for any single failure

3.2 Fault-Tolerant ATM Switch

Asynchronous Transfer Mode (ATM) was developed for use in Broadband Integrated Service Digital Networks (B-ISDN). B-ISDN is designed to meet the needs of future applications (data, voice, image and video, etc.) There are economies of scale in having a single network to provide all telecommunications and data services. It goes without saying that B-ISDN will play a significant role in future networks. Since ATM is the core data transmission technique for B-ISDN, we need high speed switches for fast switching. Under ATM transmission, no matter what kind of data is transmitted, the transmitted data is copped into basic data transmission units of 53 bytes. The basic data unit is called cell. The 5 bytes at the beginning of the cell is the cell header. The remaining 48 bytes are reserved for user data.

For switching cell streams, several ATM switch architectures have been proposed. MIN architecture is considered a better way to transmit ATM cells in terms of the balance between hardware cost and switching speed.

There are several MIN architectures suitable for ATM switches, such as Banyan, Baseline and Omega [13]. The advantage of MINs for ATM switches is their parallel processing capability. For example, an N x N MIN can process N independent cells simultaneously. Under parallel processing, it is possible that two independent cells will contend for a link

or a switching element (SE) output port within MIN. This situation is called blocking or contention. Such MIN switches, called blocking switches, include, Banyan Baseline, and Omega. The method for resolving contention is to add queues to buffer the low priority cells in contention or to provide multiple paths between each input/output pair such as in the Benes switch. In this switch, the number of multiple paths and the number of stages are in proportion to the size of the switch. However, its switch size and stage number are nearly double those of the Banyan switch. Furthermore, it also needs a complex routing algorithm between each input/output pair. Contention can also be solved by rearranging the paths that have been set up already. There are some switches that are non-blocking. Batcher-Banyan is one of them. It used a switch called Batcher to sort incoming cells in a certain order in front of the Banyan switch in order to avoid internal blocking. Hence, the Batcher-Banyan switch can avoid internal blocking without using any buffers [12]. Normally, the Batcher switch is more complicated and larger than the Banyan switch. This means that we need to pay a high hardware cost to resolve the internal blocking problem.

Several ATM switch architectures were presented to satisfy the high speed switching requirements. We can classify them into two: with fault tolerance or without. A switch architecture without fault tolerance concentrates on making the switching rate as fast as possible. Owing to

no fault tolerance, if a fault occurs in a switch, the switch will be partially or totally disrupted. A few papers have dealt with fault tolerance ATM switch designs. These approaches are based on a 2x2 modified SE but add extra SEs or links, or use multiple pass to provide fault tolerance. A modified Delta network, which provides fault tolerance, was presented in [13]. One extra stage and double links were added to the original Delta network. Thus, a 2x2 SE was replaced by a 4x4 SE in the middle stage of the resulting network and 4x2 SE in the last stage, respectively. Some subswitches were used to enhance the fault tolerance of the conventional multistage interconnection network by providing alternative paths between each input/output pair. The larger the switch size is, the greater the number of redundant paths it has.

The SE will present here is a 2x2 FTSE (Fault Tolerant Switching Element) which can be the basic building block of an MIN switch for high broadband networks. Since the fault tolerance capability is quite important in a high speed network, the proposed FTSE can satisfy this need. The fault tolerance scheme in the FTSE employs one spare input controller (IC) and two spare output controllers (OCs) to provide multiple paths between each IC/OC pair. This enables the FTSE to tolerate faults.

Chapter 4

Extra Stage Omega Network

4.1 Introduction

The demand for very high speed processing coupled with falling hardware costs has made large-scale parallel and distributed supercomputer systems both desirable and feasible. An important component of such supersystems is a mechanism for information transfer among the computation nodes and memories. Because of system complexity, assuring high reliability is a significant task. Thus, a crucial practical aspect of an interconnection network used to meet system communication needs is fault tolerance.

Multistage networks such as the Baseline, Delta, and Omega have been proposed for use in parallel/distributed systems. The problem with these networks is that there is only one path from a given network input to a given output. Thus, if there is a fault on that path, no communication is possible.

4.2 The Extra Stage Omega Network

The Extra Stage Omega (ESO) interconnection network, a Fault-tolerant structure, is proposed for use in large-scale parallel and distributed supercomputer systems [4]. It has all of the interconnecting capabilities of the multistage networks that have been proposed for many supersystems. The ESO network is derived from the Omega network by the addition of one stage of interchange boxes and a bypass capability for two stages. It is shown that the ESO network provides fault tolerance for any single failure. Further, the network can be controlled even when it has a failure, using a simple modification of a routing tag scheme proposed for the Omega network. The ESO network consists of an Omega network with one additional stage at the input and hardware to allow the bypass, when desired, of the extra stage or output stage. Thus, the ESO network has a relatively incremental cost over the Omega network (and its equivalent networks). The extra stage provides an additional path from each source to each destination. The known useful attributes of partitionability and distributed control through the use of routing tag are available in the ESO network. Therefore, the ESO network is a practical answer to the need for reliable communications in parallel/distributed supersystems. The ESO network can be used to

provide fault-tolerance in addition to the usual Omega network communication capability and can be used in various ways in different computer systems.

The ESO network is formed from the Omega network by adding an extra stage along with a number of multiplexers and demultiplexers. Its structure is illustrated in Fig. 18 for N=8. The extra stage, stage 0, is placed on the input side of the network.

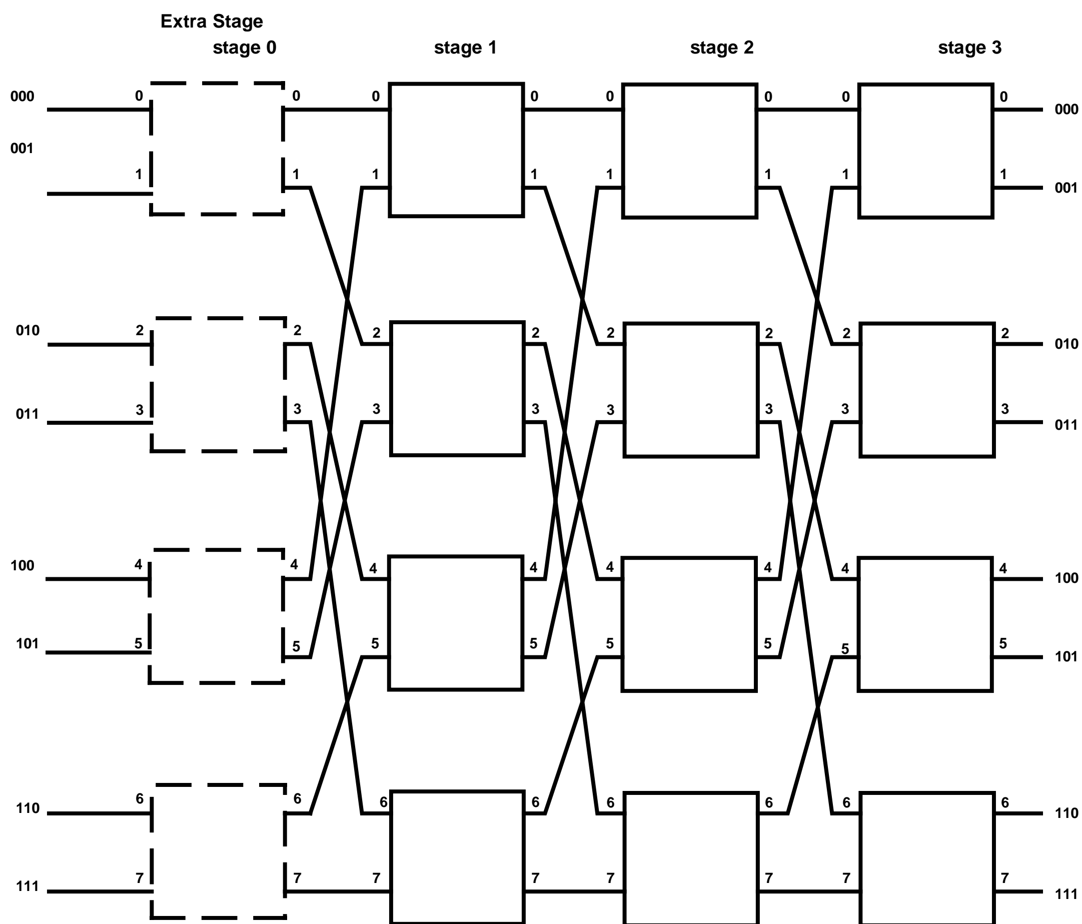
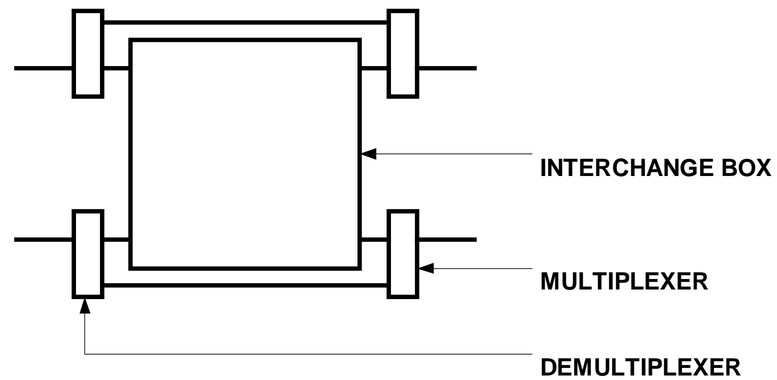


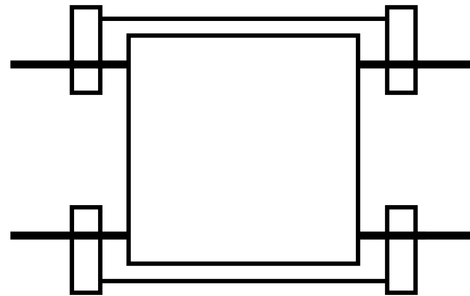
Fig. 18 Extra Stage Omega (ESO) network with N=8.

Stage 0 and Stage n can each be enabled or disabled (bypassed). A stage is enabled when its interchange boxes are being used to provide

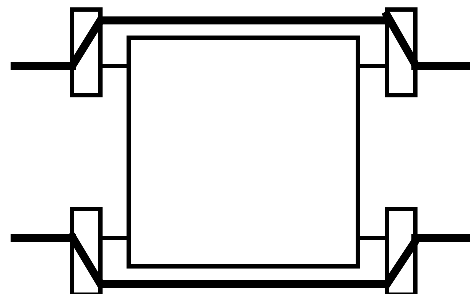
interconnection. It is disabled when its interchange boxes are being bypassed. Enabling and disabling in stages 0 and n is accomplished with a demultiplexer at each input and a multiplexer at each output. Fig. 19 details an interchange box from stage 0 or n.



(a)



(b)



(c)

Fig. 19 (a) Deatail of interchange box with multiplexer and demultiplexer of enabling and disabling. (b) interchange box enabled. (c) Interchange box disabled.

Stage enabling and disabling is performed by a system control unit. Normally, the network will be set so that stage 0 is disabled and stage n is enabled. The resulting structure is that of Omega network. If after running fault detection and location tests a fault is found, the network is reconfigured. If the fault is in stage n then stage 0 is enabled and stage n is disabled. For a fault in a link or box in stage 1 to n-1, both stages 0 and n will be enabled. A fault in stage 0 requires no change in network configuration, stage 0 remains disabled. If a fault occurs in stages 1 through n-1, in addition to reconfiguring the network the system informs each source device of the fault by sending it a fault identifier.

In the fault model to be used, failures may occur in network interchange boxes and links. However, the input and output ports and the multiplexers and demultiplexers directly connected to the ports of the ESO are always assumed to be functional. If a port or the stage 0 demultiplexers or stage n multiplexers were to be faulty, then the associated device would have no access to the network. Such a circumstance will not be considered.

In this research, our goal is to implement a central controller unit for multistage interconnection network with fault tolerant structure, by implementing ESO network which can alleviate the problem of internal blocking by using buffering or/and wavelength conversion as shown on Fig. 20.

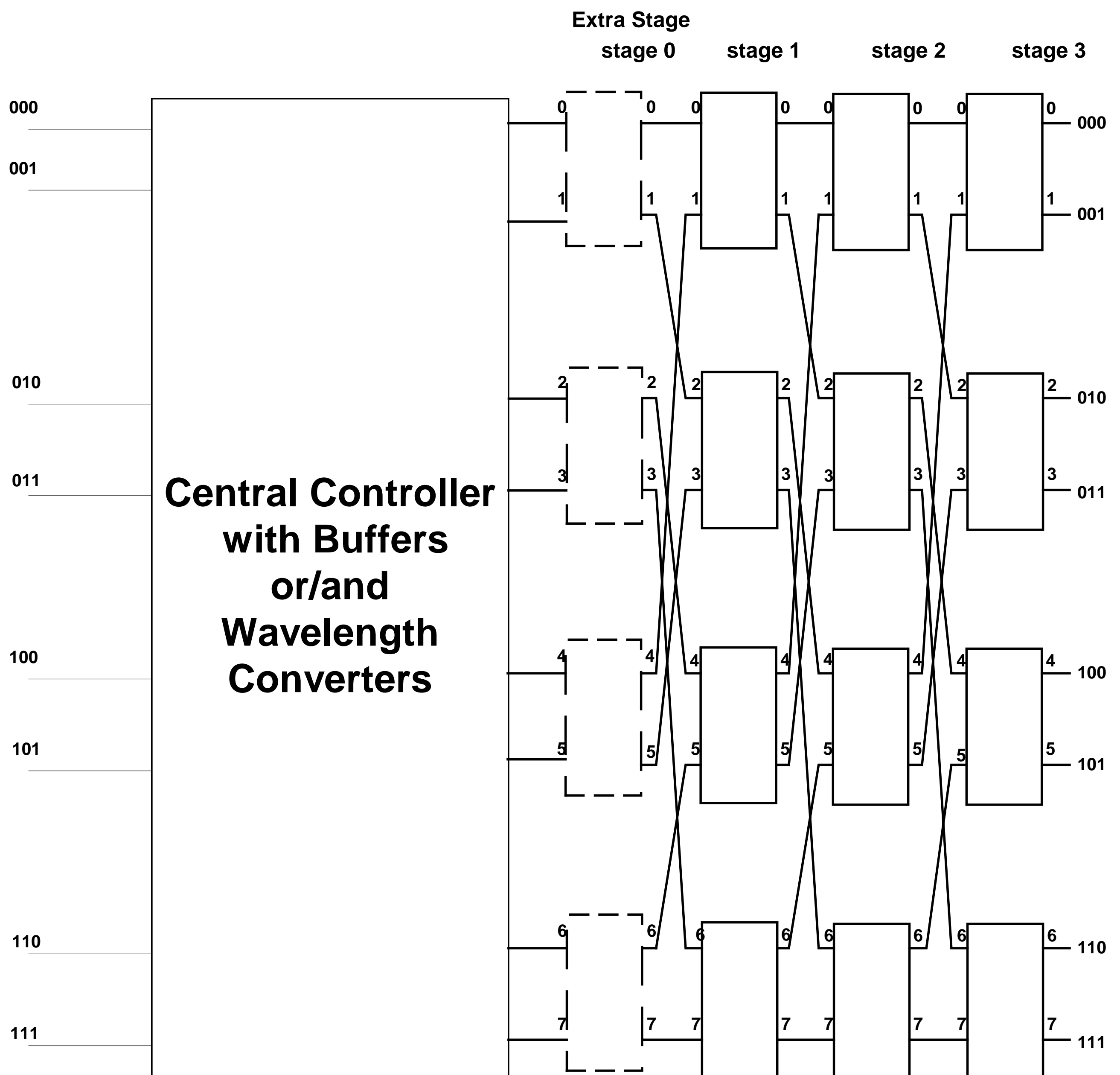


Fig. 20 Extra Stage Omega Network with Central Controller with buffers or/and wavelength converters.

4.3 The algorithm

1- Initialize summary values

2- Define the Extra Stage Omega connections **Perfect Shuffle** (σ) between the switches

3- Generate a random output permutations

4- Generate a random input packets per bundles (input links)

5- Compute bundles routing for each stage

   - We start with bundle 0 on channel 0 of the first switch on the first stage
   - The function will go through all stages recursively and then continue the path with the next bundle until the last bundle

Note: We try to reduce the merging of bundles

6- Compute packets passing

   - If there is more than one input bundle on a channel, a merge must be done
   - Initialize merge bundle
   - While merging, packets might be converted, buffered or dropped
   - We will start with the conversion.
   - If there is no availability for a conversion, we are going to a buffering

- If there is no availability for both, packet will be dropped from the network

7- Compute statistic values

- Converted packets
- Buffered packets
- Dropped packets

8- Save statistics in excel tables

4.4 Performance Results

This section will present the performance of the Extra Stage Omega network of the two algorithms described in chapter 2. The first algorithm is a collision-free Omega Network, which can initially detect any internal blocking and buffer these packets causing that problem, and the second algorithm, which considers wavelength converters in addition to buffering. Also, we will present the comparison between the performance of the two different networks ESO network and Omega network by using the different algorithms. The performance of these algorithms have been analyzed by simulation.

The Simulator consists of two parts. The first one randomly creates 10,000 permutations. For each of them, the simulator also creates a set of inputs, which has at most 8 random packets with different wavelengths, for each input channel of an 8x8 ESO network. We run the program for different arrival load. The second part simulates the behavior of an ESO Network. Also, it reads the data generated by the first part and generates the performance parameters such as *packet dropping probability, buffering probability and wavelength conversion probability*.

An 8x8 ESO network is considered in the simulation. It is assumed that each input link can carry at most ω packets, each with a different wavelength. The arriving set of packets on each input link is assumed to be independent, and has a Uniform distribution.

4.4.1 Extra Stage Omega (ESO) Network Performance

Considering the first architecture with limited number of buffers, Fig. 21 illustrates the probability of packet dropping versus the arrival load. As is expected, the packet dropping probability decreases with increasing number of buffers.

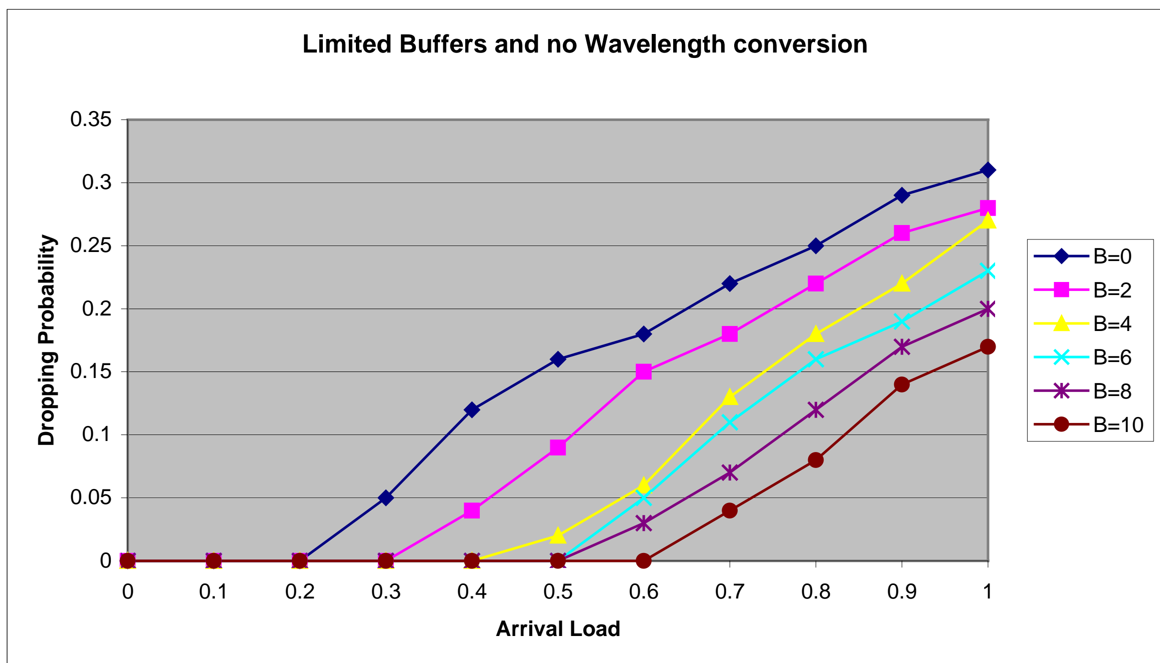


Fig. 21 Packet Dropping Probability versus Arrival Load (Limited Buffers and no Wavelength Conversion).

Fig. 22 illustrates the probability of buffering versus arrival load. We define the probability of buffering as: when a random packet arrives to this network, what is the probability that this packet will be buffered.

Initially, probability of buffering for different number of buffers increase linearly with the network load because the load of the network is satisfied with the available buffers. However, at some point these curves start to saturate and then go down. The reason is that the number of packets in the network becomes very large with respect to the available buffers and some of them are dropped. Also, Fig 22 shows that the point of saturation moves to the right with the increase of buffers.

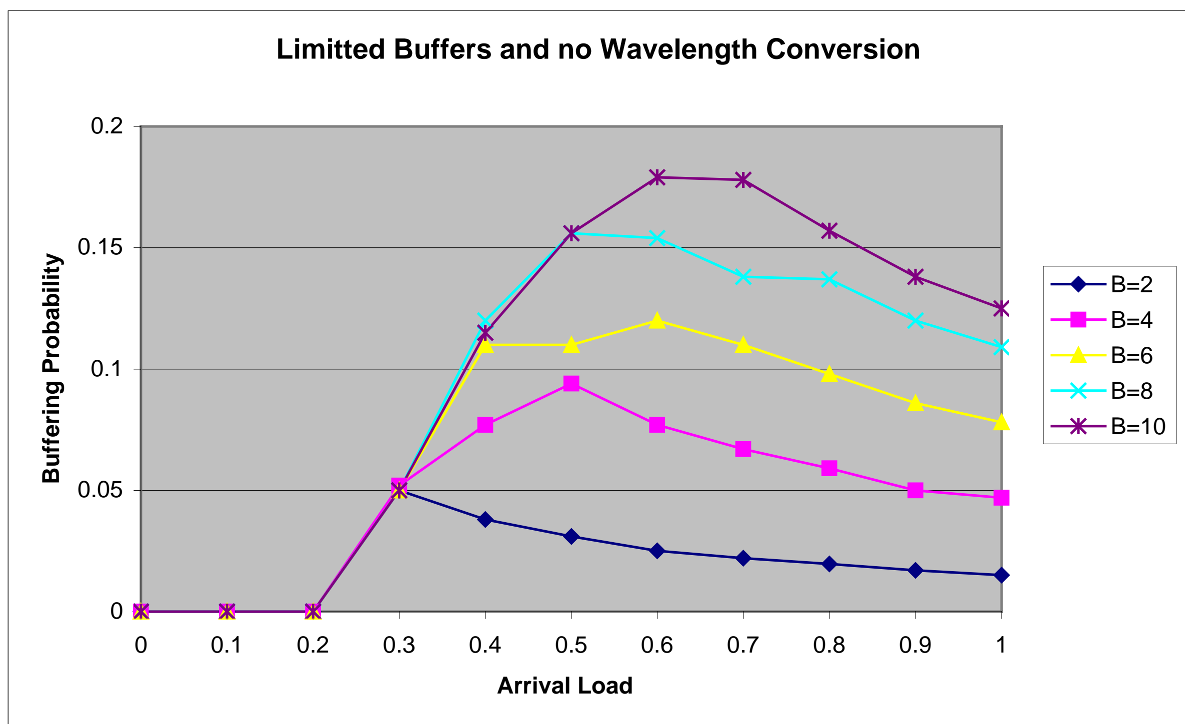


Fig. 22 Packet Buffering Probability versus Arrival Load (Limited Buffers and no Wavelength Conversion).

If we use wavelength converters, we can obtain a considerable improvement. Fig. 23 illustrates the packet dropping probability versus arrival load. It shows that using just two wavelength converters will

improve the system performance. As is expected, the packet dropping probability decreases with increasing number of wavelength converters. Note that when the arrival load is equal one, the packet dropping probability will have no improvement over a network without wavelength converters, since all wavelengths are utilized and none of them are available for conversion.

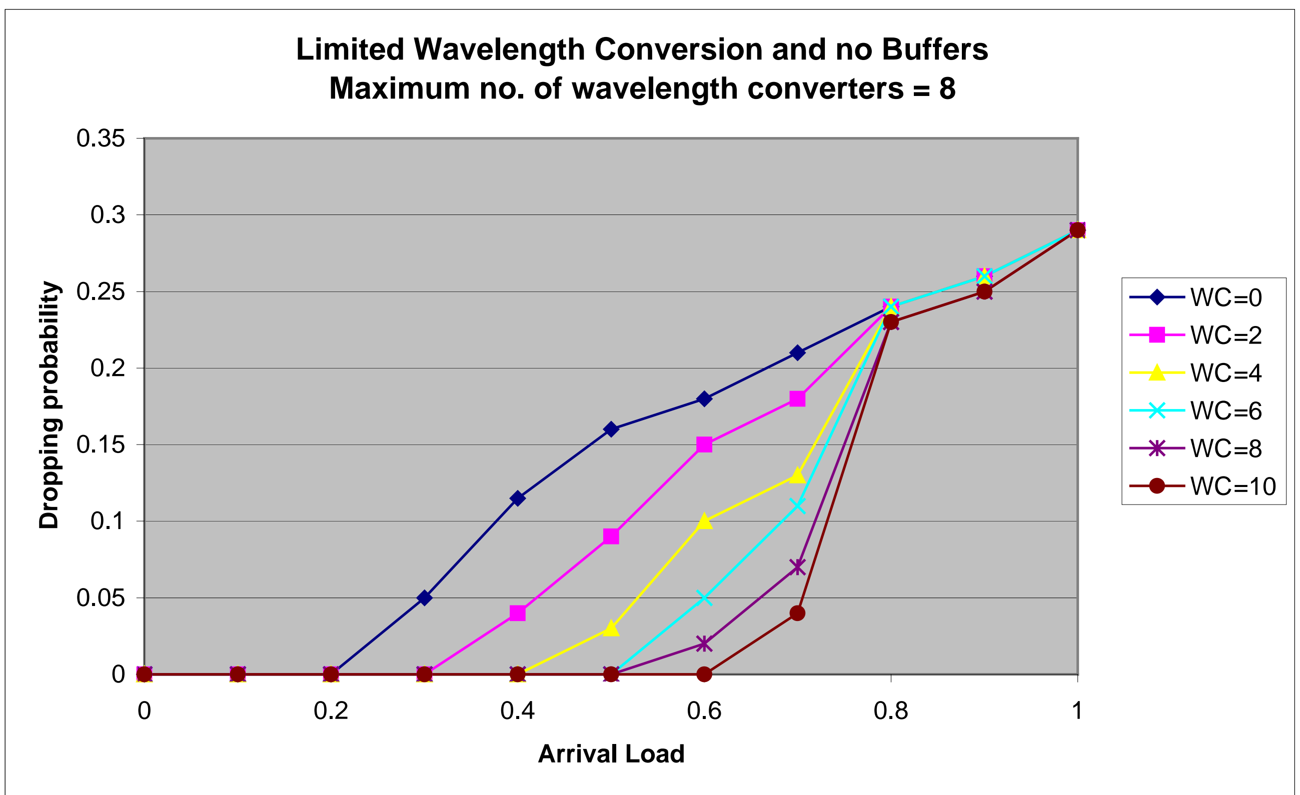


Fig. 23 Packet Dropping Probability versus Arrival Load (Limited Wavelength converters and no buffering).

Fig. 24 illustrates wavelength conversion probability versus arrival load. As is expected, the wavelength conversion probability will increase with increase of wavelength converters. Initially, when the arrival load is low no need for wavelength conversion, then all curves increase linearly

because the system is satisfied with the available wavelength converters. Then the curves saturate, and the wavelength conversion probability starts to decrease till reach to 0 when the arrival load is equal one, since there is no wavelength available to convert a wavelength to another one. Again note the point of saturation moves right with the increase of wavelength converters.

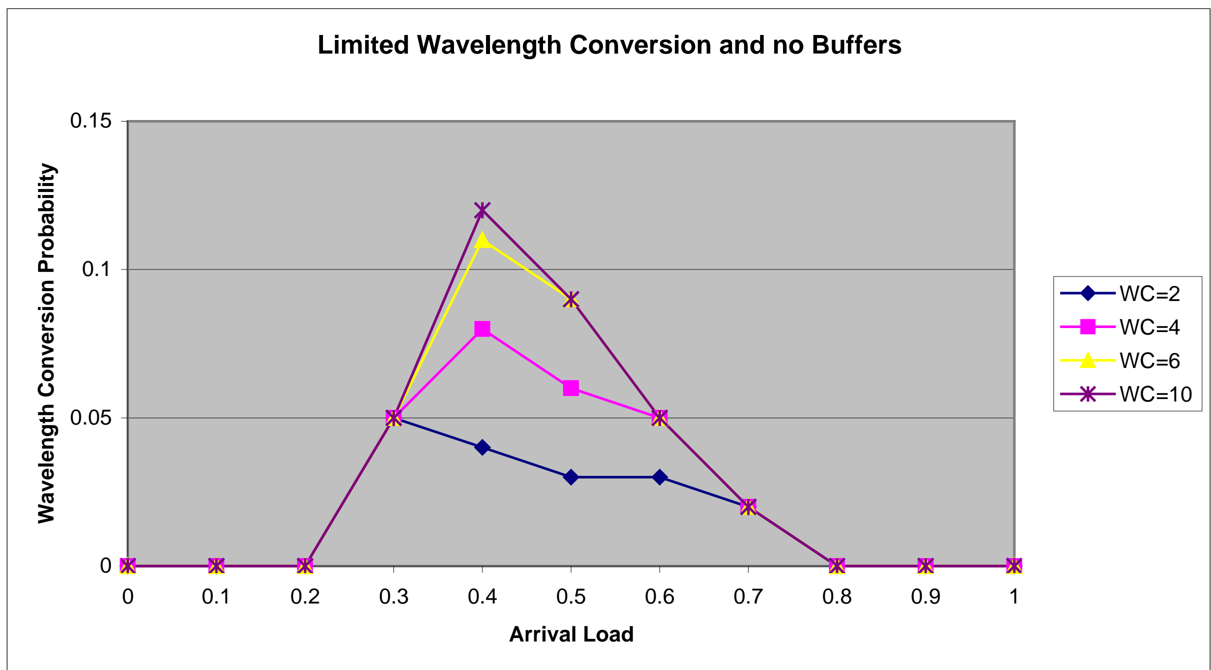


Fig. 24 Wavelength Conversion Probability versus Arrival Load (Limited Wavelength Converters and no buffering).

Fig. 25 shows a comparison among Extra Stage Omega network with different configurations. It shows that the packet dropping probability for a system with 2 buffers is better than a system with 2 wavelength converters. The reason behind that is we may have a wavelength

converter at a certain time but we may not have any available wavelength to convert to. When the arrival load is equal to one, you may notice that there are different points; each reflects a system with a certain number of buffers, and the number of wavelength converters does not have any affect on the system performance. However, when the load of a network is low to medium, you can notice the improvement by increasing number of available wavelength converters with a constant number of buffers. This improvement represents a load of packets that have been transmitted through the network in the current switching cycle, rather than buffering or even dropping them.

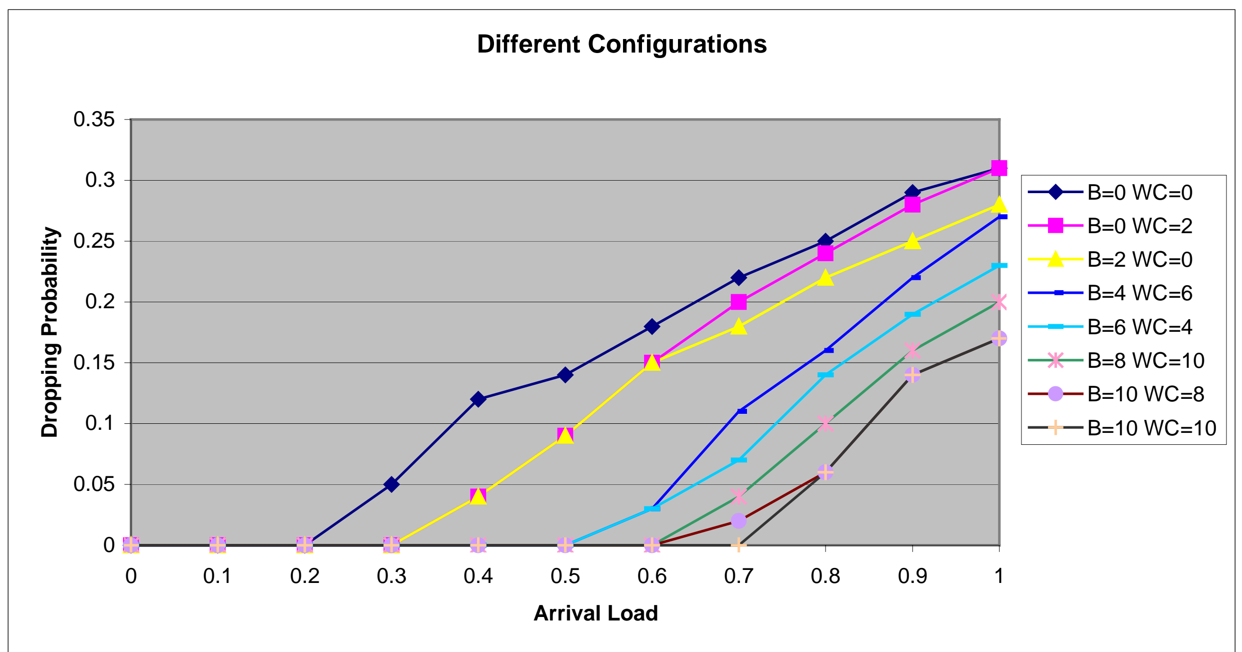


Fig. 25 Packet Dropping Probability for different configurations
(Limited Buffers and Wavelength converters).

4.4.2 Comparison between the performance of ESO network and Omega network

Considering the first architecture with a limited number of buffers. Fig. 26 illustrates the probability of packet dropping versus arrival load for the ESO and Omega networks. As was expected, the ESO network provided better performance compared to the Omega one. The reason for the performance advantage of the ESO network can be attributed to the extra stage, which provides an additional path from each source to each destination. Having an additional path gives packets more flexibility, thus reducing the probability of packet dropping.

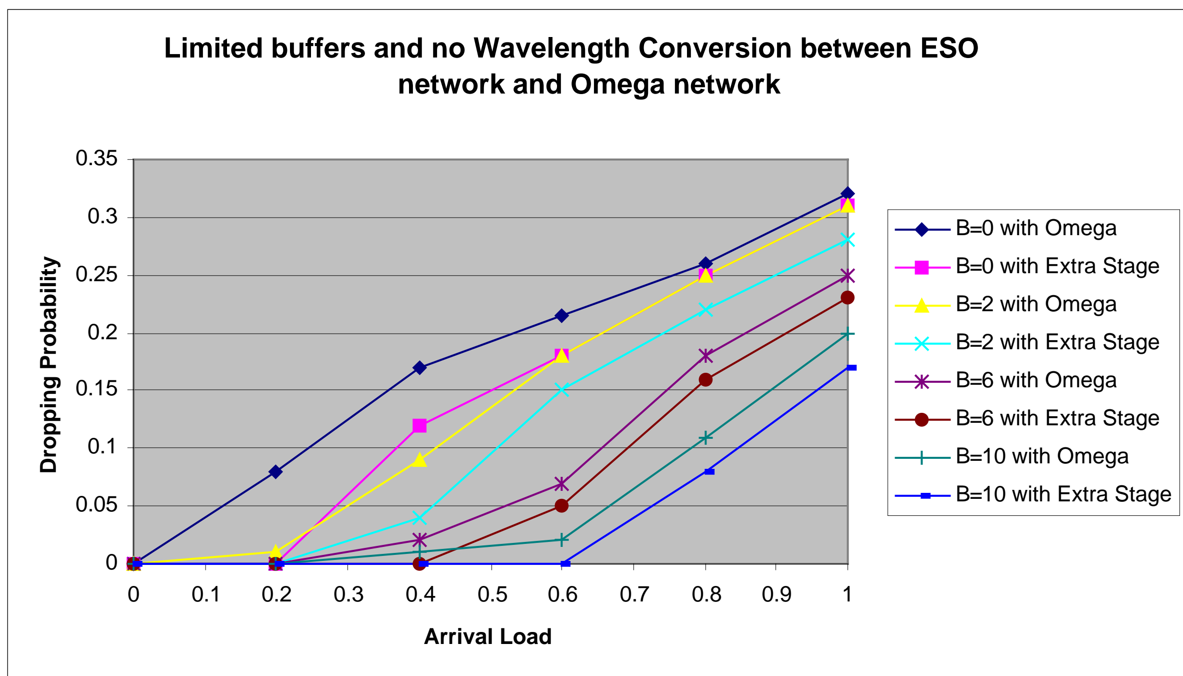


Fig. 26 Packet Dropping Probability versus Arrival Load (Limited Buffers and no wavelength conversion).

Fig. 27 illustrates the probability of buffering versus arrival load for the ESO and Omega networks. We define the probability of buffering in such a way that when a random packet arrives to the network, what is the probability that this packet will be buffered. Initially, the probability of buffering for different number of buffers increases linearly with the network load because the load of the network is satisfied with the available buffers. However, at some point in time these curves saturate and then go down. The reason is that the number of packets in the network becomes very large with respect to the available buffers and some of them are dropped. Also, Fig. 27 shows that the buffering probability of Omega network is higher than the ESO network for the same number of buffers.

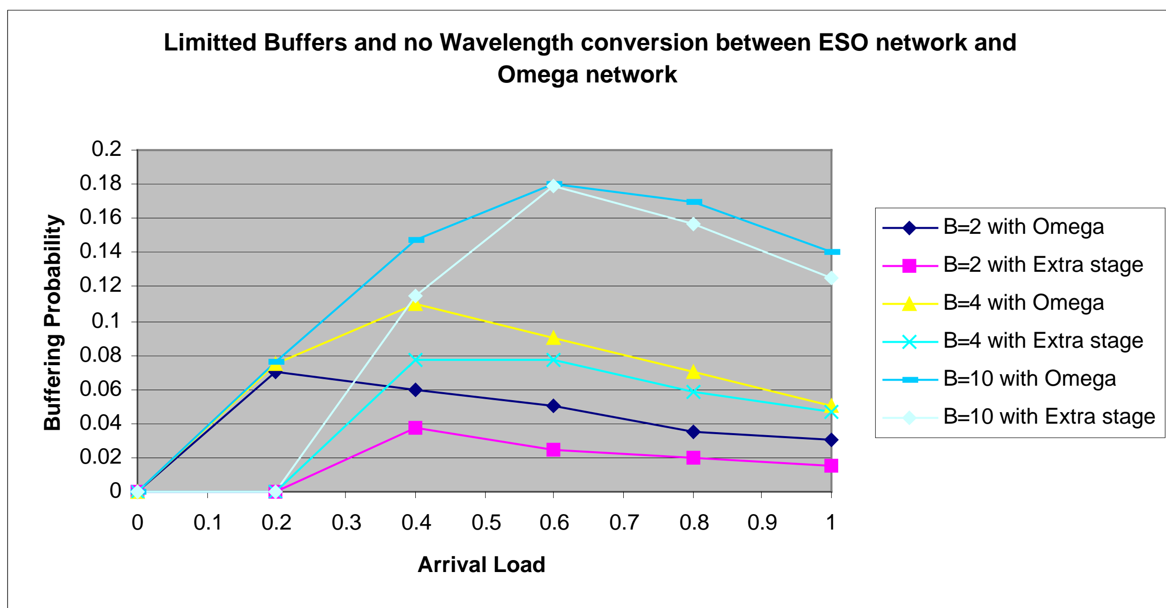


Fig. 27 Packet Buffering Probability versus Arrival Load (Limited Buffers and no Wavelength converters).

If we use wavelength converters, we can obtain a considerable improvement. Fig. 28 illustrates the packet dropping probability versus arrival load for the ESO and Omega networks. Also, Fig. 28 shows that the dropping probability of the ESO network is lower than the dropping probability of the Omega network for the same reason for the same number of wavelength converters.

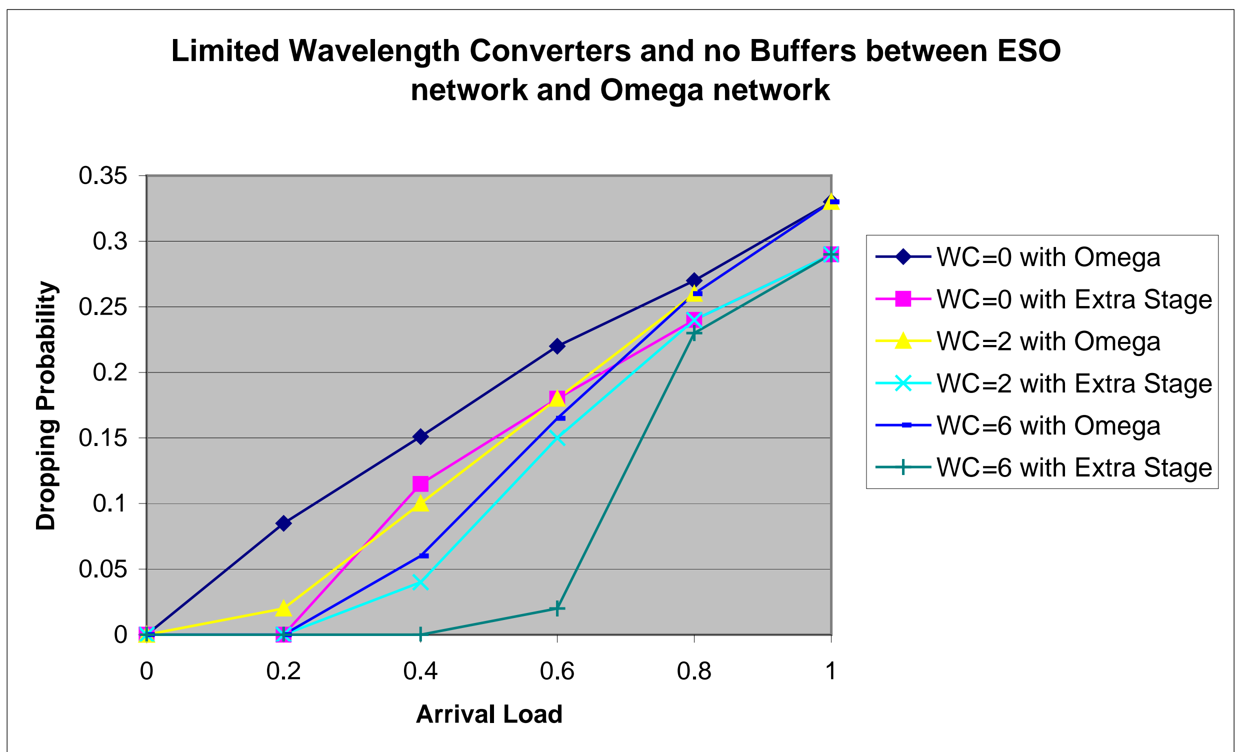


Fig. 28 Packet Dropping Probability versus Arrival Load
(Limihted Wavelength converters and no buffering).

Fig. 29 Illustrates wavelength conversion probability versus arrival load for the ESO and Omega networks. As is expected, the wavelength conversion probability will be decreased by the ESO network more than the Omega one for the same number of wavelength converters.

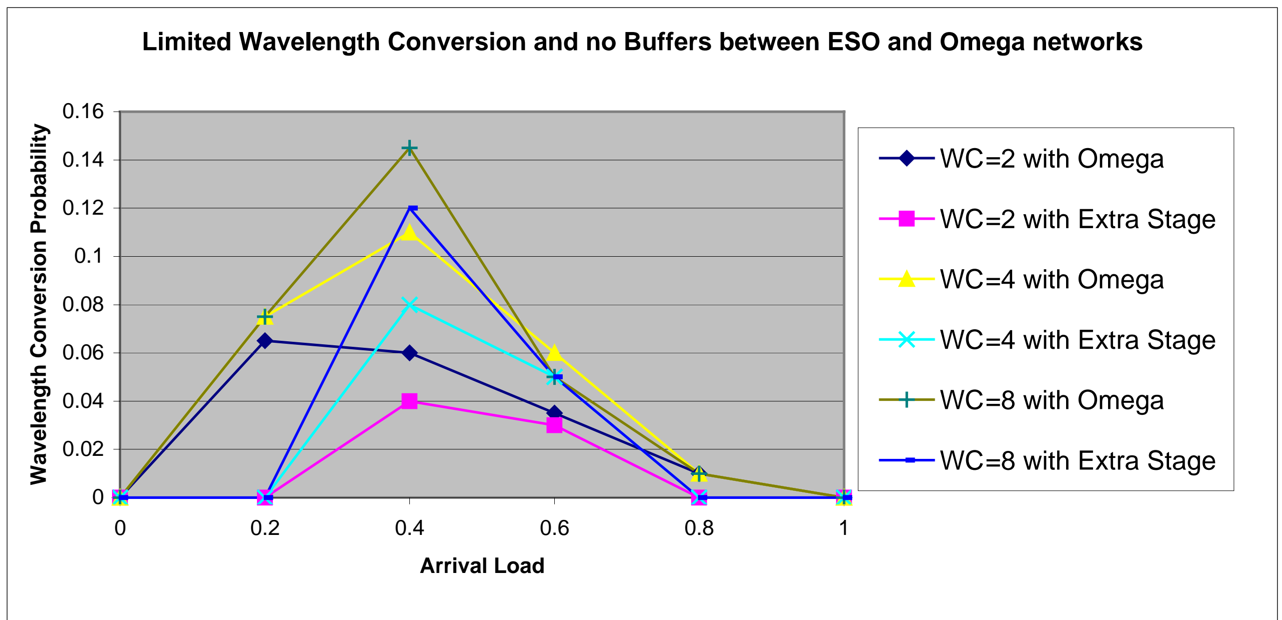


Fig. 29 Wavelength Conversion Probability versus Arrival Load (Limited Wavelength Converters and no buffering).

Fig. 30 shows a comparison between the ESO and Omega networks with different configurations. It also shows that the packet dropping probability of the ESO is better than the Omega network with the same configuration for the same reason.

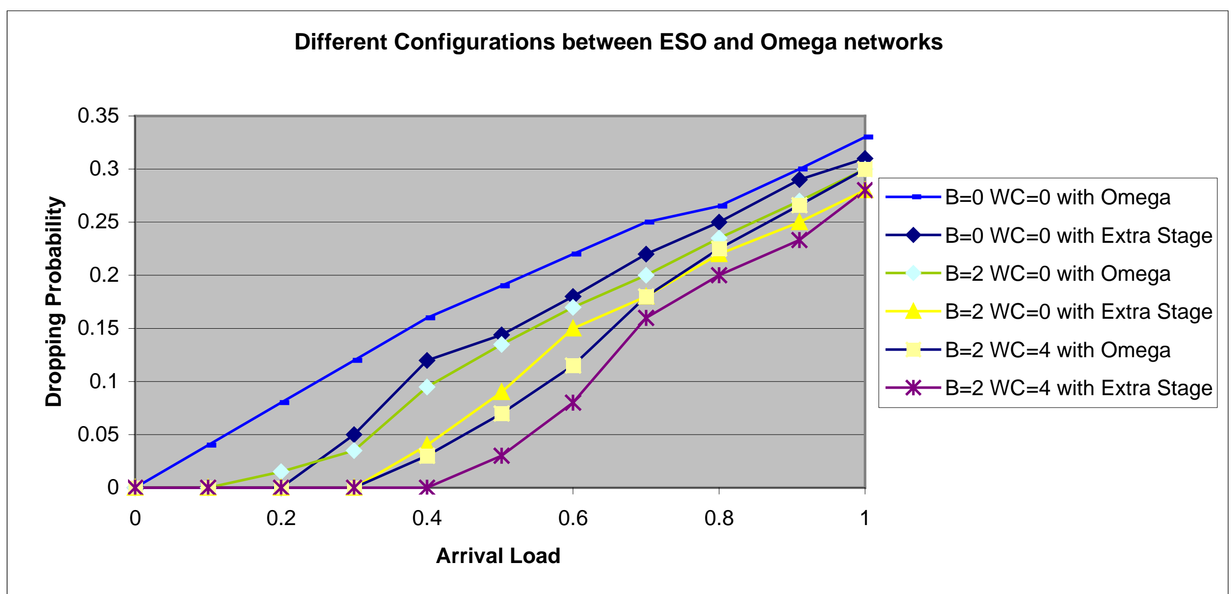


Fig. 30 Packet Dropping Probability for different configurations (Limited Buffers and Wavelength Converters).

Chapter 5

Conclusion and Future Research

5.1 Conclusion

In this research we apply several algorithms to resolve the problem of internal blocking in Extra Stage Omega (ESO) interconnection network, a fault-tolerant structure derived from Omega Network by adding an extra stage. The extra stage provides an additional path from each source to each destination. The concept of these algorithms are based on a central controller which acts as an interface in front of the ESO. Once the central controller resolves the internal blocking by buffering, wavelength conversion, or dropping, it directs the packets through the network without any collision.

We analyze the performance of the ESO networks with modification to a 2x2 switch element to incorporate the new configurations, namely the splitters and the mergers. We have found that the structure of the central control unit is very simple to implement with minimal buffers available only at the central controller rather than being distributed over all 2x2 switch elements. Therefore, we consider centralized buffers as a basis of our research to increase the utilization of

the buffers. In addition, we add wavelength converters in the central controller. Rather than buffering a packet in the current switching cycle, we convert its wavelength to another wavelength to enable the Extra Stage Omega network to accommodate it. This has increased the load of transmission and increased the utilization of the ESO network.

Then, we consider more realistic assumptions for the central controller. We assume that the permutations between a set of inputs and a set of outputs are random, and in addition, the buffered packets in the present switching cycle will be considered with newly arrived packets in the next cycle. Using these assumptions, we analyze the performance of the ESO network. We found a considerable improvement in the network performance using centralized buffers and wavelength converters.

Also, a comparison was made between the performances of the ESO and Omega networks. The ESO network provided better performance compared to the Omega one. The reason for the performance advantage of the ESO can be attributed to the extra stage, which provides an additional path from each source to each destination. Having an additional path gives packets more flexibility, thus reducing the probability of packet dropping.

Finally, the reliability of large-scale multiprocessor supersystems is a function of system structure and the fault tolerance of system components. Fault-tolerance intercommunication network can aid in achieving satisfactory reliability. The ESO has the capabilities of the Omega network plus fault tolerance for a relatively low additional cost. Thus, the ESO has the potential for being a useful interconnection network for large-scale parallel/distributed supersystems.

5.2 Future Research

Other problems related to the ESO networks based on WDM can be studied.

5.2.1 ESO Network with single failure

We have assumed that the ESO network without any failure. Therefore, we want to study the performance of the ESO network with single failure.

5.2.2 ESO Networks with each input has packets to different destinations

Initially, we have assumed that all packets of any input channel should be forwarded to a single destination. Therefore, we want to relax this restriction, and to investigate the possibility of developing an algorithm such that packets of an input channel can have different destinations.

APPENDIX A : THE 8x8 EXTRA STAGE OMEGA SIMULATOR

```
// The Extra Stage Omega Network.cpp



// includes
#include "stdAfx.h" // microsoft-specific include.
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <limits.h> // for definition of the maximum integer INT_MAX
#include <time.h> // for time functions
#include <afx.h> // for directory functions

// constants and global variables
#define NUMBER_OF_RUNS 100
#define MAX_FREE_BUFFERS 10
#define MAX_FREE_CONVERTERS 10
#define NUMBER_OF_STAGES 4
#define NUMBER_OF_BUNDLES 8
#define NUMBER_OF_PACKETS_PER_BUNDLE 8
#define NUMBER_OF_TOTAL_PACKETS
(NUMBER_OF_BUNDLES *
NUMBER_OF_PACKETS_PER_BUNDLE)
```

```
typedef struct tag_Channel
{
        int numberOfInputBundles;
        int inputBundles[NUMBER_OF_BUNDLES];
} Channel;

char *statisticFunctionDescription[] =
{
        "minimum",
        "maximum",
        "average",
};

enum statisticFunctions
{
        MINIMUM,
        MAXIMUM,
        AVERAGE,
};

char *packetStateDescription[] =
{
        "converted",
        "buffered",
        "dropped",
        "transmitted",
};
```

```
enum statisticStates
{
                        CONVERTED,
                        BUFFERED,
                        DROPPED,
                        TRANSMITTED,
};

#define NUMBER_OF_PACKET_STATES
(sizeof(packetStateDescription) / sizeof(packetStateDescription[0]))
#define NUMBER_OF_STATISTIC_FUNCTIONS
(sizeof(statisticFunctionDescription) /
sizeof(statisticFunctionDescription[0]))


// in-line functions
#ifndef max
#define max(a, b) (( (a) > (b) ) ? (a) : (b) )
#endif
#ifndef min
#define min(a, b) (( (a) < (b) ) ? (a) : (b) )
#endif
#define getConnection(i) (( (i) * 2 >= NUMBER_OF_BUNDLES ) ? (i) *
2 - NUMBER_OF_BUNDLES + 1 : (i) * 2)
#define isEvenNumber(i) (( (i) % 2 == 0) ? true : false)
```

```
// functions
void displaySummary(int summary[])
{
                printf("\nPacket summary:\n");
                for (int state = 0; state <
NUMBER_OF_PACKET_STATES; ++state)
                {
                    printf("%s=%03d ",
packetStateDescription[state], summary[state]);
                }
                printf("\n\n");
}

void displayPackagesOfInputBundles(int
inputBundles[][NUMBER_OF_PACKETS_PER_BUNDLE])
{
                printf("\nInput packets:");
                for (int bundle = 0; bundle <
NUMBER_OF_BUNDLES; bundle++)
                {
                    printf("\nbundle %02d:", bundle);
                    for (int packet = 0; packet <
NUMBER_OF_PACKETS_PER_BUNDLE; ++packet)
                    {
                         printf(" %1d",
inputBundles[bundle][packet]);
                    }
                }
                printf("\n");
```

```
}

void displayBundleRouting(Channel
channels[][NUMBER_OF_BUNDLES])
{
                printf("\nBundles on each input channel after
routing:");
                for (int stage = 0; stage < NUMBER_OF_STAGES;
++stage)
                {
                    printf("\nStage %02d:", stage);
                    for (int channel = 0; channel <
NUMBER_OF_BUNDLES; ++channel)
                    {
                        printf(" %d{", channel);
                        int numberOfInputBundles =
channels[stage][channel].numberOfInputBundles;
                        for (int inputBundle = 0; inputBundle <
numberOfInputBundles; ++inputBundle)
                        {
                            printf("%d",
channels[stage][channel].inputBundles[inputBundle]);
                            if (inputBundle <
numberOfInputBundles - 1)
                            {
                                printf(",");
                            }
                        }
                        printf("}");
```

```
            }
        }
        printf("\n");
}

void displayOutputPermutation(int permutation[])
{
        printf("\nOutput channel permutation:\n");
        for (int bundle = 0; bundle <
NUMBER_OF_BUNDLES; ++bundle)
        {
            printf("%02d ", permutation[bundle]);
        }
        printf("\n");
}

int findFreePacketSpaceInBundles(int destinationPackets[], int
inputBundles[][NUMBER_OF_PACKETS_PER_BUNDLE], int
sourceBundle)
{
        for (int packet = 0; packet <
NUMBER_OF_PACKETS_PER_BUNDLE; ++packet)
        {
            // if the current space in both packages is free
            if (destinationPackets[packet] == 0 &&
inputBundles[sourceBundle][packet] == 0)
            {
                return packet;
            }
```

```
			}
			return -1;
}

void mergeBundles(int summary[], int &freeBuffers, int
&freeConverters, int
inputBundles[][NUMBER_OF_PACKETS_PER_BUNDLE], int
destinationPackets[], int sourceBundle)
{
			// merge all packets from source bundle into
destination bundle
			for (int packet = 0; packet <
NUMBER_OF_PACKETS_PER_BUNDLE; ++packet)
			{
				// if source and destination packet exist
(wavelength overlapping)
				if (inputBundles[sourceBundle][packet] == 1
&& destinationPackets[packet] == 1)
				{
					// find free position in destination bundle
					int freePosition =
findFreePacketSpaceInBundles(destinationPackets, inputBundles,
sourceBundle);

					// if found and a converter is free
					if (freePosition != -1 && freeConverters
> 0)
					{
```

```
                                        // shift converted package in source
bundle

                    inputBundles[sourceBundle][packet] = 0;

                    inputBundles[sourceBundle][freePosition] = 1;

                                        // if the free position was already
processed, set package in destination bundle.
                                        // Else it will be set there
automatically later on, without causing a second conflict.
                                        if (freePosition < packet)
                                        {

                    destinationPackets[freePosition] = 1;
                                        }

                                        // update statistics and reduce
number of free converters.
                                        --freeConverters;
                                        ++summary[CONVERTED];
                                    }
                                    else if (freeBuffers > 0) // if a buffer is
free
                                    {
                                        // buffer package
                                        --freeBuffers;
                                        ++summary[BUFFERED];
```

```
                    inputBundles[sourceBundle][packet] = 0;
                            }
                            else // if no converter or buffer is free
                            {
                                    // drop package.
                                    ++summary[DROPPED];
                                    --summary[TRANSMITTED];

                    inputBundles[sourceBundle][packet] = 0;
                            }
                        }
                        else if (inputBundles[sourceBundle][packet] ==
1) // if source packets exist, but no destination packet
                        {
                                destinationPackets[packet] = 1;
                        }
                        //else if no source packet exist do nothing
                    }
}

void computePacketPassing(int summary[], int freeBuffers, int
freeConverters, int
inputBundles[][NUMBER_OF_PACKETS_PER_BUNDLE], Channel
channels[][NUMBER_OF_BUNDLES])
{
                    for (int stage = 1; stage < NUMBER_OF_STAGES;
++stage)
                    {
```

```
        for (int channel = 0; channel <
NUMBER_OF_BUNDLES; ++channel)
        {
            // if there is more than one input bundle
on a channel, a merge must be done
            Channel *pCurrentChannel =
&channels[stage][channel];
            int numberOfInputBundles =
pCurrentChannel->numberOfInputBundles;
            if (numberOfInputBundles > 1)
            {
                // initialize merged bundle:
                int
mergedPackages[NUMBER_OF_PACKETS_PER_BUNDLE] = {0};
                int firstInputBundle =
pCurrentChannel->inputBundles[0];
                for (int packet = 0; packet <
NUMBER_OF_PACKETS_PER_BUNDLE; ++packet)
                {
                    mergedPackages[packet] =
inputBundles[firstInputBundle][packet];
                }

                // merge all other bundles into it.
                // While merging, packages might
be converted, buffered or dropped, that means also the summary might
change.
                for (int inputBundle = 1;
inputBundle < numberOfInputBundles; ++inputBundle)
```

```
                                        {
                                             mergeBundles(summary,
freeBuffers, freeConverters, inputBundles, mergedPackages,
pCurrentChannel->inputBundles[inputBundle]);
                                        }
                                   }
                              }
                         }
}

bool computeBundleRouting(Channel
channels[][NUMBER_OF_BUNDLES], int permutation[], int bundle, int
stage, int channel)
{
                         // compute upper and lower channel of current
switch (output and input)
                         int upperSwitchChannel = 0;
                         int lowerSwitchChannel = 0;
                         if (isEvenNumber(channel))
                         {
                              upperSwitchChannel = channel;
                              lowerSwitchChannel = channel + 1;
                         }
                         else
                         {
                              upperSwitchChannel = channel - 1;
                              lowerSwitchChannel = channel;
                         }
```

```
// if it is the end stage:
if (stage == NUMBER_OF_STAGES - 1)
{
    // if bundle arrived at the correct destination
    if (bundle ==
permutation[upperSwitchChannel] || bundle ==
permutation[lowerSwitchChannel])
    {
        // add input bundle to current input
bundles
        int
numberOfInputBundlesOfCurrentChannel =
channels[stage][channel].numberOfInputBundles;

        channels[stage][channel].inputBundles[numberOfIn
putBundlesOfCurrentChannel] = bundle;

        ++channels[stage][channel].numberOfInputBundles;

        // if it is the last bundle
        if (bundle == NUMBER_OF_BUNDLES
- 1)
        {
            return true;
        }
        else // continue with next bundle on first
stage
        {
```

```
                                        int pathWasFound =
computeBundleRouting(channels, permutation, bundle + 1, 0,
getConnection(bundle + 1));

                                        // if path was not found, delete
input bundle from current input bundles
                                        if (! pathWasFound)
                                        {

                    channels[stage][channel].numberOfInputBundles =
numberOfInputBundlesOfCurrentChannel;
                                              return false;
                                        }

                                        return true;
                                  }
                            }
                            else // if bundle arrived at the wrong destination
                            {
                                  return false;
                            }
                      }
                      // else if we are not at the end stage:

                      // compute connected next-stage channels of switch
outputs
                      int upperConnection =
getConnection(upperSwitchChannel);
```

```
int lowerConnection =
getConnection(lowerSwitchChannel);

// get number of bundles for all input and output
channels of current switch
int numberOfInputBundlesOfUpperConnection =
channels[stage + 1][upperConnection].numberOfInputBundles;
int numberOfInputBundlesOfLowerConnection =
channels[stage + 1][lowerConnection].numberOfInputBundles;
int numberOfInputBundlesOfUpperChannel =
channels[stage][upperSwitchChannel].numberOfInputBundles;
int numberOfInputBundlesOfLowerChannel =
channels[stage][lowerSwitchChannel].numberOfInputBundles;
int numberOfInputBundlesOfCurrentChannel =
channels[stage][channel].numberOfInputBundles;

// check if a way is blocked,
// that means if there are no bundles at the current
channel and there is at least one bundle at each connection
if (numberOfInputBundlesOfUpperConnection > 0
&& numberOfInputBundlesOfLowerConnection > 0 &&
numberOfInputBundlesOfCurrentChannel == 0)
{
    return false;
}

// add input bundle to current input bundles
channels[stage][channel].inputBundles[numberOfIn
putBundlesOfCurrentChannel] = bundle;
```

```
                ++channels[stage][channel].numberOfInputBundles;

                // check if there is only one determined way to go,
                // that means if there is at least one input bundle at
each channel of the current switch
                int pathWasFound = false;
                if (numberOfInputBundlesOfUpperChannel > 0 &&
numberOfInputBundlesOfLowerChannel > 0)
                {
                    // look where the first input bundle at the
current channel routes to, and use the same way.
                    int firstBundle =
channels[stage][channel].inputBundles[0];
                    int currentConnection = upperConnection;
                    for (int bundleIndex = 0; bundleIndex <
numberOfInputBundlesOfLowerConnection; ++bundleIndex)
                    {
                        int outputBundle = channels[stage +
1][lowerConnection].inputBundles[bundleIndex];
                        if (outputBundle == firstBundle)
                        {
                            currentConnection =
lowerConnection;
                        }
                    }
                    pathWasFound =
computeBundleRouting(channels, permutation, bundle, stage + 1,
currentConnection);
                }
```

```
else // else try both possible paths
{
        int firstConnection = upperConnection;
        int secondConnection = lowerConnection;
        // choose upper or lower way by random if both
output channels are used by same number of bundles.
        // This will especially improve the performance
for setups with many stages where many solutions exist,
        // and will find a solution with least bundle
merging and therefore package loss.
        if (numberOfInputBundlesOfLowerConnection
== numberOfInputBundlesOfUpperConnection)
        {
                if (rand() % 2 == 0)
                {
                        firstConnection =
lowerConnection;
                        secondConnection =
upperConnection;
                }
        }
        // to avoid as much bundle merging as possible,
use least used channel as the first way.
        else if
(numberOfInputBundlesOfUpperConnection >
numberOfInputBundlesOfLowerConnection)
        {
                        firstConnection =
lowerConnection;
```

```
                                        secondConnection =
upperConnection;
                                }
                                // else if
(numberOfInputBundlesOfUpperConnection <
numberOfInputBundlesOfLowerConnection) keep the default.

                                // try both paths
                                pathWasFound =
computeBundleRouting(channels, permutation, bundle, stage + 1,
firstConnection);
                                if (! pathWasFound)
                                {
                                        pathWasFound =
computeBundleRouting(channels, permutation, bundle, stage + 1,
secondConnection);
                                }
                        }

                        // if path was not found, delete input bundle from
current input bundles
                        if (! pathWasFound)
                        {

                        channels[stage][channel].numberOfInputBundles =
numberOfInputBundlesOfCurrentChannel;
                                return false;
                        }
```

```
                    return true;
}

void generateRandomOutputPermutation(int permutation[])
{
                    // place all bundles in sorted order in array
                    for (int bundle = 0; bundle <
NUMBER_OF_BUNDLES; ++bundle)
                    {
                         permutation[bundle] = bundle;
                    }

                    // scramble bundle order
                    for (int bucketSize = NUMBER_OF_BUNDLES;
bucketSize > 1; --bucketSize)
                    {
                         // draw a random bundle number of the bucket
and swap it with the one at the end of the bucket
                         int currentBucketIndex = rand() % bucketSize;
                         int lastBucketIndex = bucketSize - 1;
                         int lastBundle = permutation[lastBucketIndex];
                         permutation[lastBucketIndex] =
permutation[currentBucketIndex];
                         permutation[currentBucketIndex] = lastBundle;
                    }
}
```

```
void generateRandomInputBundles(int
inputBundles[][NUMBER_OF_PACKETS_PER_BUNDLE], int
numberOfPackets)
{
                    // place all packets in sorted order in array
                    for (int packet = 0; packet < numberOfPackets;
++packet)
                    {
                         int currentBundle = packet /
NUMBER_OF_PACKETS_PER_BUNDLE;
                         int currentPacketInBundle = packet %
NUMBER_OF_PACKETS_PER_BUNDLE;

                    inputBundles[currentBundle][currentPacketInBundl
e] = 1;
                    }

                    // scramble packet order
                    for (int bucketSize =
NUMBER_OF_TOTAL_PACKETS; bucketSize > 1; --bucketSize)
                    {
                         // draw a random packet out of the bucket and
swap it with the one at the end of the bucket
                         int currentBucketIndex = rand() % bucketSize;
                         int currentBucketBundleIndex =
currentBucketIndex / NUMBER_OF_PACKETS_PER_BUNDLE;
                         int currentBucketPacketIndex =
currentBucketIndex % NUMBER_OF_PACKETS_PER_BUNDLE;
                         int lastBucketIndex = bucketSize - 1;
```

```
                    int lastBucketBundleIndex = lastBucketIndex /
NUMBER_OF_PACKETS_PER_BUNDLE;
                    int lastBucketPacketIndex = lastBucketIndex %
NUMBER_OF_PACKETS_PER_BUNDLE;
                    int lastPacket =
inputBundles[lastBucketBundleIndex][lastBucketPacketIndex];

                inputBundles[lastBucketBundleIndex][lastBucketPa
cketIndex] =
inputBundles[currentBucketBundleIndex][currentBucketPacketIndex];

                inputBundles[currentBucketBundleIndex][currentBu
cketPacketIndex] = lastPacket;
                }
}

bool runAndComputeSummary(int summary[], int freeBuffers, int
freeConverters)
{
                // generate random output permutation
                int permutation[NUMBER_OF_BUNDLES] = {0};
                generateRandomOutputPermutation(permutation);
                displayOutputPermutation(permutation);

                // compute bundle routing (set input bundles for each
switch)
                // start with bundle 0 on channel 0 of first stage.
                // the function will go through all stages recursively
and then continue the path with the next bundle until the last bundle.
```

```
		Channel
channels[NUMBER_OF_STAGES][NUMBER_OF_BUNDLES] = {0};
		bool pathWasFound =
computeBundleRouting(channels, permutation, 0, 0, 0);
		if (! pathWasFound)
		{
			printf("\nUnable to find path for bundle
routing.\nRetrying again with new output permutation:\n");
			return false;
		}

		displayBundleRouting(channels);

		// generate bundles with random input packets
		int
inputBundles[NUMBER_OF_BUNDLES][NUMBER_OF_PACKETS_P
ER_BUNDLE] = {0};
		generateRandomInputBundles(inputBundles,
summary[TRANSMITTED]);

		displayPackagesOfInputBundles(inputBundles);

		// compute packet passing
		computePacketPassing(summary, freeBuffers,
freeConverters, inputBundles, channels);

		// display summary
		displaySummary(summary);
```

```
                return true;
}

void saveStatistic(int freeBuffers, int freeConverters, int
statistic[][NUMBER_OF_PACKET_STATES][NUMBER_OF_TOTAL
_PACKETS])
{
                char *errorMessage = "\nUnable to write to file
%s.\nPress 'Enter' to exit:";

                // open file
                char filename[100] = {0};
                sprintf(filename,
"Configuration_Buffers=%02d_Converters=%02d.csv", freeBuffers,
freeConverters);
                FILE *stream = fopen(filename, "w");
                if (stream == NULL)
                {
                    printf(errorMessage, filename);
                    getchar();
                    exit(1);
                }

                // save statistic values
                for (int state = 0; state <
NUMBER_OF_PACKET_STATES; ++state)
                {
                    // save table header ("converted packets", or
"buffered packets", ...)
```

```
				int numberOfBytesWritten = fprintf(stream,
"%s packets (buffers=%2d converters=%2d)\n",
packetStateDescription[state], freeBuffers, freeConverters);
				if (numberOfBytesWritten <= 0)
				{
					printf(errorMessage, filename);
					getchar();
					exit(1);
				}

				// save column headers (number of input
packets):
				// save first column header ("input")
				numberOfBytesWritten = fprintf(stream, "%-
15s, ", "input");
				if (numberOfBytesWritten <= 0)
				{
					printf(errorMessage, filename);
					getchar();
					exit(1);
				}
				// save second up to last column header (number
of input packets)
				for (int input = 0; input <
NUMBER_OF_TOTAL_PACKETS; ++input)
				{
					numberOfBytesWritten = fprintf(stream,
" %3d,", input + 1);
					if (numberOfBytesWritten <= 0)
```

```
				{
					printf(errorMessage, filename);
					getchar();
					exit(1);
				}
			}
			// save newline character
			numberOfBytesWritten = fprintf(stream, "\n");
			if (numberOfBytesWritten <= 0)
			{
				printf(errorMessage, filename);
				getchar();
				exit(1);
			}

			for (int function = 0; function <
NUMBER_OF_STATISTIC_FUNCTIONS; ++function)
			{
				// save row header ("minimum", or
"maximum", ...)
				numberOfBytesWritten = fprintf(stream,
"%-15s, ", statisticFunctionDescription[function]);
				if (numberOfBytesWritten <= 0)
				{
					printf(errorMessage, filename);
					getchar();
					exit(1);
				}
```

```
					// save all comma-separated values
(columns) of one row
					for (int input = 0; input <
NUMBER_OF_TOTAL_PACKETS; ++input)
					{
						numberOfBytesWritten =
fprintf(stream, " %3d,", statistic[function][state][input]);
						if (numberOfBytesWritten <= 0)
						{
							printf(errorMessage,
filename);
							getchar();
							exit(1);
						}
					}

					// save newline character
					numberOfBytesWritten = fprintf(stream,
"\n");
					if (numberOfBytesWritten <= 0)
					{
						printf(errorMessage, filename);
						getchar();
						exit(1);
					}
				}
			}

			//close file
```

```
                fclose(stream);
}

void computeGraph(int freeBuffers, int freeConverters)
{
                // initialize all statistic values
                int
statistic[NUMBER_OF_STATISTIC_FUNCTIONS][NUMBER_OF_PA
CKET_STATES][NUMBER_OF_TOTAL_PACKETS] = {0};
                for (int state = 0; state <
NUMBER_OF_PACKET_STATES; ++state)
                {
                    for (int input = 0; input <
NUMBER_OF_TOTAL_PACKETS; ++input)
                    {
                         statistic[MINIMUM][state][input] =
INT_MAX;
                    }
                }

                // compute y-coordinates for all x-coordinates of the
graph
                for (int input = 0; input <
NUMBER_OF_TOTAL_PACKETS; ++input)
                {
                    // initialize sum
                    int sum[NUMBER_OF_PACKET_STATES] =
{0};
```

```
// compute y-coordinates for one x-
coordinate(=input) of the graph
for (int run = 0; run < NUMBER_OF_RUNS;
++run)
{
    // display number of run and total input
packets
    printf("\nTotal input packets=%03d,
run=%04d:\n", input + 1, run);

    // initialize summary values
    int
summary[NUMBER_OF_PACKET_STATES] = {0};
    summary[TRANSMITTED] = input + 1;

    // This function does the main work.
    // It generates a random output
permutation,
    // generates random input packet bundles,
    // computes packet routing (network
switches), and
    // computes packet passing (sets
summary[state])
    bool success =
runAndComputeSummary(summary, freeBuffers, freeConverters);
    if (! success)
    {
        // No path was found, so do the
same run again with different output permutation
```

```
                                                --run;
                                                continue;
                                        }

                                        // compute minimum and maximum of
statistic
                                        for (int state = 0; state <
NUMBER_OF_PACKET_STATES; ++state)
                                        {
                                                sum[state] += summary[state];
                                                statistic[MINIMUM][state][input]
= min(statistic[MINIMUM][state][input], summary[state]);
                                                statistic[MAXIMUM][state][input]
= max(statistic[MAXIMUM][state][input], summary[state]);
                                        }
                                }



                                // compute average of statistic
                                for (int state = 0; state <
NUMBER_OF_PACKET_STATES; ++state)
                                {
                                        statistic[AVERAGE][state][input] =
sum[state] / NUMBER_OF_RUNS;
                                }
                        }

                        // save statistics in excel tables
                        saveStatistic(freeBuffers, freeConverters, statistic);
```

```
}

int main()              // The main program
{
                        // display welcome message
                        printf("Omega Network computation started\n");

                        // init random number generator
                        time_t secondsSince1970 = time(NULL);
                        srand((unsigned int)secondsSince1970);

                        // create directory for statistics and change to it.
                        struct tm *now = localtime(&secondsSince1970);
                        char directoryName[30];
                        sprintf(directoryName,
"Statistic_%04d_%02d_%02d_%02dh%02dm%02ds", now->tm_year +
1900, now->tm_mon + 1, now->tm_mday, now->tm_hour, now-
>tm_min, now->tm_sec);
                        CreateDirectory(directoryName, NULL);
                        SetCurrentDirectory(directoryName);

                        // compute all graphs for different configurations and
save the statistic
                        for (int freeBuffers = 0; freeBuffers <=
MAX_FREE_BUFFERS; ++freeBuffers)
                        {
                            for (int freeConverters = 0; freeConverters <=
MAX_FREE_CONVERTERS; ++freeConverters)
                            {
```

```
					printf("\nComputing graph for
configuration with:\nbuffers=%02d converters=%02d\n", freeBuffers,
freeConverters);
					computeGraph(freeBuffers,
freeConverters);
				}
			}

			// display exit message and wait for user's key press
			printf("\nOmega Network computation
stopped\nPress 'Enter' to exit:");
			getchar();

			return 0;
}
```

REFERENCE

[1] Mohammed Amer Arafah, "Centralized Buffering and Wavelength Conversion In Multistage Interconnection Networks", A PHD Dissertation, University of Southern California, December 1997.

[2] Duncan H. Lawrie, "Access Alignment of Data in an Array Processor", IEEE Transaction on Computer, vol. C-24, No. 12, Dec. 1975.

[3] Janak M. Patel, "Processor-Memory Interconnection for Multiprocessor", Proc. 16th Annual Symp. On Computer Architecture, April 1979.

[4] George Adams, III, and Howard J. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection for Supersystems", IEEE Transaction on Computer., vol. C-31, No. 5. May 1982.

[5] G. Hugh Song, "Asymmetric Dilation of Multwavelenegth Cross-Connect Switches for Low-Crosstalk WDM Optical Networks", IEEE Journal of Lightwave Technology, vol. 15, No. 3, March 1997.

[6] H. Scot Hinton, "Switching to photonics", IEEE Spectrum, February 1992.

[7] H. Scott Hinton, "Photonic Switching Fabrics", IEEE Communication Magazine, PP.71-88, April 1990.

[8] C. B. Stunkel and et. la. The SP2 Communication Subsystem. Technical report, IBM Thomas J. Watson Research Center, Aug 1994.

[9] N. Koike. NEC Cenju-3: A Microprocessor-Based Parallel Computer. In Proc. Of the Int'l Parallel Processing Sumposium, pages 396-401, April 1994.

[10] G. B. Adams III, D. P. Agrawal, and H. J. Siegal. "A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks", IEEE Computer, 20:14-27, 1987.

[11] A. Varma ans C. S. Raghavendra. “Fault-tolerant Routing in Multistage Interconnection Networks”, IEEE Transactions on Computers, 38(3):385-393, Mar. 1989.

[12] Kuochen Wang and Feng-Ming Lin, “ Design and Implementation of a Fault Tolerant Switch”, Journal of Information Science and Engineering 15, 521-541 1999.

[13] V. E. Bense, “Mathematical Theory of Connecting Networks and Telephone Traffic, Acadmic Press, New York, 1965.

[14] Chiung-Shien et al., “Extended Baseline Architecture for Nonblocking Photonic Switching”, IEEE Journal of Lightwave Technology, vol. 15, No.5, May 1997.

[15] Kuo-Chun Lee, “Wavelength Conversion and Wavelength Routing in All-Optical Networks”, A PHD Dissertation, Electrical Engineering, University of Southern California, August 1994.

[16] Janak H. Patel, “Performance Of Processor-Memory Interconnection for Multiprocessors”, IEEE trans. On Computers, vol. C-30, Oct. 1981.