Universidad Politécnica de Valencia
Departamento de Informática de Sistemas y Computadores

Low-Memory Techniques for Routing and Fault-Tolerance on the Fat-Tree Topology

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

Author

Crispín Gómez Requena

Advisors

María Engracia Gómez Requena
Pedro Juan López Rodríguez

Valencia, Julio de 2010

Acknowledgments

There have been a lot of people that have supported and helped me in several ways during all these years at the university. Most of them have greatly influenced my research and changed me as a person. I could not thank all of them, but I want to remark the support of several people.

First and foremost, I thank my parents and brothers for being always there. There are so many reasons to thank them that I should write an encyclopedia to enumerate them. Just to sum up, I would like thank my mother and father for being the strongest people I have ever met, I hope one day I could be half strong as them. I would like to thank my brother for introducing me in the world of the computers, I have spent most of my childhood watching him playing in the computer, and also thanks for all the great moments that we have had playing together. I would like to thank my sister for directing me during all my PhD, since she is indeed one of my advisors, and for all the great moments we have enjoyed working together. Also, I want to note that her children (Belén and Juan) are the joy of my live.

In second place, I would like to thank my friends, they all could make me laugh during the hard moments, so I could continue working to finish the PhD. Mainly, I would like to thank Paco, Merlyx, Blas and Diego. Paco is almost like a member of my family, I spend most of my free time playing or watching movies with him. Also, we have traveled around the world together, and we have several moments that I will remember forever. I would like to thank Merlyx (Paco's wife) for overfeeding me during our weekly D&D game, and for being such a kind person. I would like to thank Blas for introducing me to the "Futbol Xtreme", a sport that is only meant to we played by real men who can risk their lives in every single football match. Finally, I would thank Diego for being such a freak, and traveling with me every year to the “Salò del Manga de Barcelona”.

Of course, I would like to thank to my advisors, my sister (María Engracia) and Pedro López, for their guidance during all these years. This dissertation would not have been possible without all the time that we have spent discussing, and without their valuable knowledge. I hope that someday I could know the interconnection networks as well as you do. Like a friend told me once: “I have learned a great deal by simply observing their excellent example and behavior. I do not have any doubt that my research and non-research life has forever been changed by their advice”. Also, I would like to thank Prof. José Duato for allowing me to be a part of the big family composed by all the members of the Parallel Architecture Group. Many of the other faculty members in the Parallel Architecture Group have helped me too, thanks to you all.

I have met many interesting students while in the lab. Although I cannot possibly mention everyone who has enriched my experience or provided moral support, I wish to specifically thank a few individuals: Gaspar, Andrés (both are lost in the mists of Intel), Blas, Paco, Samuel, Héctor, Carles, Rafa, and David. They all have unselfishly helped me in different aspects of my research and I would like to highlight their support. Finally, I have to highlight the amazing work of Ricardo, he is the one responsible of keeping our clusters up and running, and the one that help us when our computers start to act crazy.

Contents

List of Figures

List of Tables

Abstract

Currently, clusters of PCs are considered a cost-effective alternative to large parallel computers. In these systems, thousands of computing nodes are connected through a high-performance interconnection network. The interconnection network must be carefully designed, since it heavily impacts the performance of the whole system. Two of the main design parameters of the interconnection networks are topology and routing. Topology defines the interconnection of the elements of the network among themselves, and between them and the computing nodes. Routing defines the paths followed by the packets through the interconnection network.

Performance has traditionally been the main metric to evaluate the interconnection network. However, we have to consider two additional metrics nowadays: cost and fault-tolerance. Interconnection networks have to scale in terms of cost, in addition to scale in performance. That is, they not only need to maintain their performance as the system size is increased, but also without heavily increasing their cost. On the other hand, as the number of nodes increases in cluster-based machines, the interconnection network grows accordingly. This increase in the number of elements of the interconnection network raises the probability of faults, and thus, fault-tolerance has become mandatory for current interconnection networks.

This dissertation focus on the fat-tree topology, which is one of the most-commonly used topologies for clusters. Our aim is to exploit its characteristics to provide fault-tolerance; and a load-balanced routing algorithm that provides a good cost/performance tradeoff.

First, we focus on the fault-tolerance of the fat-tree topology. Most of the works in the literature provide fault-tolerance at the cost of adding resources to the network, either switches, links or virtual channels. On the contrary to these works, we provide the same degree of fault-tolerance without increasing the resources of the network by taking advantage of the abundant plurality of equivalent paths in the fat-tree. In particular, we define a mechanism that avoids the use of those paths that lead to the faulty elements, in such a way that packets that would cross the faults are deviated through non-faulty paths to their destination. As all the non-faulty paths can be used without any restriction, the mechanism presents a low performance degradation due to faults while providing the maximum degree of fault-tolerance that can be achieved without adding new resources to the network.

Next, we take the challenge of designing a deterministic routing algorithm that can compete in terms of performance with the adaptive routing algorithms that are used for the fat-tree topology. Although, adaptive routing algorithms need more resources to be implemented than deterministic ones, they usually outperform the latter ones. With our work, we present a deterministic routing algorithm that obtains similar - or even better- performance than the best adaptive routing algorithm, but at a lower cost, since it does not require the use of a selection function in each switch. Furthermore, when in-order delivery of packets is required, adaptive routing algorithms require the utilization of additional mechanisms, since deterministic routing algorithms preserve the delivery order of the packets by design. The results will show that our proposed deterministic routing algorithm can clearly outperform adaptive routing when in-order delivery of packets is required.

Finally, we take advantage of the particular characteristics of the proposed deterministic routing algorithm to simplify the fat-tree topology. This topology almost halves the resources required by the fat-tree topology, achieving in many cases the same performance as the fat-tree. In general, the cost/ performance ratio of the new topology is almost half of the cost/performance ratio of the fat-tree.

Resumen

Actualmente, los clústeres de PCs están considerados como una alternativa eficiente a la hora de construir ordenadores con un alto grado de paralelización. En estos sistemas, miles de nodos de computación se conectan mediante una red de interconexión. La red de interconexión tiene que ser diseñada cuidadosamente, puesto que tiene una gran influencia sobre las prestaciones globales del sistema. Dos de los principales parámetros de diseño de las redes de interconexión son la topología y el encaminamiento. La topología define la interconexión de los elementos de la red entre sí, y entre éstos y los nodos de computación. Por su parte, el encaminamiento define los caminos que siguen los paquetes a través de la red.

Las prestaciones han sido tradicionalmente la principal métrica a la hora de evaluar las redes de interconexión. Sin embargo, hoy en día hay que considerar dos métricas adicionales: el coste y la tolerancia a fallos. Las redes de interconexión además de escalar en prestaciones también deben hacerlo en coste. Es decir, no sólo tienen que mantener su productividad conforme aumenta el tamaño de la red, sino que tienen que hacerlo sin incrementar sobremanera su coste. Por otra parte, conforme se incrementa el número de nodos en las máquinas de tipo clúster, la red de interconexión debe crecer en concordancia. Este incremento en el número de elementos de la red de interconexión aumenta la probabilidad de aparición de fallos, y por lo tanto, la tolerancia a fallos es prácticamente obligatoria para las redes de interconexión actuales.

Esta tesis se centra en la topología fat-tree, ya que es una de las topologías más comúnmente usadas en los clústeres. El objetivo de esta tesis es aprovechar sus características particulares para proporcionar tolerancia a fallos y un algoritmo de encaminamiento capaz de equilibrar la carga de la red proporcionando

una buena solución de compromiso entre las prestaciones y el coste.

En primer lugar, nos centramos en la tolerancia a fallos en la topología fat-tree. Muchos de los trabajos dedicados a este tema proporcionan tolerancia a fallos a cambio de añadir recursos a la red; ya sean encaminadores, enlaces o canales virtuales. Al contrario que estos trabajos, nuestra propuesta proporciona el mismo nivel de tolerancia a fallos sin incrementar los recursos de la red, aprovechando la abundancia de caminos equivalentes disponibles en el fat-tree. En concreto, se ha definido un mecanismo que evita que se usen aquellos caminos que lleven a los elementos fallidos, de forma que los paquetes que fuesen a tomar uno de dichos caminos son desviados hacia su destino por caminos que no contengan elementos fallidos. Dado que todos los caminos que no han sufrido un fallo pueden ser usados sin ninguna restricción, el mecanismo propuesto presenta una degradación de prestaciones mínima ante la presencia de fallos, mientras que proporciona el máximo nivel de tolerancia a fallos que se puede conseguir sin añadir nuevos recursos a la red.

A continuación nos centramos en el diseño de un algoritmo de encaminamiento determinista que pueda competir en prestaciones con los algoritmos de encaminamiento adaptativos usados en la topología fat-tree. Pese a que los algoritmos adaptativos requieren más recursos que los deterministas a la hora de ser implementados, los algoritmos adaptativos suelen tener unas prestaciones mayores que los deterministas. En nuestro trabajo presentamos un algoritmo de encaminamiento determinista que puede obtener prestaciones similares, o mayores, que el mejor algoritmo de encaminamiento adaptativo, pero a un coste menor, puesto que no requiere el uso de mecanismos complementarios, ya que los algoritmos de encaminamiento determinista proporcionan entrega en orden de los paquetes por definición. Los resultados mostrarán que el algoritmo de encaminamiento determinista propuesto puede sobrepasar claramente las prestaciones de los algoritmos adaptativos cuando la entrega en orden de los paquetes es obligatoria.

Finalmente, nos aprovechamos de ciertas características de dicho algoritmo de encaminamiento determinista para simplificar la topología fat-tree. La nueva topología prácticamente reduce a la mitad los recursos necesarios para construir un fat-tree, a la vez que proporciona en muchos casos las mismas prestaciones. En general, la relación entre costes y prestaciones de la nueva

topología es casi la mitad que la del fat-tree.

Resum

Actualment, els clusters de PCs estan considerats com una alternativa eficient a l'hora de construir ordinadors amb un alt grau de paralelització. En aquests sistemes, milers de nodes de computació es connecten per mitjà d'una xarxa d'interconnexió. La xarxa d'interconnexió ha de ser dissenyada cuidadosament, ja que té una gran influència sobre les prestacions globals del sistema. Dos dels principals paràmetres de disseny de les xarxes d'interconnexió són la topologia i l'encaminament. La topologia definix la interconnexió dels elements de la xarxa entre si, i entre aquestos i els nodes de computació. Per la seua banda, l'encaminament definix els camins que seguixen els paquets a través de la xarxa.

Les prestacions han sigut tradicionalment la principal mètrica a l'hora d'avaluar les xarxes d'interconnexió. No obstant això, hui en dia cal considerar dos mètriques addicionals: el cost i la tolerància a fallades. Les xarxes d'interconnexió a més d'escalar en prestacions també han de fer-ho en cost. s a dir, no sols han de mantindre la seua productivitat conforme augmenta la grandària de la xarxa, sinó que han de fer-ho sense incrementar en una gran medida el seu cost. D'altra banda, conforme s'incrementa el nombre de nodes en les màquines de tipus cluster, la xarxa d'interconnexió ha de créixer en concordança. Aquest increment en el nombre d'elements de la xarxa d'interconnexió augmenta la probabilitat d'aparició de fallades, i per tant, la tolerància a fallades és pràcticament obligatòria per a les xarxes d'interconnexió actuals.

Esta tesis es centra en la topologia fat-tree, ja que és una de les topologies més comunament usades en els clusters. El nostre propòsit consistix a aprofitar-nos de les seues característiques particulars per a proporcionar tolerància a fallades i un algoritme d'encaminament capaç d'equilibrar la càrrega de la xarxa que proporcione una bona solució de compromís entre les prestacions i el cost.

En primer lloc, ens centrem en la tolerància a fallades en la topologia fat-tree. Molts dels treballs dedicats a aquest tema proporcionen tolerància a fallades a canvi d'afegir recursos a la xarxa; ja siguen encaminadors, enllaços o canals virtuals. Al contrari que aquests treballs, nosaltres proporcionem el mateix nivell de tolerància a fallades sense incrementar els recursos de la xarxa, aprofitant-nos de l'abundància de camins equivalents disponibles en el fat-tree. En concret, definim un mecanisme que evita que s'usen aquells camins que porten als elements fallits, de manera que els paquets que anessen a prendre un d'eixos camins són desviats cap al seu destí per camins que no continguen elements fallits. Atés que tots els camins que no han patit una fallada poden ser usats sense cap restricció, el mecanisme presenta una degradació de prestacions mínima davant de la presència de fallades, mentres que proveïx el màxim nivell de tolerància a fallades que es pot aconseguir sense afegir nous recursos a la xarxa.

Posteriorment, ens centrem en el disseny d'un algoritme d'encaminament determinista que pot competir en prestacions amb els algoritmes d'encaminament adaptatius usats en la topologia fat-tree. A pesar que els algoritmes adaptatius requerixen més recursos que els deterministes a l'hora de ser implementats, els algoritmes adaptatius solen tindre unes prestacions majors que els deterministes. En el nostre treball presentem un algoritme d'encaminament determinista que pot obtindre prestacions semblants, o majors, que el millor algoritme d'encaminament adaptatiu, però a un cost menor, ja que no requerix l'ús de mecanismes complementaris, ja que els algoritmes d'encaminament determinista proporcionen entrega en orde dels paquets per definició. Els resultats mostraran que l'algoritme d'encaminament determinista proposat pot sobrepassar clarament les prestacions dels algoritmes adaptatius quan l'entrega en orde dels paquets és obligatòria.

Finalment, ens aprofitem de certes característiques d'aquest algoritme d'encaminament determinista per a simplificar la topologia fat-tree. La nova topologia pràcticament reduïx a la mitat els recursos necessaris per a construir un fat-tree, al mateix temps que proporciona en molts casos les mateixes prestacions. En general, la relació entre costos i prestacions de la nova topologia és quasi la mitat que la del fat-tree.

Chapter 1

Introduction

"You take the blue pill, the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill, you stay in Wonderland, and I show you how deep the rabbit hole goes."

Morpheus, The Matrix.

In this chapter, we describe the reasons that have motivated this dissertation (Section 1.1). Then, we briefly define the objectives aimed by the dissertation (Section 1.2). Finally, we conclude this chapter by presenting the structure of this thesis (Section 1.3).

1.1 Motivation

Nowadays, large parallel computers with hundreds of thousands of nodes [2,4, 126] are required to meet the computational requirements of a growing set of applications. These applications need such computational capabilities mainly due to two main reasons: the complexity of the application itself or the number of concurrent users of the application. In the first group of applications, we can include all the applications that help in resolving very complex problems, such as, high definition multimedia encoding, protein sampling, simulations of the universe, weather forecast, and so on. These applications for their own nature are very performance consuming; the more resources they can get, the sooner they can end.

On the other hand, the applications that belong to the second group do not need extremely high performance capabilities, but their concurrent utilization boosts their requirements. For instance, a web page server that supports dynamic pages with access to a database does not require a high-performance computer to be executed. However, if the number of users skyrockets, the requirements to be able to concurrently answer all the petitions from the users in a reasonable time are also extremely increased. This is the case of the Internet auction webpage eBay [83], or Google, that is powered by four hundred thousand personal computers [89].

Most of the high-performance computers are cluster-based computers. Cluster popularity is so high that 82% of the computers of the Top 500 list of June 2009 are clusters [12]. Due to this fact, this dissertation is focused on this group of high-performance computers.

As it can be seen, high-performance computing is now widely spread and it is involved in daily activities, like searching a web page on the Internet. High-performance computing has suffered a tremendous growth, since at the beginning it was only applied on scientific and military fields. Nevertheless, this tremendous growth has brought several new challenges to this field that in the beginning of high-performance computing did not seem so important. These new challenges are mainly reducing the overall system cost and providing fault-tolerance.

Concerning the first challenge, a high-performance cluster may cost several millions of dollars. For instance, the price of a single BlueGene/P [126] rack is 4.5 millions of dollars [29], a full cluster with Intels Dual Core processors costs 18.4 millions of dollars [29]. Hence, one of the goals that a designer of a high-performance cluster must keep in mind is to reduce the system cost as much as possible or at least avoid increasing it unnecessarily. System cost can be broken down mainly in two components [68]: the cost of all the elements that compose it and the development and design costs, both highly related to the system complexity [68].

The first component may be high, due to the huge number of elements that compose the whole system. In this cost, designers have to consider all the processing nodes; the interconnection network (that includes switches, links, and network interfaces); disk servers; backup servers; and so on. Just to have a glance on the number of elements in a cluster, lets consider the RoadRunner cluster [10], jointly developed by IBM and LANL, that is composed by 12960 IBM PowerXCell 8i CPUs and 6480 AMD Opteron dual-core processors. The cost of these processors by themselves is very high, but to this cost we have to add the cost of the interconnection network, the shared disk system and the managing servers. The interconnection network adds 26 288-port switches and 7488 quadruple links to the total number of elements [69]. As it can be seen, the total number of components in a high-performance computer is enormous. So, anything that helps in reducing the cost of the components will have a great impact on the total system cost, since they are highly replicated. This is why in the last years, many research works and developments have been focused on reducing the cost of the clusters while keeping their performance [62, 92, 98].

The second component of the system cost is the development and design cost. High-performance clusters are becoming so complex that development and design costs can only be defrayed by hundreds of dollars projects. For example, the RoadRunner cluster has been a 6 year project with an initial budget of 100 millions of dollars, in which hundreds of people at LANL and IBM have been working on. As it can be easily deduced, any step that helps in reducing the complexity of such systems will have a direct impact on reducing their development cost, and, therefore, in their final cost.

Moreover, these machines have another cost that is also very important, the operational and maintenance cost [29, 92]. This extra cost includes the power consumed by the cluster, the cooling system required to cool down the computer, and several other issues related to the special facilities where it has to be installed at. In total, this operational cost may be from 3 to 5 millions of dollars per year [29]. Moreover, the power consumption problem in computers is so extreme that 8% of the total power in the United States of the America in 2007 was consumed by computers [92]. Due to this fact, in the last years, research on reducing the power consumed by the system has become of great interest. This can be done in several ways, but most of them are based on the issues that have been aforementioned: reduce the complexity of the system or some of its components, or decrease the number of elements without penalizing the system performance [88, 92, 99].

As mentioned before, the second challenge of high-performance clusters is to deal with their high fault probability due to the large number of components that compose them. The global probability of suffering a single fault in a cluster increases linearly with the number of its elements, since each of them can independently fail. As it has been aforementioned, high-performance clusters are built with a very large number of elements, so their global fault probability is very high, and, therefore, fault-tolerance is a design key issue. Moreover, the importance of fault-tolerance is increased if the cluster cost is taken into account, since a cluster that costs several millions of dollars cannot be stopped due to the presence of a single fault.

This dissertation is focused in the interconnection network of cluster machines since it covers a high percentage of the system hardware, as it has been shown in the RoadRunner example. Furthermore, it plays a key role in the overall system performance [33, 40, 41]. Moreover, failures in the interconnection network may isolate a large portion of the machine containing several processors that could otherwise be used. So, the improvements made in the interconnection network will also benefit the rest of the system, in cost, performance and fault-tolerance. Because of this fact, this dissertation is focused on the improvement of the cluster interconnection network from these three points of view: cost, performance and fault-tolerance. In order to do this, we focus on some of the classical design parameters of interconnection networks [33, 40]: topology, routing and fault-tolerance.

Topology defines the way in which the processing nodes are connected. In other words, topology defines the shape of the interconnection network. This dissertation only considers clusters based on the fat-tree topology [76], since fat-trees have risen on popularity in the past few years, and the fat-tree topology has become the default or recommended topology for most of the commercial cluster interconnect vendors (i.e., Myrinet [8], InfiniBand [7], Quadrics [9]).

Routing determines the path that each packet follows between two processing nodes through the interconnection network. Routing has a great impact on the interconnection network performance, fault-tolerance and switch complexity. In cluster-based computers, routing is usually distributed and based on forwarding tables. Distributed routing based on forwarding tables relies on implementing a table at each switch that stores the output port that must be used for each possible destination node. However, routing based on forwarding tables suffers from a lack of scalability, as table size grows linearly with the system size and the time required to access the table also depends on its size.

On the other hand, routing can be classified as deterministic or adaptive. In deterministic routing schemes, an injected packet traverses a fixed, predetermined route between source and destination nodes. In adaptive routing schemes, a packet may traverse any of the different alternative routes available from the packet source to its destination. The route for a packet is usually selected taken into account the status of the network. Adaptive routing usually better balances network traffic, thus allowing the network to obtain a higher throughput. However, with adaptive routing, in-order packet delivery cannot be ensured, which is mandatory for some applications, like some cache coherence protocols [84]. On the other hand, deterministic routing algorithms usually do a very poor job balancing traffic among the network links, due to the lack of path diversity. Nevertheless, they are usually easier to be implemented and easier to be deadlock-free. Moreover, deterministic routing guarantees in-order packet delivery by design. In general, an adaptive routing algorithm should outperform a deterministic one.

Regarding complexity, it is usual that fault-tolerance increases the complexity of the routing algorithm, and the whole system. A traditional approach for fault-tolerant routing algorithms is to start with a non-fault-tolerant routing algorithm and modify it to tolerate faults. This modifications are usually the addition of new routes for the packets, mechanisms to detect faults, mechanisms to switch the routes of the packets when faults appear, and mechanism to deal with packets that are directly affected by the faults. As the fault-tolerant version of a routing algorithm is usually an extension of the original routing algorithm, it is usually more complex than the non-fault-tolerant version.

Just to sum up, this dissertation will make a set of proposals intended for:

- Providing efficient routing in fat-trees, but, at the same time, trying to reduce the network complexity and, therefore, its cost. This is done by an efficient load-balanced deterministic routing algorithm that also eliminates the forwarding tables, in order to reduce the switch complexity and the switch power consumption.

- Improving the fat-tree fault-tolerance without increasing the network complexity by exploiting the rich connectivity of the topology.
- Reducing the hardware of the fat-tree topology in order to reduce the network complexity and the network power consumption.

1.2 Objectives

This section presents the objectives of this dissertation. As commented in the previous section, we pretend to provide simple solutions with low hardware overhead to provide fault-tolerance and efficient routing in the interconnection network of a cluster based on the fat-tree topology. In order to do this, we pursue the following goals:

- To propose a fault-tolerant routing algorithm for fat-trees that is able to tolerate the maximum possible number of faults, but considering several restrictions about the required resources:
  - The proposed fault-tolerant routing algorithm should not be based on resource replication. Resource replication is the easiest way to provide fault-tolerance in any system, but it highly increases its cost. Our proposed fault-tolerant routing algorithm should not require the inclusion of spare network resources, that is, it should not need additional switches, links or virtual channels. In other words, a cluster with our proposed fault-tolerant routing algorithm should have the same cost than the same cluster without the fault-tolerant routing algorithm in terms of network resources. This is especially interesting for large machines, where introducing new hardware will suppose an additional large cost.
  - Furthermore, our proposal should have very low-memory requirements at the switches of the interconnection network. That is, the amount of memory required to tolerate faults should be very low.

  - It should be able to respond to faults in a very short time.
  - Additionally, it should be able to only nullify the strictly necessary paths due to the faults, allowing fully adaptive routing through all the non-faulty paths. This helps in maintaining the throughput of the interconnection network as high as possible.
  - Finally, it should not disconnect any node of the network due to faults if there is still a non-faulty path that connects this node to the rest of nodes of the network. This means that our fault-tolerant routing algorithm will take advantage of all the available paths in the topology.

- To develop a deterministic routing algorithm for the fat-tree topology with the following features:

  - Very simple implementation.
  - Low-memory and hardware requirements.
  - Very short routing time.
  - Try to balance the traffic in the network. This will allow it to obtain almost the same performance than the adaptive routing algorithms usually used for fat-trees, but with the advantages of deterministic routing.

  In this way, we pretend to simplify the interconnection network, because adaptive routing requires additional hardware in the network. The challenging point in this objective is trying to keep the same performance results as adaptive routing.

- To simplify the fat-tree topology in order to reduce network resources while maintaining similar performance results to the ones achieved by the fat-tree topology. In this way, the cost (measured as the amount of hardware and power consumption) of a cluster with the new topology is highly decreased, and the cost/performance ratio is improved.

1.3 Dissertation Outline

The rest of the dissertation is organized as follows. Chapter 2 describes the fundamentals of interconnection networks and several works related to the contributions of this dissertation. Chapter 3 presents the FT^{2}EI fault-tolerant routing strategy. Chapter 4 describes a load-balanced deterministic routing algorithm for fat-trees. Chapter 5 presents RUFT, which is the resulting topology of our proposed simplification of the fat-tree topology. The dissertation ends with Chapter 6, which summarizes our contributions and their advantages.

Chapter 2

Background and State of the Art

"I don't know half of you half as well as I should like; and I like less than half of you half as well as you deserve."

Bilbo Baggins, The Lord of the Rings.

This chapter describes the basics and terminology for understanding the main aspects of interconnection networks. We do not provide an in-depth view of interconnection networks, since interconnection networks are very complex systems, which have several aspects that are not of special interest for the dissertation. For this reason, we provide a quick introduction to the interconnection networks, and then, we provide a more extensive description of the issues of interconnection networks that are required to fully understand the contributions of this dissertation. We refer the reader to the established textbooks on this topic for further background and introductory material [33, 40]. Finally, this chapter also summarizes the previous contributions that are related to the ones proposed in this dissertation.

This chapter is composed by two sections. Section 2.1 is devoted to the aforementioned background on interconnection networks. Section 2.2 provides the state of the art in fault-tolerance and routing in fat-trees.

2.1 Interconnection Networks

Interconnection networks are an essential concept in high-performance computing. High-performance computing relies on a high number of computers working together towards the same objective. This notion implies that computers are able to coordinate themselves, or be directed by specialized software. In either option, computers need to communicate among them and/or with the software layer that coordinate them. In order to support this communication, there must be some kind of infrastructure capable of transporting information among all the computers, that is, there must be an infrastructure that interconnects them all. This infrastructure is the interconnection network, which is responsible of forwarding the information among the computers of a cluster as soon as possible.

Interconnection networks have become an essential part of a high-performance computers, in order to exploit the potential performance provided by the available high number of computers [40]. The importance of interconnection networks can be easily seen with a simple comparison. Imagine a system that works with two potent engines, which need 10 liters of fuel per second to work at their maximum power. They are connected to a fuel deposit of unlimited capacity by a pipe that can deliver 2 liters of fuel per second. As it can be seen, the engines would not be able to work at their maximum performance due to the pipe low capacity. Indeed, they would only reach 10% of their maximum potency. This is the same effect that can be identified in a high-performance computer, the low capacity of the pipe, the low bandwidth of the interconnection network in our case, can strongly limit the global system performance.

Furthermore, the interconnection network does not only play a key role on the computer performance. It has been shown by several works that it is a critic point to support fault-tolerance in high-performance computers [28, 43, 70, 74, 90, 94, 115, 117, 128, 129], since a fault in the interconnection network may isolate a large part of the machine containing several expensive and healthy processing nodes.

In order to fully understand our proposals and the related works, we should provide a basic background about interconnection networks. For this, in this section we provide a small introduction to interconnection networks by presenting their components and several interconnection network design considerations. Then, the most important of these considerations for our contributions are explained with more details in the following subsections.

2.1.1 Interconnection Network Basics

The interconnection network in a high-performance computer is the communication infrastructure that forwards the information among the devices that compose it. These devices may be computers, processors or any other kind of device. The transferred information is sent from a device, called sender or origin, to another device, referred to as destination. The information that a device needs to send to another device must be packetized in order to be transferred through the network, including all the required control information for the interconnection network, like origin, destination and type of the packet. All this control information is encapsulated in the header of the packet. This work is usually done by a software protocol in the origin of the packet. Once the packet is ready to be transferred, it is copied to the network interface card (NIC) of the device. NICs are the first component of the interconnection network that we are introducing. They are responsible of injecting the packets into the network.

In order to inject packets, NICs must monitor the link of the network at which they are connected to. When the link is not being utilized, the NIC can inject a new packet into it, therefore NICs act as some kind of semaphore. NICs also make the required electrical conversions between the network and the device, since they may be implemented in different technologies. Notice that the contrary operation, that is, ejecting the packet from the network, is done by the NIC of the destination device, following a symmetric procedure.

The interconnection network is also composed by links that are the responsible of transporting the packets. They can be considered the fundamental part of the interconnection network, since they are the medium through which packets traverse the network. We refer to link as the set of wires, or any other medium, that are used to interconnect one device to another, and allow sending packets between them. In the first interconnection networks, there was only one link that connected all the NICs of the system. This kind of

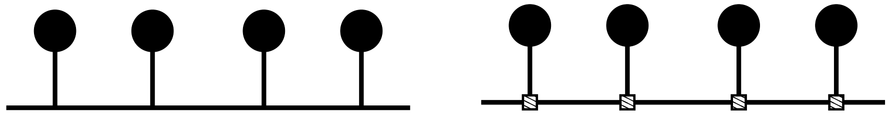

(a) Bus network. (b) Segmented bus network.

Figure 2.1: An example of shared-medium network (a), and a switch-based network (b).

interconnection networks are called shared-medium interconnection networks, and their typical and most widely-used implementation is the bus, which is represented in Figure 2.1(a). In such networks, the NICs have to be snooping the link in order to read all the packets, delivering to their corresponding device only the ones that were directed to it. Also, NICs have to ensure that the medium is free when their attached device needs to send a packet to a destination. In case that the medium is free, it just sends the packet. Otherwise, it has to store the packet while the medium is not free. This waiting time was the main cause of abandoning the paradigm of shared-medium networks, since it is proportional to the number of devices in the system, and in large systems this limitation strongly diminishes the time that each node can use the network, highly reducing the overall system performance [40, 88, 99]. This kind of networks is represented by several well-known commercial networks: Token Bus, Token Ring, and the first generations of Ethernet.

The most-widely adopted solution to avoid the growing waiting time in shared-medium networks consisted on segmenting the bus like in Figure 2.1(b), in which the bus has been segmented by inserting a switch per each node, represented as small grey boxes. These switches break the medium in smaller independent parts, thus avoiding the problem of the long waiting times in the NIC of the shared-medium networks, since in each of those parts the medium can only be accessed by a fixed number of devices, no matter the number of total devices in the system. For instance, in the Figure 2.1(b), each of the links can only be accessed by the switches that it directly connects. In most cases, in high-performance computers, links are bidirectional, allowing to have two packets using it at the same time, but each one traveling in an opposite direction. Furthermore, this also allows to simplify the NICs, since they no

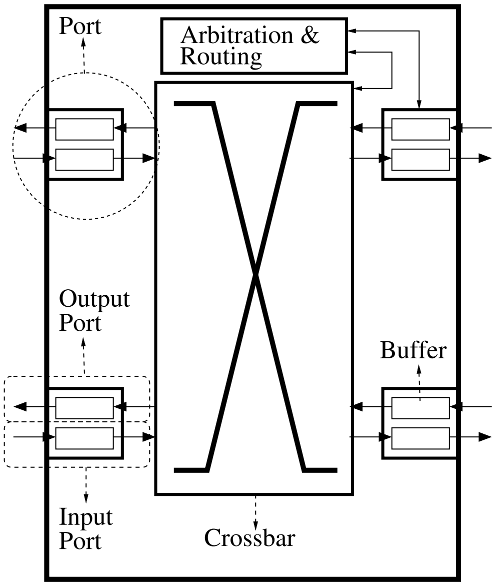


Figure 2.2: Simplified view of a 4x4 switch.

longer need to snoop the medium to know if it is free or not. As there is only one NIC per (vertical) link (see Figure 2.1(b)) the link will always be free if that NIC is not using it. So, the logic to know the state of the link becomes extremely simple.

Switches act as some kind of intelligent barrier. When a switch receives a packet from one of its links, it has to route the packet to one of its possible output links, preventing in this way to flood unnecessarily the entire network with just a message. For example, in the Figure 2.1(b), when a switch receives a packet from its right link, it has to check if the packet is directed to the device this switch is attached at. If that is the case, it just forwards the packet to the device, and the switches that are at the left of this switch will not see the packet, thus the network is more efficiently used than in the case of shared-medium networks.

Switches are composed by several elements. The most important ones are highlighted on Figure 2.2. Each of the connections between the switch and a link is made through a port of the switch. As we assume that links are bidirectional, each of these ports can be split into two symmetric ports with opposite directions. The input ports are the responsible of receiving the packets from the link, and sending the packets to the crossbar when the arbitration logic indicates it. On the other hand, output ports have to receive the packets from the crossbar of the switch, and have to forward the packets out of the switch through their connected output link. Routing logic has to decide which input port has to be connected to which output port through the crossbar. Arbitration has to avoid contention of the packets in the crossbar, allowing to connect only one input to each output port. As contention may arise, input ports implement buffers to store the packets while they are waiting to get an assigned port. On the other hand, output ports implement buffers to decouple the transmission of packets through the output link and their switch traversing. In this way, if a packet cannot be sent to the next switch, the packet is stored in the output port and its corresponding input buffer is released to be used by other packets. Crossbar is a configurable matrix of connections between input and output ports, dynamically connecting input and output ports as arbitration and routing logic demand it. Finally, we refer to the links that connect the switch to the devices or computers as injection and ejection links depending on their direction, and we refer as network links to the links that interconnect the switches of the network.

2.1.2 Interconnection Network Design Parameters

As aforementioned, interconnection networks play a major role in the design of modern high-performance computers. Nevertheless, they are not simple; there are many factors that may affect the choice of an appropriate interconnection network at design time. These factors include, but they are not restricted to, the following:

- *Performance.* As commented, performance is a key point in interconnection networks, not only from the point of view of raw throughput, also from the point of view of latency. Latency is a critical design issue in several systems such as real-time systems.

- *Scalability.* Scalability is the first design rule that an interconnect designer should keep in mind. Scalability in interconnection networks implies that the bandwidth of the network increases proportionally to the number of elements of the system. Otherwise, the interconnection network would become a bottleneck, limiting the efficiency of the whole high-performance computer. Scalability also implies that network cost and resources are proportional to the network size.

- *Reliability.* An interconnection network should be able to deliver information in a reliable way. Interconnection networks should be designed for continuous operations in the presence of a limited number of faults.

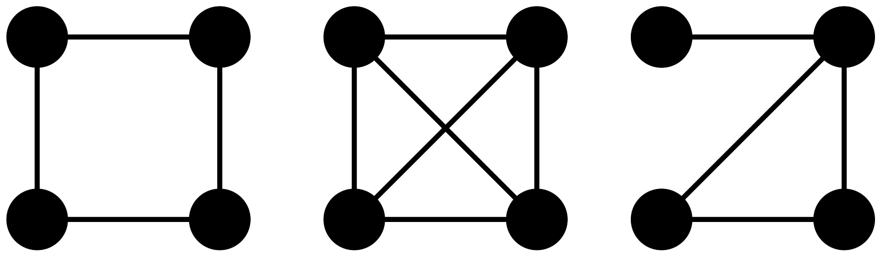
- *Cost.* Cost is becoming one of the most important constrains for interconnection networks, as commented previously. Often, the best network for a given design is too expensive, and the designers have to make trade-offs between cost and performance.

- *Simplicity.* Not only for the sake of cost, also because simpler designs can usually be implemented with higher working frequencies, thus, increasing the system performance.

Interconnection network designers should reach all of these goals by manipulating several design parameters. In this chapter, we focus on the most important ones: topology, switching, routing and fault-tolerance. Nevertheless, there are a lot of other design parameters that are not explained in detail in the following sections because they are not directly related with our contributions, but could also have a great impact on the system performance. As aforementioned, if the reader desires more information about these issues we refer him to the established books on this topic [33, 40].

2.1.3 Topology

Topology refers to the static connection of the nodes to the switches, and the connection among the switches. That is, topology defines which nodes are connected with which switches and the connections among the switches, defining the “roads” that packets can follow across the interconnection network. Several examples of topologies are represented in Figure 2.3. Notice that each of the lines that connect the nodes in the figure represent two channels, one in each direction.



(a) Bidirectional ring. (b) Totally connected. (c) Irregular topology.

Figure 2.3: Example of different topologies to interconnect four nodes.

Nevertheless, topology does not only define the shape, or the road-map, of the network, it also sets the width of the channels, their length, and other physical parameters concerning links [40]. In this way, topology defines the maximum bandwidth of the links, their latency, and the bisection bandwidth of the network. Therefore, it has a high impact over the interconnection network performance. Topology can be considered the most important design parameter since not only performance is highly dependent on it, but most of the other design decisions of an interconnection usually also depend on the chosen topology, i.e., the routing algorithm depends on the topology of the network in most of the cases.

Topologies have been traditionally studied using graph-like representations, such as the ones presented in Figure 2.3. This is a very simplistic representation because it does not consider implementation details that may affect to the topology. Nevertheless, the graph representation of a network helps to study several interesting properties of the interconnection network:

- Switch degree: The number of channels that connect a switch with its neighbors[1]. For example, in the topology presented at Figure 2.3(a), the degree of all the switches is two; in Figure 2.3(b), the degree of the switches is three; and in Figure 2.3(c), we can see switches with degree one, two and three. In case that the number of input channels and output channels in a switch does not coincide, switch degree is defined as the maximum value of input switch degree and output switch degree. Input

[1]Two switches are considered to be neighbors if they are directly connected by a channel.

switch degree is the number of input channels that connect its neighbor switches to a given switch. Output switch degree is defined in a similar way but considering output channels. Switch degree is an important factor when trying to determine the complexity and cost of the network, since switches with high degrees are more expensive that switches with a lower degree. Nevertheless, by using high-degree switches, the number of switches in the network is reduced. So, designers of networks have to reach a trade-off between the number of switches in the network and their degree.

- Diameter: The maximum distance between two nodes in the network. For instance, the diameter of the network represented in Figure 2.3(a) is two; the diameter of the one shown at Figure 2.3(b) is one; and the diameter of the network presented in Figure 2.3(c) is two. Usually, as the diameter of a network is increased, the latency of the network is also increased and, at the same time, the throughput of the network is also reduced. For this, low-diameter networks are commonly the preferred choice when designing a network.

- Average distance: The average distance between any pair of nodes in the network. In other words, the average number of hops for a packet that traverses the network from a source node to a destination node. Average distance is a more realistic approach to calculate the packet latency in a network than its diameter, since the diameter of a network represents the worst possible case which may only eventually appear.

- Regularity: a network is regular when all the switches have the same degree. From the topologies presented in Figure 2.3 only the topology shown in Figure 2.3(c) is not a regular topology. Usually, regular topologies are preferred over irregular topologies, because in regular networks the performance of the network can be more easily predicted in the design phase, and easier to implement by using commodity elements.

- Symmetry: a network is symmetric when it looks alike from every node. From the topologies presented in Figure 2.3 only the topology shown in Figure 2.3(c) is not a symmetric one. Nevertheless, the reader should not associate the regularity of a network and its symmetry, since there are topologies that are regular and are not symmetric, and vice versa.

- Bisection bandwidth: If the network is segmented into two equal parts, it represents the bandwidth between the two halves [65]. Bisection bandwidth is an important parameter to estimate the throughput of a network, since the throughput of the network is upper bounded by a factor of this parameter. Commonly, topologies with a high bisection bandwidth are preferred over topologies with a low bisection bandwidth.

Topology Types

Topologies are usually classified as direct or indirect. Direct networks are the networks were all the switches of the network are directly connected to a processing node [40]. Usually, in direct networks, the switch is integrated inside the processing node [33], and they are represented like in Figures 2.3 and 2.4. This is the case of some processors in which the switch has been introduced inside the chip, like in the Alpha 21364 [93]. However, dedicated switches are commonly used for high-performance interconnection networks [40]. Figure 2.4 represents the most popular symmetric regular direct topologies: meshes and tori. These topologies are usually represented for a 2D or 3D implementations, but they can be defined for any number of dimensions. General n-dimensional meshes and tori can be built by replicating the $n-1$-dimensional design n times, and by properly connecting the switches among them. For example, a 3-dimensional mesh derived from the one shown at Figure 2.4(a) can be constructed by replicating 3 times that network, having three 2-dimensional meshes. In this meshes, we have to connect the switches that are located in the same position in both networks.

The topologies depicted in Figure 2.4 are mesh-like topologies. All the topologies that resemble a mesh have in common the interesting characteristic of having their links arranged in several orthogonal dimensions in a regular way. In fact, these topologies are particular instances of a larger class of direct network topologies known as k-ary n-cubes, where k is the number of nodes interconnected per dimension, and n is the number of dimensions of the network. The symmetry and regularity of these topologies simplify

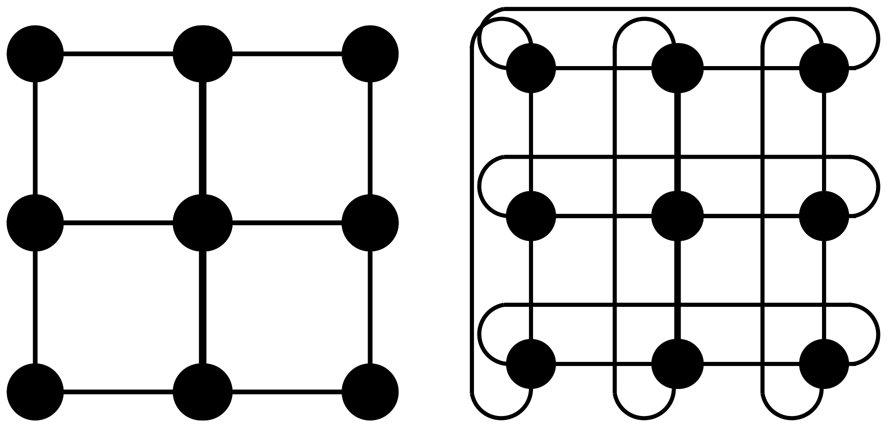
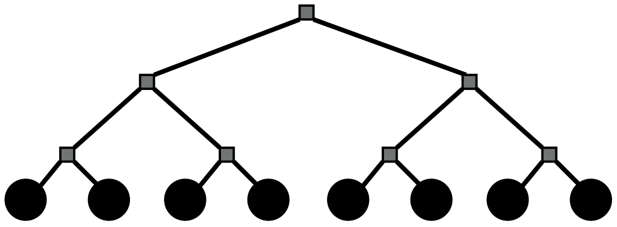

(a) Mesh topology. (b) Torus topology.

Figure 2.4: Two examples of direct networks.

network implementation (i.e, packaging and cabling) and packet routing, as the movement of a packet along a dimension does not modify the number of remaining hops in any other dimension toward its destination. For example, in the network shown in Figure 2.4(a), if a packet has to travel from the upper right node to the lower left node, the packet would have to make two hops in the vertical dimension, and two more hops in the horizontal one. If the packet is forwarded first through the vertical dimension, moving down in the figure, it would have to make one more hop in that dimension, but the number of hops in the other dimension remains the same. The same happens if the packet travels along the other dimension.

In direct networks, as the number of nodes in the system increases, the total bandwidth and processing capabilities of the computer also increase [40]. Thus, direct networks have been a popular interconnection architecture for constructing large-scale parallel computers. Nevertheless, as network size increases direct networks have a serious drawback: in order to avoid being a bottleneck for the system performance, they must be implemented using a very high number of dimensions, increasing the complexity of the switches and the wiring. Another possibility is to use a low-dimensional topology with a large number of nodes per dimension , but in this case the bandwidth per node is strongly reduced.


(a) Tree topology.

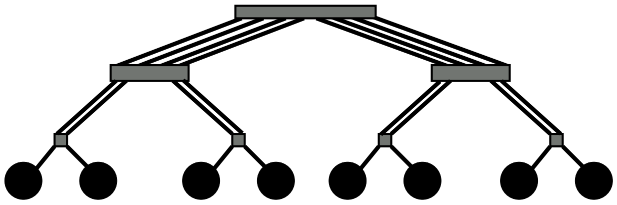
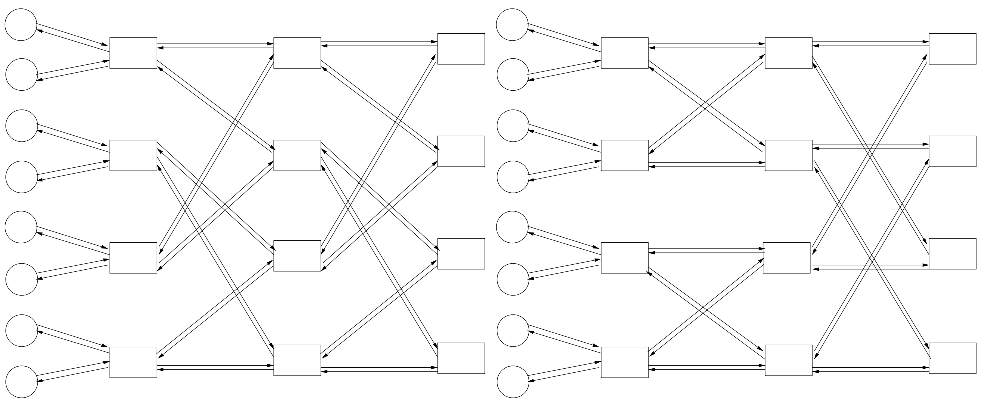

(b) Fat-tree topology.

Figure 2.5: Two examples of indirect networks.

The solution to this fact was the introduction of indirect topologies in high-performance computers. The main difference between direct and indirect topologies is that in indirect topologies there are some switches, at least one, that do not have any processing node directly connected to it. Furthermore, some ports in switches from indirect topologies can be left unused [40]. Figure 2.5 shows two examples of indirect networks. On the contrary to direct networks, indirect networks are always implemented by using dedicated switches, as it can be seen in Figure 2.5 where switches are represented as grey boxes and nodes are represented as black circles. Notice that, in both networks, the upmost switch has half of its ports unused. These switches are referred to as root switches in tree-like topologies.

In indirect networks, thanks to the switches that do not have any node connected to them, the bandwidth per node scales with the number of nodes in the system. If we want to double the number of nodes in the network in any of the networks from Figure 2.5, we have to replicate the network and add another layer of switches to connect both parts of the network. In this way, the bandwidth per node remains constant. Nevertheless, notice that the number of switches per node is usually higher in indirect networks than in direct networks, increasing the network cost per node.



(a) Omega topology. (b) k-ary n-tree topology.

Figure 2.6: Two examples of multistage interconnection networks.

This dissertation is focused on a subset of symmetric regular indirect networks known as multistage interconnection networks (MINs). The main characteristic of multistage interconnection networks is that the switches of the network are distributed in stages, the processing elements are only connected to the lowest stage of the network, and the switches are only connected to some of the switches from the previous stage and to some of the switches from the following stage following a prefixed pattern. The resulting topology depends on the chosen pattern. Two different MINs are presented in Figure 2.6, both connecting eight nodes with twelve switches, but each one following a different connection pattern. Notice that the network depicted in Figure 2.5(b) is not a MIN since it is not a regular topology.

In MINs, switch ports are divided into up and down ports. The up ports of a switch are the ports that connect a switch with switches that belong to the next stage. On the contrary, down ports are the ones that connect a switch with switches that belong to the previous stage. Usually, the stages are numbered beginning from the processing nodes. So, the switches that are directly connected to the processing nodes are considered to belong to the stage number zero.

MINs are a parametric family of interconnection networks, as they are defined by two parameters: k and n. k is the number of up or down ports of every switch in the network. That is, the switch degree is $2 \times k$ since all the switches have the same number of up and down ports. n is defined as the number of stages of the network. In this way, a MIN can connect up to $N = k^n$ nodes, being N the number of processing nodes of the system, by using nk^{n-1} switches and nk^n bidirectional links. This can be checked with the topologies from Figure 2.6, where n is three and k is two.

Fat-Tree Topology

Among all the indirect interconnection networks, this dissertation is focused on the fat-tree topology [76], presented in Figure 2.5(b). If we compare the fat-tree with the complete tree topology presented in Figure 2.5(a), it can be clearly seen that both topologies are very similar. Fat-tree was derived from the tree topology to solve the bandwidth bottleneck problem of the tree topology near its root. As it can be seen in Figure 2.5(a), in the tree topology the links that connect the root switch with its neighbor switches can easily become a bottleneck if more than one switch from one half of the network need to communicate with some of the nodes from the other half of the network.

The difference between fat-tree and tree topology is that fat-trees get thicker near the root of the tree by increasing the number of up links in each stage. In this way, in the fat-tree topology the accumulated bandwidth of all the channels that connect each stage with its following stage is the same at all the stages. On the contrary to the tree topology, in the fat-tree topology there is not a bandwidth bottleneck near the root of the tree. Furthermore, in the fat-tree topology, all the switches from one half of the network can communicate with all the switches from the other half of the network at the same time.

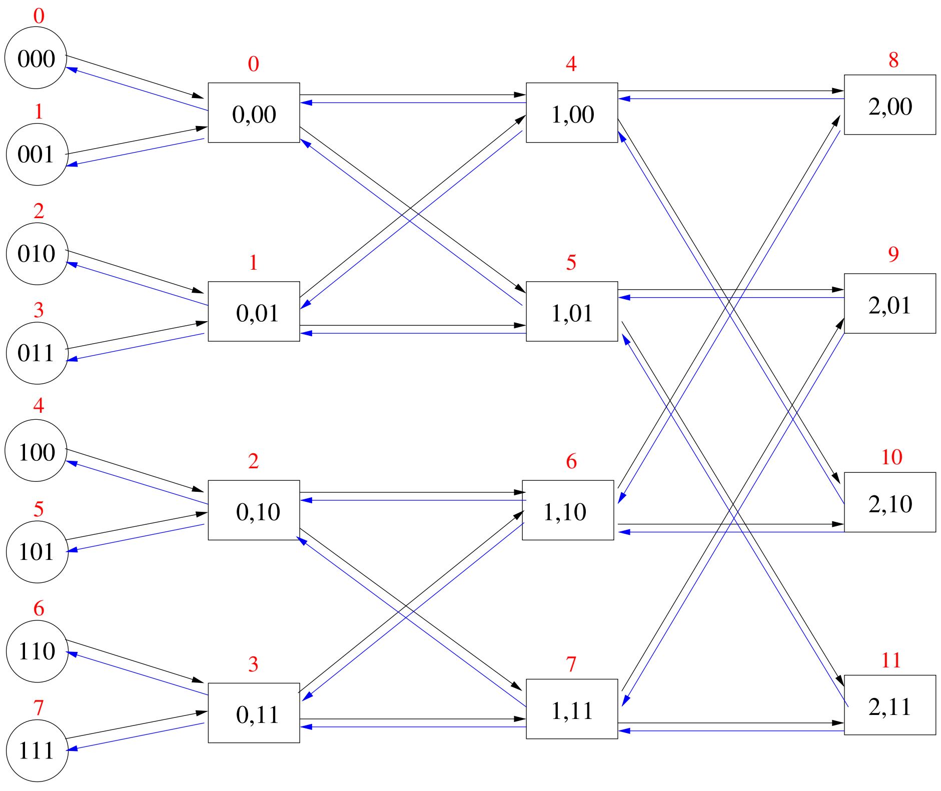
Nevertheless, this implementation of the fat-tree topology is not physically feasible. As the network size increases, the number of stages in the network is also increased, and switch degree is doubled at each stage. Considering that the maximum number of ports in a switch is limited by technological constraints such as the pin out of the switches, the depicted implementation of the fat-tree becomes nearly impossible for current clusters. For this reason, we focus on an equivalent topology to the fat-tree that only uses switches of a fixed degree: the k-ary n-tree topology [101]. k-ary n-trees are a sub-family of the MINs. Remember that, k represents the number of up links of the switches of the network, and n represents the number of stages of the network. A 2-ary 3-tree is represented in the Figure 2.6(b), being k equal to two and n equal to three. We focus on this topology since it has risen in popularity in the last few years, as it has become the suggested or the default topology for most of the commodity high-performance interconnects vendors (Myrinet [8], Quadrics [9], and Infiniband [7]).

In this dissertation, we make use of a specific nomenclature and a numbering for the nodes, switches, and links of the k-ary n-tree topology. An example of a 2-ary 3-tree can be seen in Figure 2.7. On one hand, the numbering for nodes and switches is very simple; both are numbered beginning from zero up to the number of nodes, or switches, minus one. This numbering corresponds to the red numbers that are above nodes and switches in Figure 2.7(a). In addition to this identifier, nodes and switches are labeled in a more formal way. Each processing node is represented as a n-tuple $\{0, 1, ..., k-1\}^n$, that is, nodes are labeled with n components whose value vary between 0 and $k-1$. This label is represented with black numbers inside the nodes in Figure 2.7(a). Switches are defined as a pair $\langle s, o\rangle$, where s is the stage the switch is located at, $s \in \{0..n\text{-}1\}$, and o is a $(n-1)$-tuple $\{0, 1, ..., k-1\}^{n-1}$ which identifies the switch inside the stage. This second component of the switch identifier is the same than the one received by the nodes, but with one less component. This formal labeling for switches can be observed inside the switches of the Figure 2.7(a).

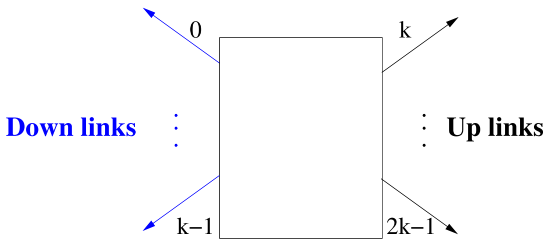
In addition, we will use the following link numbering in the switches of a k-ary n-tree: down links (highlighted in blue) are labeled from 0 to $k-1$, and up links from k to $2k-1$ (see Figure 2.7). So, down link labels are always lower than k, and up links labels are always equal or higher than k.

By using this labeling, we can easily establish if two switches are connected by performing a simple comparison. Formally, two switches $\langle s, o_{n-2}, ..., o_1, o_0\rangle$ and $\langle s', o'_{n-2}, ..., o'_1, o'_0\rangle$ are connected by a link, if $s' = s + 1$ and $o_i = o'_i$ for all $i \neq s$. In other words, two switches are directly connected by a link if they belong to consecutive stages of the network, and the components of the second element of their labels are equal but the s^{th}, this is the component corresponding to the stage where the lower switch is located at.

On the other hand, we can establish a similar relation between processing



(a) Node and switch nomenclature.



(b) Link numbering.

Figure 2.7: Used nomenclature and numbering for nodes, switches and links.

nodes and switches from the first stage of the network. Formally, there is a link between the switch $\langle 0, o_{n-2}, ..., o_1, o_0 \rangle$ and the processing node $p_{n-1}, ..., p_1, p_0$ if $o_i = p_{i+1}$ for all $i \in \{n-2, ..., 1, 0\}$. That is, a processing node and a switch are connected, if the switch belongs to the lowest stage of the network, and all the components of the second element of its label are equal to the components of the processing node identifier discarding the least significant one.

2.1.4 Switching Techniques

The next main design parameter of an interconnection network is the switching technique. Switching techniques define the timing and conditions required to allocate the resources of the network for the packets that cross it. These resources may include the crossbar connection, switch buffer ports, flow-control logic, and so on.

Switching techniques impose several aspects in the switch and network interface cards that may have a high influence over the performance, fabrication cost and power consumption of the elements of the network. As an example, switch port minimum buffer size is fixed by the switching technique. As it will be seen, packet switching forces the buffer size to be a multiple of the packet size, whereas wormhole allows to use buffer whose size is smaller than the packet size. Furthermore, switching techniques also fix the range of possible buffer sizes for the switch ports. In this way, switching technique has a high impact on network cost and power consumption [88]. In addition, the switching technique may force the delay before the routing and arbitration operations when a packet reaches a switch. A switching technique that imposes a long delay before the routing operation after the packet arrival to a switch should obtain a low throughput and high average packet latency.

Switching techniques can be mainly distinguished by the relative timing of the resource allocation operations. For the purpose of comparison, we consider the computation of the base latency of an L-bit packet in an empty network for each switching technique. Channels can transfer up to W bits per clock cycle. The packet header is assumed to be of size W bits, being the total packet size $L+W$. Switches can route packets in t_r clock cycles, and channels operate at a frequency of B Hz, which gives a channel delay of $t_w = 1/B$. Channel bandwidth is BW bits per second. Once, a path has been set up in a switch, the time that a packet needs to traverse a switch is t_s. Finally, we assume that the distance between source and destination nodes is d hops.

Circuit Switching

In circuit switching, the network establishes a reserved path between source and destination nodes prior to the transmission of the packet. This is performed by injecting in the network the header of the packet, which contains the packet destination. The header of the packet acts as some kind of routing probe that progress toward the destination node reserving the channels that it uses. When the probe reaches its destination, a complete path between destination and source nodes has been set up, and a acknowledgment is sent to the source node to start with the transmission of the packet. As the path has been reserved for this packet, it can cross the network avoiding colliding with other packets. The circuit reservation is undone when the packet completely crosses each intermediate switch. Circuit switching has been employed in several commercial products, like the Intel iPSC/2 processor [97].

Circuit switching can be very advantageous when packets are very infrequent and long. Nevertheless, this switching technique has several important drawbacks. If circuit set up time is long compared to transmission time of the packets, it will strongly penalize the performance of the network since channels will be underused. Additionally, as channels are reserved for a given packet, no other packets can use them despite that the packet that reserved the channel may not be using it, thus channels may become even more underused.

Using the proposed values to compare base latency of packets, the base packet latency of circuit switching is:

$$\begin{aligned}
t_{cs} &= t_{setup} + t_{data} \qquad (2.1)\\
t_{setup} &= d[t_r + 2(t_s + t_w)]\\
t_{data} &= \left\lceil \frac{L}{W} \right\rceil t_w
\end{aligned}$$

Store and Forward

This switching technique is also known as packet switching. On the contrary to circuit switching, there is not path set up phase on this switching technique. Packet data is transmitted immediately after packet header transmission. When the packet arrives to a switch, the switch waits to store the whole packet in its input port buffer to read the destination of the packet and perform the routing. So, input port buffers must be at least of size equal to the maximum possible packet size, in order to be able to store the whole packet. Opposite to circuit switching, base latency of packets is only composed by the transmission time of the packet through the network, and it can be computed as follows:

$$t_{ps} = d\left\{ t_r + (t_s + t_w)\left\lceil \frac{L+W}{W} \right\rceil \right\} \tag{2.2}$$

As it can be seen, latency is dominated only by the packet size and the distance between the nodes, which is a high reduction compared to circuit switching. However, notice that store and forward relies on using buffers at the switches, increasing the cost and complexity of switches compared to the ones from circuit switching.

Virtual Cut-Trough Switching

Packet switching is based on completely receiving a packet before any routing decision can be made. But, this is a very conservative, since packet header contains all the required information to perform the routing at the switches, and it is physically located at the beginning of the packet. So, packet routing process can be started as soon as the packet header has arrived to a switch, without waiting for the rest of the packet. This is what is done in virtual cut-through (VCT) switching. Base latency is as follows:

$$t_{VCT} = d(t_r + t_s + t_w) + max(t_s, t_w)\left\lceil \frac{L}{W} \right\rceil \tag{2.3}$$

In this case, the impact of the packet size is smaller than in packet switching, and the base latency for this switching technique is more influenced by the distance between the nodes. Despite this, buffer requirements are the same for VCT and packet switching because, in VCT, when a packet cannot advance through the network it has to be completely stored in the buffers of the switches. This is the switching technique commonly-used in off-chip high-performance interconnects [33, 40].

Wormhole Switching

The requirement to completely store a packet in the buffer of a switch makes difficult to design a small, compact, and fast switch [40, 88]. For this reason, in wormhole switching, packets are divided in smaller parts of fixed size called flits. Buffers at the ports of a switch only have to provide enough space to store one flit, instead of the whole packet. Thus, buffer requirements are strongly reduced making possible to introduce high-performance networks in heavily constrained environments like networks on-chip [88]. In wormhole, when a packet cannot advance through the network, it is stopped and it does not release the buffers and channels that the packet is using. The stopped packets can be allocated in several switches and they resemble worms that are trying to traverse the network. Wormhole packet base latency is the same than VCT base latency, since the buffer size difference between both techniques only influences when the number of packets in the network is enough to provoke collisions among them [40].

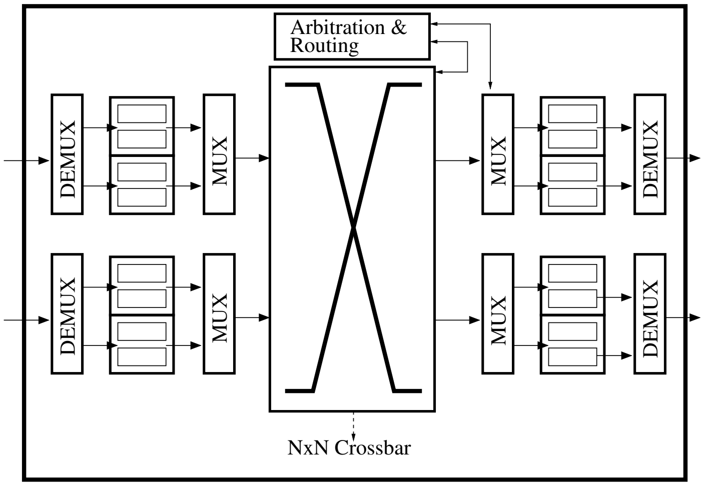
2.1.5 Virtual Channels

Virtual channels [32] where introduced in the interconnection networks for throughput improvement purposes. In [42], the authors showed that a typical switch can only achieve an efficiency of 58.6% due to the effects of the Head of Line blocking (HOL) effect. HOL is produced in the input buffers of the switches when a packet in the head of the buffer is stopped for any reason, but at least one packet in that queue could be forwarded through the crossbar, since the output port it demands is free. However, this second packet cannot be forwarded since the packet on the head of the queue is blocking its advance. In a switch with virtual channels, there are several input and output parallel buffers per each port like in Figure 2.8[2]. For example, in Figure 2.8(a), each input port has two virtual channels, since the buffers of each physical port have been duplicated. Each input and its corresponding output buffer act as a virtual channel or port. Virtual channels are treated like physical channels. That is, arbitration logic has to be extended to arbitrate among the different virtual channels, and packets can only change from one virtual channel to another one in the crossbar. By using virtual channels, as we have several virtual networks, when a packet at the header of the queue is blocked, it only affects to the packets of that virtual channel. Despite the benefits of virtual channels, notice that they increase the switch complexity and cost.

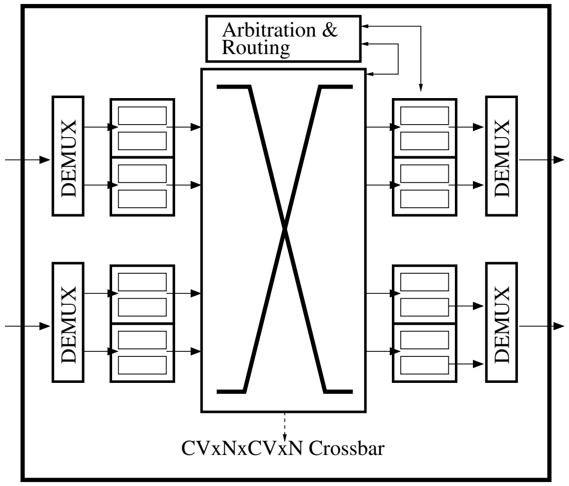
Concerning buffer requirements, virtual channels can be implemented in two ways. The first one is a very conservative scheme that departs from a switch without virtual channels, and just split the buffers of the switch ports to form the virtual channels. In this way, the buffer requirements at the ports of the switch are not increased, since the total buffer capacity per port remains the same. The second one, is mire aggressive; instead of splitting the buffers at the switch ports, it fully replicates them. This approach obtains better results than the previous one, but the buffer requirements per switch port are multiplied by the number of virtual channels.

Despite the fact that virtual channels may be implemented without increasing the buffer requirements of the switch by sacrificing network performance, the use of virtual channels involves introducing several extra logic, as it can be seen by comparing the Figure 2.2, which represents a typical switch without virtual channels, and Figures 2.8(a) and 2.8(b), which show two different ways to implement virtual channels in a switch. First, we focus on the multiplexed crossbar implementation shown in Figure 2.8(a). As it can be seen, virtual channels require a demultiplexer at their input ports to virtually split the single physical channel into several virtual channels, two in this case. When a packet reaches the input port, the switch logic stores it in the input buffer corresponding to the virtual channel indicated in the packet header. In this implementation, crossbar size is not increased, comparing it with the one from

[2]Notice that in the figure only one direction is represented for the sake of clarity, but the number of multiplexers, demultiplexers and virtual channel connections inside the switch is twice the one shown in the figures.



(a) Multiplexed crossbar.



(b) Non-multiplexed crossbar.

Figure 2.8: Two types of virtual channels switches.

a switch without virtual channels. Therefore, virtual channels have to be multiplexed to enter the crossbar, and demultiplexed when they have crossed it. Notice that arbitration becomes a two-step process, since it has to decide to which of the virtual channels per each port has to be granted the access to the crossbar, and after that it has to perform the traditional channel arbitration. Arbitration phase is heavily increased since finding an optimal resource matching in a non-multiplexed crossbar is not a trivial problem [37], and it can decrease network throughput, also increasing packet latency [38]. Finally, the packet is stored in the output port, and the switch has to perform an additional arbitration process to decide which of the virtual channels can use the output channel.

In the other hand, virtual channels can be implemented by using non-multiplexed crossbars like the one shown in Figure 2.8(b). In this implementation, there are not multiplexers and demultiplexers between the virtual channels and the crossbar, which considerably simplifies the arbitration process. Nevertheless, notice that crossbar size has been multiplied by the square of the number of virtual channels, since a $N \times N$ crossbar has become a $(CV \times N) \times (CV \times N)$ crossbar, being N the number of ports of the switch, and CV the number of virtual channels. Furthermore, this increase in the crossbar size implies that arbitration phase becomes more complex and slower since it has a higher number of resources to match.

To summarize, virtual channels has shown that they are a good approach to improve switch effective throughput, but no matter how they are implemented they have a significative impact on switch complexity and cost, in terms of logic and buffer requirements.

2.1.6 Routing Technique

Topology defines all the paths that a packet can take along the network. Nevertheless, there must be a mechanism that defines which route from all these possible paths is going to be used by a packet when it tries to cross the network from its origin to its destination. This is the role of the routing algorithm in an interconnection network. The routing algorithm is the responsible of deciding the path across the network for all the packets. The routing algorithm has a great impact over the system performance. For instance, if the routing algorithm tends to use always the same channels, these channels will be overused, creating a hot spot of traffic, while the rest of the channels of the network are underused. In this way, the routing algorithm can become a bottleneck for the system performance. So, from the point of view of the performance, the routing algorithm should try to keep balanced the utilization of the links of the network.

Another important aspect of routing algorithms shows up when we consider the fact that packets may collide in the switches. It is very common that two or more packets located at different input ports of the same switch, request to use the same output port. In this situation, the port can only be granted to one of the requesting packets. The chosen packet will be forwarded through the output port, while the other requesting packets remain stored in the buffers of the input port without progressing through the network. When the packet that got the output port is completely transmitted, the port is granted to another of the requesting packets, and in this way, all the packets from the network can reach their destinations even in the case that they collide with other packets in the switches.

Nevertheless, the packet that got the output port may not be able to complete its transmission if it is involved in a deadlock. Deadlocks may arise when a set of packets cannot advance through the network because the resources that they need are being used by another packet from this set. In this way, packets keep waiting in a cyclic-like way to the other packets to release the resources that they need to progress through the network. This can be better understood with the example presented in Figure 2.9. In the figure, we can see a typical example of deadlock. The deadlocked packets are highlighted in blue. These packets are stopped since the space they require in the next switch of their route to store them is not available. The next switch for each packet is indicated with a blue arrow. As it can be seen, packet labeled as 1 is waiting to get space in the switch where packet labeled as 2 is stored. Indeed, packet 1 is waiting that packet 2 leaves the switch to go on with its travel through the network. In a more formal way, we say that packet 1 depends on packet 2. In the same way, packet 2 depends on packet 3, packet 3 depends on packet 4, and finally packet 4 depends on packet 1. None of the deadlocked packets will be able to go on traversing the network, since they all are waiting

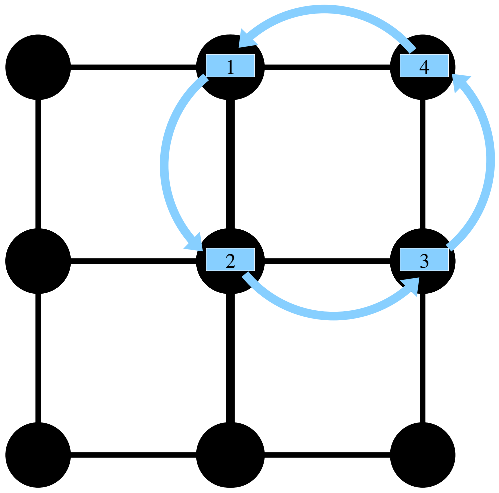


Figure 2.9: Example of a deadlock of 4 packets in a mesh network.

in a cyclic dependency.

In order to avoid deadlocks, we have to consider the channel dependency graph (CDG) of the routing algorithm. In the CDG, the channels are the nodes of the graph, and the edges represent the dependencies among the switches. That is, in the CDG, we will draw an edge between two nodes, if the routing algorithm allows a packet at the first node arrives to the second one. In this graph, deadlocks can be easily identified as loops. If there is a loop in the CDG, then packets may deadlock. More formal and extensive information about CDG and avoiding deadlocks in routing algorithms is available in [39,40].

Routing Algorithms Taxonomy

There are several criteria to classify routing algorithms, so we only explain the most representative ones for the work presented in this dissertation.

In first place, routing algorithms can be classified according to the minimality of the routes that they provide. A routing algorithm is classified as profitable, or minimal, if it only provides the shortest routes between the source and destination routes for all the packets. In the other hand, if the routing algorithm enables using larger routes for packets, we say that it allows misrouting packets, and the routing algorithm becomes non-minimal. For instance, consider the case shown in Figure 2.10, in which the node in dark

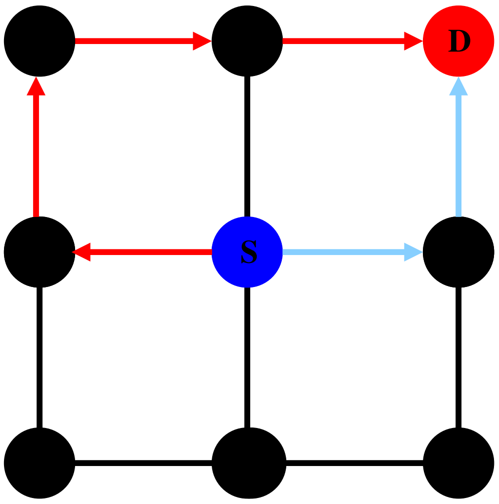


Figure 2.10: Difference between non-minimal path (in red) and minimal path (in light blue) between two nodes in a mesh.

blue requires to send a packet to the node in red. One of the minimal routes between both nodes is highlighted in light blue. A minimal routing algorithm would only provide that route or an equivalent route of the same length between the source and destination nodes. On the contrary, if the routing algorithm allows misrouting a packet through a longer route, such as the one highlighted in red, the algorithm will be classified as non-minimal. Usually, minimal algorithms are the preferred choice since they provide shorter routes than non-minimal algorithm, thus lowering the average latency of the packets. Despite of this fact, there are several scenarios where non-minimal algorithms are very useful. That is the case of several congestion management techniques, and fault-tolerant routing algorithms.

Notice that with the use of misrouting, packets may enter in a livelock. Livelocks are very similar to deadlocks, since both imply that packets never reach their destination after being injected in the network. Nevertheless, on the contrary to deadlocks, in a livelock, packets are never stopped, they keep moving through the network indefinitely without reaching the destination node. Livelocks should also be avoided when considering to design a non-minimal routing algorithm, as well as deadlocks.

In addition, routing algorithms can be also classified taking into account where the routing decisions are taken. If the routing decisions for all the packets in the network are taken in a single place, the routing algorithm is centralized. This central element that route the packets of the interconnection network may be a dedicated computer, or a piece of software in any of the computing nodes of the cluster. On the other hand, if the routing decisions for each of the packets are taken in their source nodes, it is said that the routing algorithm uses source routing. In source routing, the source node computes the path prior to packet injection and stores it in the packet header, which may noticeably increase the packet header length. Source routing has been used in some networks because routers are very simple [8, 15]. In this routing scheme, the switches of the network just read the stored route in the packet header and forward the packet through the indicated output port, which makes routing very quickly and switches very simple. Nevertheless, storing the whole route in the packet header does not scale, since packet header size grows with network size. For this, source routing is not usually used in high-scale supercomputers. Finally, if the routing decisions are taken in the switches, the routing algorithm becomes distributed. In distributed routing, source nodes store the identifier of the destination node into the packet header instead of storing the route for the packet, which makes shorter the packet header. In distributed routing, the switches use the identifier of the destination from the packet header to dynamically construct the route for each packet by computing the next link that will be used by the packet while the packet travels across the network. Distributed routing has been used in most high-performance routers for efficiency reasons, since it allows more flexibility, as it allows to exploit the fact that, at each switch, different ports can be available to reach a given destination (distributed adaptive routing). Notice that with distributed routing, switches require additional logic in order to take the routing decisions, thus increasing the complexity of the switches.

Centralized routing was mainly used on the first interconnection networks. Nowadays, is considered to be outdated, since centralized routing does not scale with the network size. As the number of elements of the network increases, the central element that performs the routing decision becomes a bottleneck limiting the performance of the network. This is the case of the Cell BE processor [16]. Distributed routing has become the common case for high-

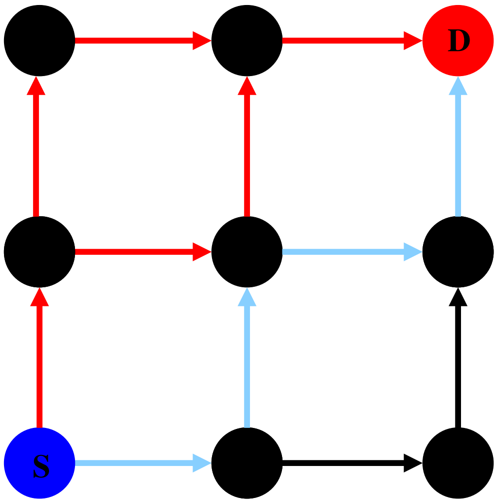


Figure 2.11: Difference between deterministic routing (in light blue), partially adaptive routing (in red), and fully adaptive.

performance routing [7, 9], since it is more easily to assemble a cluster with commodity components and provide adaptive routing. Nevertheless, source routing is used in several high-performance networks where the complexity of the switch is a key point [1, 8, 31].

The last way of classifying the routing algorithms is the most important for the work presented in this dissertation. Routing algorithms may also be classified depending in the number of alternative routes that they provide to communicate two nodes of the network. A routing algorithm that only provides a single prefixed route for each source-destination pair is a deterministic routing algorithm. For instance, in Figure 2.11, if all the packets sent from the node S to the node D always follow the route highlighted in light blue, the network is using deterministic routing. Packets between those two nodes cannot deviate from this route, no matter if the route is being occupied by other packet and all the other routes in the network are free. As it can be seen, with deterministic routing the traffic of the network may make an unbalanced use of the network links depending on the traffic pattern, thus limiting the performance of the network.

For this reason, it was proposed to adapt the route of the packets to the traffic conditions of the network, providing adaptive routing. Commonly, adaptive routing is implemented in conjunction with distributed routing schemes, since adaptive routing does not easily fit in centralized and source routing schemes. In a centralized adaptive routing scheme, the central routing element would require to gather instantly information from the entire network in order to optimally adapt the route for all the packets in the network, which limits even more the scalability of this kind of schemes. On the other hand, adaptive source routing is also quite limited since the only decision is taken in the injection of the packet, and the conditions of the network may change noticeably while the packet travels along the network.

In adaptive routing, there are several routes between the source and destination nodes. The decision of which route is going to be used for each packet from all the possible ones is taken at each switch by the selection function. The routing algorithm returns several useful output ports that can be used in the current switch and the selection function decides which of the ports returned by the routing algorithm will be finally used. This decision is based on local information to the switch, using this local information as some kind of heuristic to know the traffic status of the interconnection network. For instance, the routing algorithm returns that the packet can be forwarded through two different output ports; at this point the selection function looks to the buffers of the candidate output ports, and chooses the one whose buffer has more free space. This selection function chooses the output port whose buffer has more free space, since it considers that this output port is less used than the other one, so the packet would be able to arrive sooner to its destination than using the other output port. This is only an example of selection function, as the selection functions may use very different criteria to make the final decision, as it can be seen in [50].

Adaptive routing algorithms may be subdivided into two groups depending on whether they allow using all the possible routes between source and destination nodes or not. If the routing algorithm only allows using a set from all the possible routes from source and destination nodes, the routing algorithm becomes partially adaptive. On the contrary, if the routing algorithm allows using all the routes provided by the topology from source and destination nodes, the routing algorithm is a fully adaptive routing algorithm. Figure 2.11 shows an example of partially adaptive routing in red. As it can be seen, the routing algorithm provides two different routes from node S to node D, but they are only a subset from all the possible routes between these nodes. A fully adaptive routing would provide all the routes from the network indicated by an arrow no matter its color. That is, a fully adaptive routing algorithm will allow using the blue, red and black routes.

As commented, adaptive routing usually provides a higher performance than deterministic routing due to its capacity to adapt the routes of the packets to the conditions of the network. Nevertheless, adaptive routing requires a selection function at the switches, increasing their complexity. In deterministic routing, a selection function is not required, since there is only one route for all the packets. Furthermore, there are several applications that require in-order delivery of the packets [84]. Infiniband technology also requires in-order delivery of packets, dropping all the packets that arrive out-of-order to their destination [7]. In deterministic routing, in-order delivery of the packets is provided by default. As all the packets from a given node to another node always follow the same route, there is no way that a packet can overtake a previous packet. Whereas, in adaptive routing, as packets between two nodes may follow different routes, the order of the packets may not be preserved since the traffic conditions of the different routes may be different delaying more some packets than others. In order to preserve in-order delivery of packets with adaptive routing, the designer should apply additional techniques, which increase even more the complexity of the network when we compare them with deterministic routing, since these techniques are based on having large buffers at the source and destination NICs that perform a reordering of the packets that do not arrive in order [86], or on performing this reordering at the switches by means of virtual channels and/or operations to move packets in the buffers of the switches [23].

Routing in Fat-Trees

Fat-tree topology has a great diversity of routes between every pair of nodes, as it can be seen in Figure 2.12. The figure shows all the possible paths from node 1 to node 6 highlighted in red. As it can be seen, between these two nodes there are four possible different, but equivalent, routes. This great diversity of routes makes the fat-tree an excellent topology for adaptive routing. For

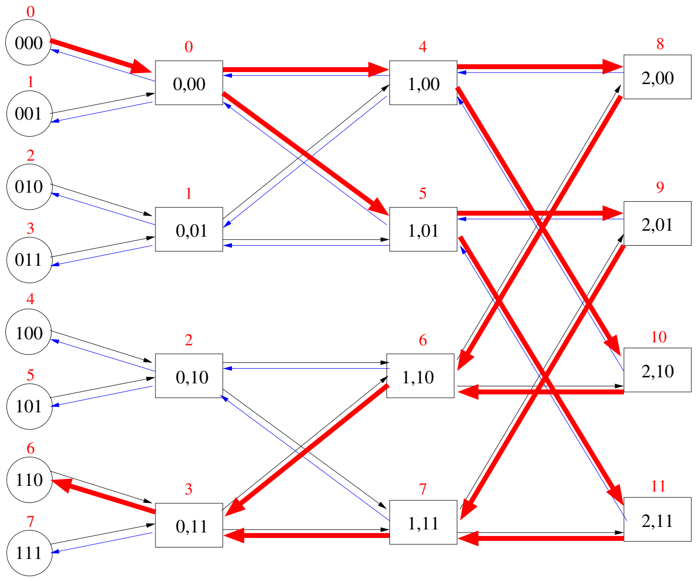


Figure 2.12: All the possible routes for a packet from node 1 to node 6 in a 2-ary 3-tree.

this reason, we focus on minimal adaptive routing in fat-trees.

In order to perform minimal routing in fat-trees, we have to identify the switches that are the nearest common ancestor of the packet source and destination nodes. A common ancestor of two nodes is a switch that is able to reach both nodes by utilizing only down links. For example, in the network presented in 2.12, switch 7 and 11 are common ancestors of nodes 5 and 7, but switches 2 and 5 are not common ancestors of these nodes since they cannot reach both nodes only using down links. From the set of common ancestors of two nodes, the nearest common ancestors are the ones that belong to the lower possible stage. For example, in the same figure, the nearest common ancestor of nodes 5 and 7 are the switches 6 and 7, whereas the nearest common ancestors of nodes 1 and 6 are the switches 8, 9, 10, and 11.

Minimal adaptive routing in fat-trees is performed in two phases. The first phase is the ascending phase of the routing algorithm, since the packet travels from one of the processing nodes located at the leaves of the tree upwards to the root of the tree. This phase is fully adaptive, that is, at each switch reached in this phase, any of its up ports can be used. Ascending phase ends when the packet reaches one of the nearest common ancestors between the source and destination nodes of the packet. As this phase is fully adaptive, it is enough to identify only the stage at where the nearest common ancestor switches are located. There is no need of identifying them individually, since all of them are located at the same stage and any of them can be reached during the ascending phase of the routing algorithm.

In a formal fashion, the stage at which the packet must be forwarded up to is obtained by comparing the source and destination identifiers components beginning from the $(n-1)^{th}$, the most significant one. The first pair of components that differs indicates the last stage to forward up the packet. For instance, in order to send a packet from the node $p_{n-1}, ..., p_1, p_0$ to the node $p'_{n-1}, ..., p'_1, p'_0$, the packet must be sent up to the stage i, if $p_j = p'_j$ for $j \in \{n-1..i+1\}$ and $p_i \neq p'_i$. For example, a packet traveling from node 4 ($\langle 100 \rangle$) to node 6 ($\langle 110 \rangle$) has to reach the second stage of the network, since this is the first different component between both identifiers beginning from the most significative one.

Once the packet has reached the stage at where the common ancestors are located, the second phase of the routing algorithm begins. In the second phase, there is only one possible path from any of the nearest common ancestors to the destination node. Hence, this second phase is deterministic. Once in the descending phase, at each stage, the descending link to choose is indicated by the component corresponding to that stage in the destination n-tuple. In the previous example, from stage i, the packet must be forwarded through link p'_i at stage i, at stage $i-1$ through link p'_{i-1}, and so on.

For instance, in Figure 2.12, a packet generated at node 0 whose destination is node 2 will be forwarded up to stage number one (through switch $\langle 0, 00 \rangle$ and choosing either path to $\langle 1, 00 \rangle$ or $\langle 1, 01 \rangle$). From any of these switches, the remaining bits (bits 1 and 0) of the destination identifier (10 in our example) correctly forwards the packet to node 2.

Notice that the deterministic path on this second phase completely depends on the reached switch during the ascending phase, therefore, the decisions made during the adaptive phase may be critical to balance the traffic in the network. However, despite the reached switch in the ascending phase, the combination of down ports that delivers the packet to its destination during the descending phase is always the same and is given by the destination n-tuple.

Routing Implementation Schemes

In this section, we focus on the implementation of distributed routing algorithms. As we intend to propose a fault-tolerant adaptive routing algorithm and a deterministic routing algorithm for fat-trees both using distributed routing, we analyze the advantages and disadvantages of the most common schemes to implement distributed routing algorithms.

The first presented routing implementation scheme is the logic-based one. In this scheme, the switches have a dedicated hardware that performs all the required operations to take the routing decision. Following with the example of the previous section, this hardware in a fat-tree using adaptive routing has to check whether the packet is in its first routing phase or in its second one; in case the packet is the first phase, it must check whether the packet has to begin the second phase or not. If the packet continues in its upwards phase, all the up output ports are returned. If the packet has to begin the downwards phase, the only link to return is the destination component corresponding to the current stage. This implementation scheme is extremely fast and efficient since it is fully implemented on hardware. Nevertheless, it lacks from flexibility, since hardwired logic cannot be changed to adapt to the changes in the network. Hence, this is not a good scheme for fault-tolerance.

Many of the large parallel cluster-based machines usually base their routing on forwarding tables. In the forwarding tables routing scheme, there is a table at each switch that stores, for each destination node, the output port that must be used. Despite that this scheme was only first used with deterministic routing, it can be extended to support adaptive routing by storing several outputs in each table entry [85]. The main advantage of table-based routing is that any topology and any routing algorithm can be used. So, it may seem a good scheme for fault-tolerance. However, routing based on forwarding tables suffers from a lack of scalability, as table size grows linearly with the network size. Moreover, the time required to access the table also depends on its size, so it is also increased with the system size.

Another alternative is the Interval Routing (IR) routing scheme [111]. IR is a scalable distributed routing scheme since its requirements do not grow linearly with network size. IR is based on grouping the destinations that are physically reachable from the same output port of a switch into an interval. Switches route packets through the output port whose interval contains the destination of the packet. To implement IR, it is sufficient to store the bounds of each interval and perform a parallel comparison. Despite that routing with intervals may seem very limited, hypercubes, n-dimensional meshes with deterministic routing, and multistage networks support IR [20]. Moreover, some generations of the Quadrics network [9] have used IR. In addition, in [62], IR was extended with the proposal of Flexible Interval Routing (FIR), a routing strategy for switch-based networks that allows to implement the most commonly-used deterministic and adaptive routing algorithms in meshes and tori.

2.1.7 Fault-Tolerance

Fault-tolerance is the capability of a system to keep working correctly in the presence of faults. When a fault appears in a non-fault-tolerant system, it may stop working at all or it may keep working but without ensuring that the results of its operations are correct. In a fault-tolerant system, we can ensure both properties under the presence of faults: the system keeps working and its results are not corrupted by the fault.

In fault-tolerance, the most important terms are reliability, availability, and dependability [104]. Reliability refers to the ability of the system to operate continuously without failing. Reliability, in the most relaxed form, is defined by the exponential distribution, which assumes that the probability of faults is random, and they affect to random elements of the system:

$$R(t) = e^{-\lambda t} \tag{2.4}$$

λ represents the failure rate of the system expressed as the percentage of faults per time (usually expressed as faults per one thousand hours or as faults per hour). The Mean Time Between Faults (MTBF) is derived from the previous equation, as the average time a system will run between two consecutive faults. MTBF is usually expressed in hours and it is defined by

Table 2.1: MTBF of several large-scale high-performance computers.

Computer	#CPUs	MTBF
ASCI Q	8192	6.5 hours, 114 unplanned shutdowns/month
ASCI White	8192	5 hours (2001), 40 hours (2003)
Seaborg	6656	14 days
Lemieux	3016	9.7 hours
Google	more than 100k	20 reboots/day
Abe	9600	6 hours
BlueGene	65535	6.16 days

the next equation:

$$MTBF = \int_0^\infty R(t) = \int_0^\infty e^{-\lambda t} = \frac{1}{\lambda} \approx \frac{1}{1 - R(t)} \tag{2.5}$$

Table 2.1 shows the measured MTBF for several real large-scale high-performance computers. These values where obtained from [3, 11, 73, 91, 108, 114]. Notice that MTBF is expressed in hours in most of the cases, but complex computational tasks on supercomputers often last for in days. For instance, the complete simulation of the 50 models of the Hafnium Gate Material takes approximately 250 days on a BlueGene/L cluster [14]. As it can be seen, fault-tolerance is a must in cluster-based supercomputers.

Availability is defined as the probability that a system operates correctly and is available at a given time. Availability differs from reliability in that reliability involves an interval of time whereas availability involves an instant of time. A system can be highly available despite experiencing frequent periods of inoperability as long as the duration of these periods is extremely short. In other words, availability depends not only on how often a system becomes inoperable due to faults but also, how quickly it can be repaired. In particular, in case of interconnection networks, availability defines the average fraction of the total connection time that it is expected to be running. Availability can be computed as a function of the MTBF and the Mean Time To Repair (MTTR):

$$A(t) = \frac{MTBF}{MTBF + MTTR} \tag{2.6}$$

Finally, dependability is used to encapsulate the concepts of reliability and availability. Dependability indicates the quality of service provided by a particular system [72].

Regarding to faults, they are commonly classified depending on their duration. Permanent faults are the faults that affect a component of the system forever. For example, the physical implementation of the adder of an ALU may introduce a permanent fault that prevents the adder to activate its overflow signal. However, they may not be present since the creation of the system, like in the example. Permanent faults may appear during the normal operation of the system, but once they appear they last forever. Usually, this kind of faults can be more easily solved than the transient faults.

On the other hand, we have transient faults. Transient faults may appear only during a period of time of the system operation, and they may be repeatable. A transient fault may appear under certain circumstances, so it is repeated every time that these circumstances are met. A real example of transient fault was observed in the computers from a logistic company, everyday from 7pm to 8pm their computers on the warehouse could not use the printer located at the main office of the building. After investigating the case, they realized that at 7pm the freezer was programmed to recycle all the air inside it, which involved getting the cooling engines of the freezer to work at their maximum potency. Unfortunately, the engines of the freezer were creating an interference with the cable of the printer that was deployed near them. As it can be seen, transient errors may be very tricky to track down and solve.

Another complementary classification of the faults comes out if we consider the behavior of the faulty component. In on-off faults, the faulty component just stops working; it does not response to the rest of the system, and does not communicate in any way with it. A typical on-off fault in a cluster is a processing node that stops working due to some hardware error. On the other hand, we have byzantine faults. In this type of faults, the faulty element may seem to work properly, but it is introducing wrong information in the system. A typical example of byzantine fault in computers is a virus or malware that tries to corrupt the file system of the machines.

Fault-tolerance may be achieved in several layers of a system. It may be introduced at the hardware level, at the software level or at the data level.

In hardware fault-tolerance, the hardware itself is the responsible of providing fault-tolerance in the system. For example, most of the flight control pieces of hardware in an airplane are developed in a redundant way to tolerate a limited number of faults. In the same way, fault-tolerance may be introduced by the software of the system. For example, the program that tracks the trains in a subway system has several modules that check the results of the program to avoid train collisions. The last type of fault-tolerance is the data-based fault-tolerance. In this kind of fault-tolerance the data of the system provide enough information to check its correctness. The typical example of data-based fault-tolerance is the *CRC* checksum code in the packets of a network, since with the *CRC* the receiver of the packet can know if the packet has suffered some modification since it was injected into the network [100].

Finally, a fault-tolerant technique can be classified depending on the additional elements that it requires to work. A wide range of fault-tolerant techniques are based on replication. In this approach, we have several copies of the system working in parallel. So, in case one of them suffers a fault, it is switched off and the other copies of the system copies keep working. As it can be seen, these techniques have a big cost overhead, not only in implementation due to having several copies of the entire system, but also in power consumption, as all the copies of the system are working in parallel all the time. A more cost-effective approach consists on having spare resources, which usually are switched off, and are only switched on to replace the main system when it fails. In this case, the problem of power consumption is solved, since the spare copies are not working, but the overhead on implementation cost is kept high. Finally, the most cost-effective techniques do not require any kind of replicated or spare element, since they are based on the reconfiguration of the system. These techniques are able to reconfigure the system in order to tolerate a fault without requiring any additional hardware. Nevertheless, the scope of tolerated faults with the reconfiguration techniques is narrower than the one from replication-based techniques.

Reconfiguration techniques are usually classified depending on how they model the faults. In reconfiguration techniques that follow a static fault model, each time a new fault is detected, the system activity is stopped; appropriate actions to handle the fault are taken and then, the system activity is resumed. It needs to be combined with checkpointing techniques to be effective [104, 125]. On the other hand, in the reconfiguration techniques that follow a dynamic fault model, once a new failure is found, actions are taken in order to appropriately handle the faulty component in parallel with the system activity, avoiding the need for halting the network and checkpoint techniques. Given a large network with a high fault frequency, static fault-tolerance may be inefficient since the system will be stopped frequently and restarted from the last performed checkpoint.

2.1.8 Network Metrics

When comparing different routing algorithms there are several parameters to consider, depending in which ones we decide to compare the conclusions may vary. For this, in this section, we summarize the network metrics that are usually used when comparing routing algorithms, and some specific metrics for fault-tolerant routing algorithms.

Performance

From the point of view of the performance, there are two main parameters to consider when comparing different routing algorithms. These two parameters are the average latency of the packets and the throughput of the network. The latency of a packet is the elapsed time from the packet injection into the network till its reception at the destination node. This latency is also referred to as network latency, in opposition to latency from generation which is the elapsed time from the generation of the packet to its reception in the destination node. Latency from generation includes network latency plus the waiting time at the NIC of the source and destination nodes. On the other hand, network latency only includes the time introduced by traversing the network. If no other thing is stated, from now on we will refer as latency to the network latency. Usually, the best routing algorithm is the one that provides the lowest average latency of packets.

Throughput is the amount of information that the network delivers to the processing nodes per time unit. To make this metric independent of the network size, throughput is measured as the total information delivered to all the processing nodes divided by the number of processing nodes per time unit, decoupling the efficiency of the network in delivering information from the network size. When comparing routing algorithms from the point of view of the throughput, the higher throughput a given algorithm can reach, the better the algorithm is considered.

Figure 2.13 shows the typical behavior of a network. The figure represents the typical relation between average network latency of packets and the accepted traffic of the network. As it can be seen in the figure, average latency of packets increases as the accepted traffic of the network increases. This happens because as the accepted traffic of the network increases, the number of packets traveling through the network also increases, raising the number of packet collisions at the switches, thus increasing the waiting times of the packets at the switches of the network. This increase grows exponentially when the network becomes congested; at this point the latency rises quickly. The network becomes congested when the number of packets trying to traverse it is so high that almost all its channels and buffers are occupied, so the waiting time (latency) of the packets skyrockets. This situation highly resembles to a traffic jam in a city. In a city, latency would correspond to the elapsed time of the cars traveling along its streets. By its side, accepted traffic would correspond to the number of cars that reach their destination per hour. As the number of cars in the streets of the city raises, the number of cars that reach their destination (accepted traffic) grows, but also the latency, since it would be more frequent that cars make other cars to stop during their trips in stop signals or narrow streets, for example. If the number of cars is increased too much, a traffic jam happens, the city streets become congested of furious drivers, and their traveling time may rise to infinity. This is exactly what happens in a network when it becomes congested.

Notice that when congestion is reached accepted traffic cannot be longer increased. This amount traffic is the upper bound for the accepted traffic in the network and it is denoted as Θ_{max} or throughput. Furthermore, it may happen that accepted traffic decreases if the injected traffic into the network increases beyond the saturation point. In addition, the figure also shows the minimum possible latency or latency lower bound, denoted as T_0; which represents the average latency of packets in a network without collisions of packets (zero load

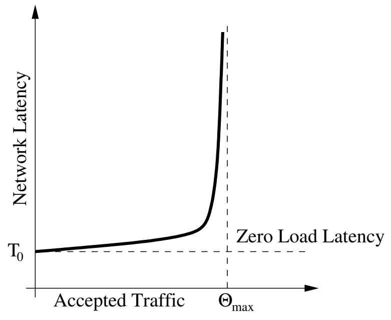


Figure 2.13: Ideal performance graphics of a network.

latency).

Fault-Tolerance

Concerning fault-tolerant routing algorithms for high-performance interconnection networks, there are several additional parameters to the previous ones to consider. The first and, in most cases, the most important one is the number of tolerated faults, which indicates the maximum number of tolerated faults for an interconnection network. We say that an interconnection network can tolerate n faults when all the combinations of n faults are tolerated. A combination of faults is tolerated if the routing algorithm can accomplish the two following conditions: it must be able to provide at least one path without faults between all the origins and destinations, and it must not provide any path that leads a packet to a fault. In this way, the routing algorithm preserves the connectivity of the network, and at the same time, ensures that all the packets are delivered to their destinations. Obviously, in a fault-tolerant routing algorithm, it is desirable to have the highest possible number of tolerated faults. Notice that if a system tolerates n faults, it may tolerate a combination of a higher number of faults, but it is not assured.

In Section 2.1.7, availability was defined as a function of the MTBF and the MTTR. The MTTR in a reconfiguration technique is referred to as reconfiguration time. Reconfiguration time defines the time required by a fault-tolerant technique to completely reconfigure the system once a new fault is detected. Reconfiguration time should be as short as possible to keep high the availability of the system.

Moreover, in a dynamic fault-model, the reconfiguration of the network is performed without stopping the system activity. This implies that during reconfiguration there could be several packets that are dropped due to encountering faults in their paths, since the reconfiguration process has not yet been completed. The number of dropped packets is strongly related with the reconfiguration time. If the reconfiguration takes a long time, there will be a higher number of lost packets. Ideally, the number of lost packets should be zero, but as this is not usually possible. The fault-tolerant algorithm should keep the number of lost packets as low as possible.

Finally, we should also analyze the degradation on performance after reconfiguration is finished. In a fault-tolerant system, it is assured that the system will continue working after failing. Nevertheless, as the system is working with fewer resources than in the case without faults, it is very common that the system run in a degraded state. In this degraded state, it is not possible in most cases to keep the same performance than in the case without faults. For this, it is important to compare the performance degradation of the different fault-tolerant techniques, since the degradation is an indicator of the efficiency of the fault-tolerant technique. If the degradation of the performance is very high, this indicates that despite the fault-tolerant technique can tolerate the faults, it cannot efficiently utilize the remaining healthy resources.

2.2 State of the Art

In this section, we provide the results of our search along the literature in the two fields that are mainly treated in this dissertation: fault-tolerance and routing algorithms for fat-trees. First, we analyze the proposed works for fault-tolerance in MINs. Following, we summarize the main proposals in fault-tolerance for interconnection networks, not only for MINs. Finally, we focus on the routing algorithms that either have been proposed or implemented in MINs.

2.2.1 Fault-Tolerance in MINs

A large amount of work in fault-tolerance for MINs is based on using additional hardware to increase the number of alternative paths by either adding links between switches in the same stage, more links between stages, more switches per stage or extra stages. All these techniques are based on replicating resources, and therefore they significatively increase the cost of the network and complicate its design and implementation. Most of these techniques are also based on a static fault-model, requiring the introduction of checkpointing techniques. Checkpointing is a very time and resource-consuming procedure, since it relies on maintaining a copy of the state of the machine, including memory images, sent and received packets, and so on. As the reader may notice, this involves a high amount of traffic, processing and storage capabilities. Therefore, fault-tolerant techniques based on a static fault-model and replication has a double overhead on cost, and may become very expensive, specially if they are compared to techniques based on dynamic reconfiguration. A good example of such techniques is [94], where the authors perform a comparative study of four fault-tolerant MINs. The study reveals that columnwise redundancy (extra stages) is much more effective than the row-wise approach (extra rows of switches along with extra links between switches). This conclusions are reflected on the Advanced Baseline topology [79], which basically is a Baseline MIN with additional stages specifically added for the sake of the fault-tolerance.

Another approach is to keep an static fault-model while using reconfiguration techniques. One possibility consist of misrouting packets by routing them in multiple passes [28, 74, 129], providing therefore longer paths and increasing the average packet latency. The basis of these works are the analysis performed at [66, 96], which show that under a certain high probability after a fault appears, most MINs keep their connectivity. This kind of techniques exploit a property that is usually accomplished by MIN networks, the dynamic full access property [122]. In a network that accomplish such property, the network is able to provide an alternative path for all the paths affected by faults by routing the packets that would cross the faulty elements through other nodes of the network, which involves injecting the packet into the network several times. Indeed, Varma et al. [129] show that most of the fault combinations can be avoided by reinjecting the packets $\log_2 N - 2$ times. In a network with 64 nodes, a packet can be injected up to 6 times to avoid the faults in the network. As it can be seen, the increase in the latency is significative, and highly decreases the performance of the network. In [121], the author proposes a hybrid approach combining adding hardware elements and multiple passes to forward packets in a faulty network.

An important work for reconfiguration techniques in a static fault model is [46]. This contribution shows that reconfiguration techniques that rely on a centralized element that perform all the reconfiguration process are not probabilistically feasible, and provide an alternative approach allowing to disable healthy elements of the network to keep the network fully reconfigurable. A similar technique is presented in [75] where the authors propose to create a new, but equivalent, topology from the faulty one, thus preserving the routing algorithm at the cost of disabling some healthy nodes and network elements. This works are supported by Chong et al. [30], where different choices in the construction of multiple paths in MINs are analyzed, and the authors describe methods for fault identification and network reconfiguration in MINs. They found that to achieve a good computational performance, it is necessary to eliminate nodes with poor network connectivity. Despite of the strengths of the proposals, disabling healthy network elements and nodes is a prohibitive waste of resources, since these nodes are not used despite the fact that they are not faulty.

If we focus on fault-tolerant techniques that follow a dynamic fault model, we should refer to [35]. The technique presented in this paper is based on exploiting the high number of alternative paths of the MINs to provide fault-tolerance. When a fault is detected, the network reconfigures the routing tables of the nodes. In this way, paths that use any of the faulty network elements are disabled. This provide a good fault-tolerance at almost no overhead in cost. Nevertheless, this technique was proposed for source routing, and it relies on several characteristics that are exclusive on that type of routing. Moreover, as commented in Section 2.1.6, source routing cannot usually take full benefit from adaptive routing. There are several works [43,70,71,128] that mix this proposal with replication of links to increase the number of routing options, increasing unnecessarily the cost of the network. In [117], the authors use several parallel MINs to create redundancy without any interconnecting links among them, highly increasing the hardware cost of the network. In this technique, if a given destination is not reachable due to faults in one of the networks, packets to that destination are sent through any of the other parallel MINs. In [116], a new topology that consists of two parallel fat-trees with crossover links between the switches in the same position in both networks is proposed. In this topology, when a packet encounters a fault in its path, it is forwarded through the crossover link to the other parallel fat-tree. This approach is able to tolerate only one faulty link with a very high extra hardware cost. The same authors propose a much cost-efficient technique in [115]. In this case, the technique does not require any additional hardware, and it is based on misrouting the packets when they encounter a fault. The authors provide a wide analysis to ensure that the misrouted packets cannot form a deadlock. In this dissertation, we make use of a similar mechanism, but only during reconfiguration time in order to avoid losing packets, since making use of misrouting as the only technique to avoid faults incurs in a non-desired latency increase. In [118, 127], like in the previous technique, the switches are the responsible of providing equivalent routes to avoid faults. Nevertheless, this technique relies on adding several links between the switches of the same stage. From all the analyzed techniques, considering only the ones that do not require extra network resources, the most powerful ones can tolerate the maximum number of faults that can be tolerated without increasing the resources of the network, which is $k-1$. If k or more faults are present in the network, the network can have become unconnected, therefore, it is impossible to tolerate such number of faults without adding new resources.

In order to extend our analysis and search other alternatives for fault-tolerance in interconnection networks, we extend the scope of our analysis to all the topologies, not only the MIN topologies. Chalasani et al. [26,27] devoted their work to create a fault-tolerant mechanism that allows packets to border faulty blocks in meshes and tori networks by creating rings surrounding these fault blocks. The technique requires that network faults are localized in blocks to work at full potential without disabling healthy nodes, and requires the use of five virtual channels. A similar approach is proposed in [105,106] for k-ary n-cubes, where the network dynamically reconfigures itself creating rings around arbitrary fault blocks. This proposal reduces the number of required virtual channels to three. A different approach is exposed in [63], where the authors propose to route packet avoiding faults by using intermediate nodes. This routing through intermediate nodes, may be done in several ways, requiring at worst one virtual channel, and no virtual channels in the best case. In [103], the authors propose to create two virtual networks, allowing to use one of them while the other is being reconfigured due to faults. This technique can be used in almost any topology with any routing scheme, but it require two virtual channels. As it can be seen, the authors tend to reduce the number of required virtual channels, since virtual channels increase the cost and complexity of the switches of the network.

For meshes with deterministic routing, in [77], it was proposed a technique that can tolerate a huge number of faults without needing virtual channels, however, it is based on a static fault model. This work was extended to wormhole adaptive routing by using a static fault-model in [123]. This approach is based on reconfiguring the routing tables of the switches near the faults, which is the fundamental concept of the work presented in [82]. The work in [25] describes a method of creating routing tables for irregular faulty networks in order to avoid multiple faults. However, the method does not take advantage of the topology to provide an optimum solution. Finally, Glass et al [52] demonstrate that the maximum number of tolerated faults in a n-dimensional topology is $n-1$, which is the degree of the switches in those network. In fat-trees, this result is equivalent to $k-1$, which is the limit of tolerated faults without adding extra network resources.

2.2.2 Routing in Commodity Fat-Trees

The search for a good routing algorithm specific for fat-trees began with [24]. In that paper, the authors compare several wormhole adaptive routing algorithms for several topologies including the fat–tree. However, the conclusions could not be widely applied to current high-performance fat-trees, as most of them are not wormhole-based. Indeed, a few years later, Petrini et al. [101] proposed to recover the ascending and descending routing scheme from [112], explained in Section 2.1.6. Furthermore, this scheme is the one used on the immense majority of fat-trees. The main reason of its popularity is that this scheme has several good features, like allowing to use deadlock-free minimal path routing while providing at the same time several redundant paths between all nodes. This routing scheme was even more supported in [64], where the authors show that in fat-trees is not worth to implement topology agnostic routing algorithms or to customize routing algorithms from other topologies to the fat-tree one. Furthermore, they propose a random arbitration policy for switches and NICs that try to deal with hot-spots and avoiding to saturate the network, which have become two of the main problems in modern high-performance multistage networks [101]. Nevertheless, as the proposed mechanism relies on a fully randomized function its benefits are not clear, and can even negatively affect the network performance, at worst.

Most of the research efforts on routing for fat-trees has been targeted to a given high-performance network technology, since fat-trees have become the default or recommended topology of several high-performance interconnects vendors, such as Myrinet [8], Quadrics [9], and Infiniband [7]. These technologies provide very different characteristics. For example, Myrinet and Quadrics provide adaptive routing, while standard Infiniband can only provide deterministic routing. On the other hand, Myrinet uses source routing, Infiniband only uses distributed routing based on forwarding tables, and some generations of Quadrics have used source routing and others have used adaptive distributed routing [9, 102]. For this, some of the research efforts in this field have been made to deal with the limitations of each of these technologies.

In [85], Martinez et al. propose an extension of Infiniband switches to enable the use of adaptive routing, but it requires significant changes in the switches and thus is not currently implemented. The purpose of introducing adaptive routing in Infiniband switches was to balance the traffic along the network. In [36], the authors try to achieve this traffic balance by proposing an optimized routing algorithm that requires global information, which makes this approach almost unfeasible. On other hand, Lin et al. [78] propose to exploit the capacity of creating virtual destinations in Infiniband to develop a deterministic routing algorithm that spread the paths for each physical destination among all the network. In other words, each source node has a different and disjoint path for the same destination. Thus, in this deterministic routing algorithm, node i sends its packets to node d for a totally disjoint path from the one used by node j ($i \neq j$) to send packets to node d. This routing algorithm was finally implemented in [130], but it has several technical problems, for example the requirement of large routing tables in all the switches, which renders this approach rather impractical for medium and large networks. Moreover, despite that it attenuates the negative effects of hot-spots for the packets directed to the hot-spots, it does not consider the tremendous impact of scattering the traffic of the hot-spot all over the network, affecting the traffic that is not directed to the hot-spots. Another approach to deal with hot-spots in Infiniband is presented in [45], where the authors propose a mechanism to content the congestion of hot-spots by throttling the sender nodes.

Finally, Geoffray et al. [47] propose an adaptive routing strategy for Myrinet networks. This strategy uses a deterministic routing algorithm when the network is not congested. When a sender node detects a possible congestion by using the backwards pressure information from the Myrinet flow control mechanism, it switches to adaptive routing in order to avoid the congested zones. In order to make a more accurate detection of congestion, the authors propose to establish a threshold in the sender nodes before starting to use the adaptive algorithm. In addition, the authors propose using a probing system to detect which of the possible adaptive paths are also suffering from congestion, in order to discard them. However, several nodes change may at the same time from deterministic routing to adaptive routing. In this case, the adaptive paths that are not suffering the congestion that caused the change may become congested if all the nodes try to use them. In order to avoid this effect, the authors propose that sender nodes may start the switch from deterministic to adaptive routing with a random probability. This technique present several weak points. For example, the authors state that the problem with the algorithm is that it does not guarantee that packets are delivered in order. Furthermore, they do neither establish which deterministic routes are used, or which adaptive routes can be used, or how they select the probability of changing to adaptive, or which is the value of this probability, or how they establish the threshold for this change. Moreover, they rely on specific mechanisms from Myrinet technology that may not be available in other technologies, limiting the applicability of this technique. Finally, it results in a complex system from the implementation point of view due to the probing system, and the statistics that must be implemented at low level for thresholds.

Chapter 3

FT^{2}EI : Fault–Tolerant Fat-Tree with Exclusion Intervals

"We're a bit of a specialized hospital. We generally only deal with patients when they're actually sick."

House, House MD tv show.

Nowadays, high-performance computers have thousands of nodes [2,4,126]. In such systems, fault-tolerance in the interconnection network is an issue of growing importance since the high number of components significantly increases the probability of failure. Most of these high-performance computers are clusters-based machines, and the fat-tree is the most frequently used topology.

Fault-tolerance mechanisms proposed up to now for fat-tree-based high-performance clusters are based on either adding new resources to the network such as more stages, switches or links; eliminate healthy computing nodes from the network; or increasing the length of the paths of the packets permanently once a fault appears. All these techniques lead to increase the cost of the network –in some cases it is even doubled– or to waste the investment made on the nodes that are healthy but are not used due to the fault-tolerance mechanism.

There exists another powerful technique to tolerate faults in interconnection networks which is based on dynamically reconfiguring the routing tables. This technique is extremely flexible but it may kill performance, since it is usually based in the use of generic routing algorithms that achieve a lower performance than a routing algorithm that takes advantage of the topology [113].

In cluster-based computers, routing is usually distributed and based on forwarding tables. In the forwarding tables scheme, there is a table at each switch that stores, for each destination node, the output port that must be used. This scheme can be extended to support adaptive routing by storing several outputs in each table entry [85]. The main advantage of table-based routing is that any topology and any routing algorithm can be used. However, routing based on forwarding tables suffers from a lack of scalability, as table size grows linearly with the network size and the time required to access the table also depends directly on the table size.

For these reasons, in this chapter, we propose a new mechanism for fault-tolerance in fat-trees that does not require to add new resources to the network, providing fault-tolerance without increasing the cost of the network. In addition, it never eliminates any healthy node from the network. Moreover, it allows fully adaptive routing along the healthy paths of the network. And, finally, the mechanism keeps the minimality of the paths followed by packets under any number of faults. The mechanism is based on taking advantage on the high number of alternative but equivalent paths provided by the fat-tree topology. To deal with the scalability problem of the forwarding tables, we propose to use Interval Routing (IR) instead of using table-based routing.

This chapter introduces in Section 3.1 several basic concepts about IR implementation and configuration for adaptive routing in fat-trees. Following, Section 3.2 describes our initial fault-tolerant mechanism for fat-trees based on a static fault model, which relies on stopping the network each time a new fault is detected. This mechanism is evolved to a dynamic fault model in Section 3.3, which can work without stopping the interconnection network activity. On one hand, with the static fault model, we focus on detecting where our mechanism should act for each fault. On the other hand, with the dynamic fault model, we focus on how to distribute the fault-tolerance information along the network. Both models are evaluated in Section 3.4. Finally, Section

3.5 draws some conclusions.

3.1 Introduction

The objective of this chapter is to propose a new fault-tolerant adaptive routing algorithm for fat-trees that does not require to stop the network activity, that scales with network size, and that has low storage requirements. For this, our proposal is based on the fully adaptive routing scheme proposed by Petrini for the fat-trees [101] and on Interval Routing [111] to provide a scalable implementation of the routing algorithm.

Interval Routing (IR) is a scalable distributed routing scheme based on grouping the destinations that are consecutive and physically reachable from the same output port into an interval. Each packet is forwarded through the output port whose interval contains the destination of the packet. To implement IR, it is sufficient to store the bounds of each interval. That is, all the ports of the switches have two registers that store the bounds of the interval. We will refer to LIB (Lower Interval Bound) as to the register that stores the lower bound of the interval of a port and to UIB (Upper Interval Bound) as the register that contains the upper bound of the interval of a port. Additionally, we refer to routing or inclusion interval of a port as the interval formed by its LIB and UIB registers.

Usually, the upper bound of an interval has a higher value than its lower bound. This is the case of a typical interval which includes all the destinations that are between LIB and UIB. Nevertheless, IR does not require that the upper bound of the interval has a higher value than the lower bound. In this way, we can construct cyclic intervals, also called modulo N intervals, being N the number of destinations. In cyclic intervals, the lower bound of the interval is higher than its upper bound. This kind of intervals include all the destinations that have a higher value than the one stored in LIB, and all the ones that has a lower value than the one stored in UIB. For instance, the interval [1..3] is not cyclic and it includes all the nodes between 1 and 3, both included. On the other hand, the interval [3..1] is a cyclic interval and it includes all the nodes that are higher than 3 and all the nodes that are lower than 1. This cyclic interval is equivalent to the union of the intervals [0..1]

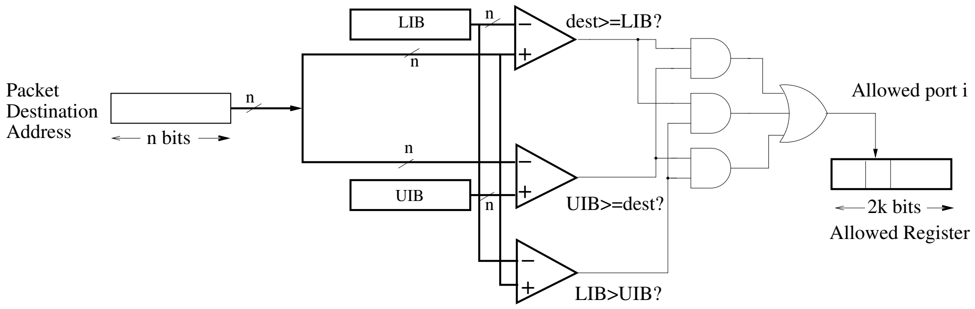


Figure 3.1: IR hardware associated to each output port.

and $[3..N-1]$.

Taking into consideration the use of cyclic intervals, the condition that a destination must meet to be inside the routing interval depends on the relative values of UIB and LIB. If $LIB \leq UIB$, the condition to meet is $Dest \geq LIB \wedge Dest \leq UIB$, whereas if $LIB > UIB$ (cyclic interval), then the condition is $Dest \geq LIB \vee Dest \leq UIB$.

The logic to check this condition is very simple, as can be seen in Figure 3.1. The logic only returns a value different to zero if the destination of the packet that is being routed can be reached through the port at which the logic is implemented. As it can be seen, the result of this comparison is stored in the i^{th} position of a register called Allowed ports Register (AR), being i the port at which the logic is located. AR is unique per switch, and in k-ary n-trees it has a size of $2k$ bits ($2k$ is the number of switch output ports). This register stores the result of the routing function. In IR, routing is performed as a parallel comparison of all the output ports. In other words, each time that a packet must be routed in a switch, its destination is compared with the intervals of all the output ports at the same time. In this way, the routing time does not depend on the switch degree. These operations introduce a very low delay, much smaller than the one incurred by forwarding tables. Notice that, as adaptive routing is supported, a given destination address may be inside several intervals, and more than one output port can be allowed in the AR. Finally, the selection function will select the output port of a packet from the set of ports included in the AR of the switch.

As explained in the previous chapter, in a k-ary n-tree, minimal routing from a source to a destination can be accomplished by sending packets forward to one of the nearest common ancestors of source and destination and then, from this common ancestor, downwards to the destination [40]. When crossing stages in the forward direction, several paths are possible, so adaptive routing is provided. In fact, each switch can select any of its upwards output ports. Once a nearest common ancestor has been reached, the packet is turned around and sent downwards to its destination. Once the turnaround is crossed, a single path is available to the destination node. That is, the upwards phase is fully adaptive while the downwards phase is deterministic. The stage to which the packet must be forwarded up is obtained by comparing the source and destination components beginning from the the most significant one. The first pair of components that differs indicates the last stage to forward up the packet.

This routing algorithm can easily be implemented in IR. Figure 3.2 shows the possible paths for a packet generated at node 0 whose destination is node 2. The figure also shows the routing interval that can be used along these paths. As it can be seen, the routing intervals of the output ports of switch 0 are [0..0] for link 0, [1..1] for link 1, and [2..7] for links 2 and 3. In the previous chapter, it was presented the following example in Figure 2.12: a packet generated at node 0 whose destination is node 2 will be forwarded up to stage number one (through switch $\langle 0, 00\rangle$ and choosing either path to $\langle 1, 00\rangle$ or $\langle 1, 01\rangle$). From any of these switches, the remaining bits (bits 1 and 0) of the destination identifier (10 in our example) correctly forwards the packet to node 2. As it can be seen in Figure 3.2, the paths obtained by using IR are the same ones that where obtained in that example. But, in this case, the routing is not made by using the components of the destination node identifier, it is done by comparing the routing intervals of the ports of the switches with the destination identifier. Following the same example than in the previous chapter but using IR, when a packet sent from node 0 to node 2 arrives to switch 0, the identifier of the destination of the packet (node 2) is compared with all the routing intervals of the switch. The only output links whose routing intervals contain destination 2 are links 2 and 3. So, the packet can be forwarded through any of those links, reaching switch 4 or 5. In any of these switches, the destination of the packet is compared with all

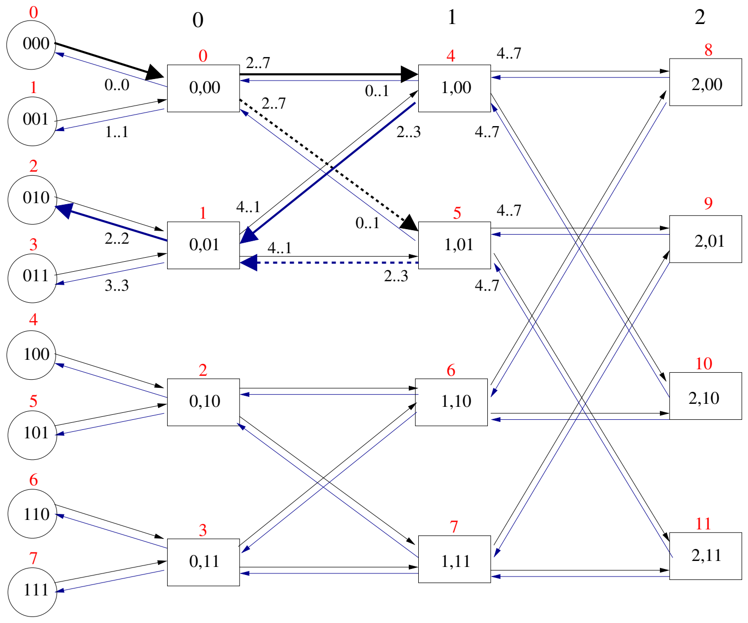


Figure 3.2: A 8-node 2-ary 3-tree. Paths coming from node 0 to node 2 are highlighted.

the routing intervals of the output ports, and the only one that contains the destination node is the link 1. Finally, the packet reaches switch 1, where again the destination identifier of the packet is compared with all the routing intervals, and the only one that contains it is the one corresponding to link 0, through which the packet reaches node 2. Figure 3.2 shows that IR can be used to provide fully adaptive routing in fat-trees, since it obtains the same paths than the ones obtained in the previous chapter.

Obviously, the figure shows only a small subset of the routing intervals. In IR, all the switches of the network have a routing interval per output port, and these intervals have to be correctly filled in order to provide fully adaptive routing in a fat-tree network. The procedure to fill the routing intervals of a switch to provide fully adaptive routing in fat-trees is shown in Figure 3.3. The prototyped switch represented in the figure is labeled as $\langle s, o_{n-2}, o_{n-3}, \ldots, o_1, o_0 \rangle$. Notice that this switch is located at the stage s. In order to fill the routing intervals of this switch, we have to identify all the destinations that are reachable through all its output ports. First, we identify

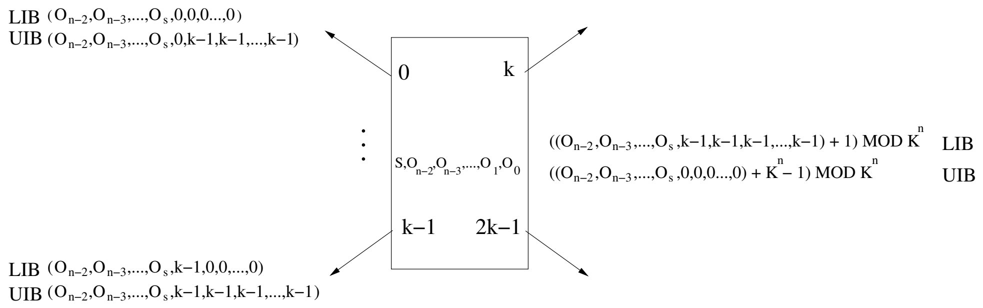

LIB (On−2,On−3,...,Os,0,0,0...,0)
UIB (On−2,On−3,...,Os,0,k−1,k−1,...,k−1)
0
k
S,On−2,On−3,...,O1,O0
((On−2,On−3,...,Os,k−1,k−1,k−1,...,k−1) + 1) MOD K^n LIB
((On−2,On−3,...,Os,0,0,0...,0) + K^n − 1) MOD K^n UIB
k−1
2k−1
LIB (On−2,On−3,...,Os,k−1,0,0,...,0)
UIB (On−2,On−3,...,Os,k−1,k−1,k−1,...,k−1)

Figure 3.3: Prototyped IR register configuration for adaptive routing in fat-trees.

the destinations reachable through the descending links, and later the rest of destinations reachable through all the ascending links.

The set of destination nodes ($\langle p_{n-1}, ..., p_1, p_0 \rangle$) that are reachable by the descending links can be easily computed from the switch components. In particular, the set of destination nodes that can be reached by this switch through its down output ports are the ones whose components accomplish the following expression: $p_i = o_{i-1}$ for $i \in \{n-1, .., s+1\}$. This set of destinations is split in several subsets with the same number of elements that can be reached from each descending link depending on the p_s component, being s the switch stage. The subset of nodes whose p_s is equal to 0 are reachable through link 0, the subset of destinations whose p_s is 1 are reachable through link 1, and so on. As an example, link 0 of switch $\langle s, o_{n-2}, ..o_1, o_0 \rangle$ forwards packets destined to nodes $\langle o_{n-2}, ..., o_s, 0, X...X \rangle$, that is, the routing registers for link 0 of that switch are $LIB_{desc.} = \langle o_{n-2}, ..., o_s, 0, 0...0 \rangle$ and $UIB_{desc.} = \langle o_{n-2}, ..., o_s, 0, k-1...k-1 \rangle$. Using the network depicted in Figure 3.2, switch 4, whose identifier is $\langle 1, 00 \rangle$, can reach through its down output ports the nodes whose most significative component are equal to 0, that is nodes from 0 to 3. This interval can be represented as [000...011], and has to be split in two subintervals since the switch has two down ports. The routing interval for link 0 is [000..001], that is it can forward packets to nodes 0 and 1. Finally, the routing interval for link 1 is [010..011], that is it can forward packets to nodes 2 and 3.

Filling the routing intervals for the up output ports of the switch is very straightforward, since they only have to route packets to the destination nodes that where not included in the routing registers of the down output ports. Concretely, the destinations that are not reachable through the descending links, so they are reachable through the ascending links are the ones that do not meet $p_i = o_{i-1}$ for $i \in \{n-1, .., s+1\}$. Indeed, this set is reachable through all the k ascending links, since all they have to route the same interval to provide fully adaptive routing during the upwards phase of the routing algorithm. The LIB register of the ascending output ports must store the next destination to the largest one reachable through the descending links. That is, the next destination to the one stored in the UIB register of the $k-1$ descending link. Likewise, the UIB register of the ascending links must store the previous destination to the smallest one reachable through the descending links. That is, $LIB_{asc.} = (UIB_{link_{k-1}} + 1) \mod k^n$ and $UIB_{asc.} = (LIB_{link_0} - 1 + k^n) \bmod k^n$, being k^n the number of nodes in the network[1]. This interval is valid for all the ascending links of the switch and can result in a cyclic interval. Finally, Figure 3.4 shows all the routing intervals of a 2-ary 4-tree.

3.2 Static Fault-tolerant Routing with Exclusion Intervals

The fault-tolerant routing methodology proposed in the following sections provides fault-tolerant adaptive routing for fat-tree networks. Concretely, we only consider permanent on-off faults (see Section 2.1.7), because transient faults can be handled by communication protocols, by means of CRC mechanisms to detect faults and retransmitting the packets that are detected as incorrect by the CRC mechanism. The methodology works with both link and switch faults. However, a switch fault can be modeled by the fault of all the links connected to it. Therefore, we focus only on link faults.

First, we use an approach based on a static fault model. In this first approach, each time a new fault is detected, the system activity is stopped, then the new routing information is computed and updated, and finally the system activity is resumed from the last performed checkpoint. The proposed

[1]For $UIB_{asc.}$ we add k^n in order to avoid setting a negative destination value.

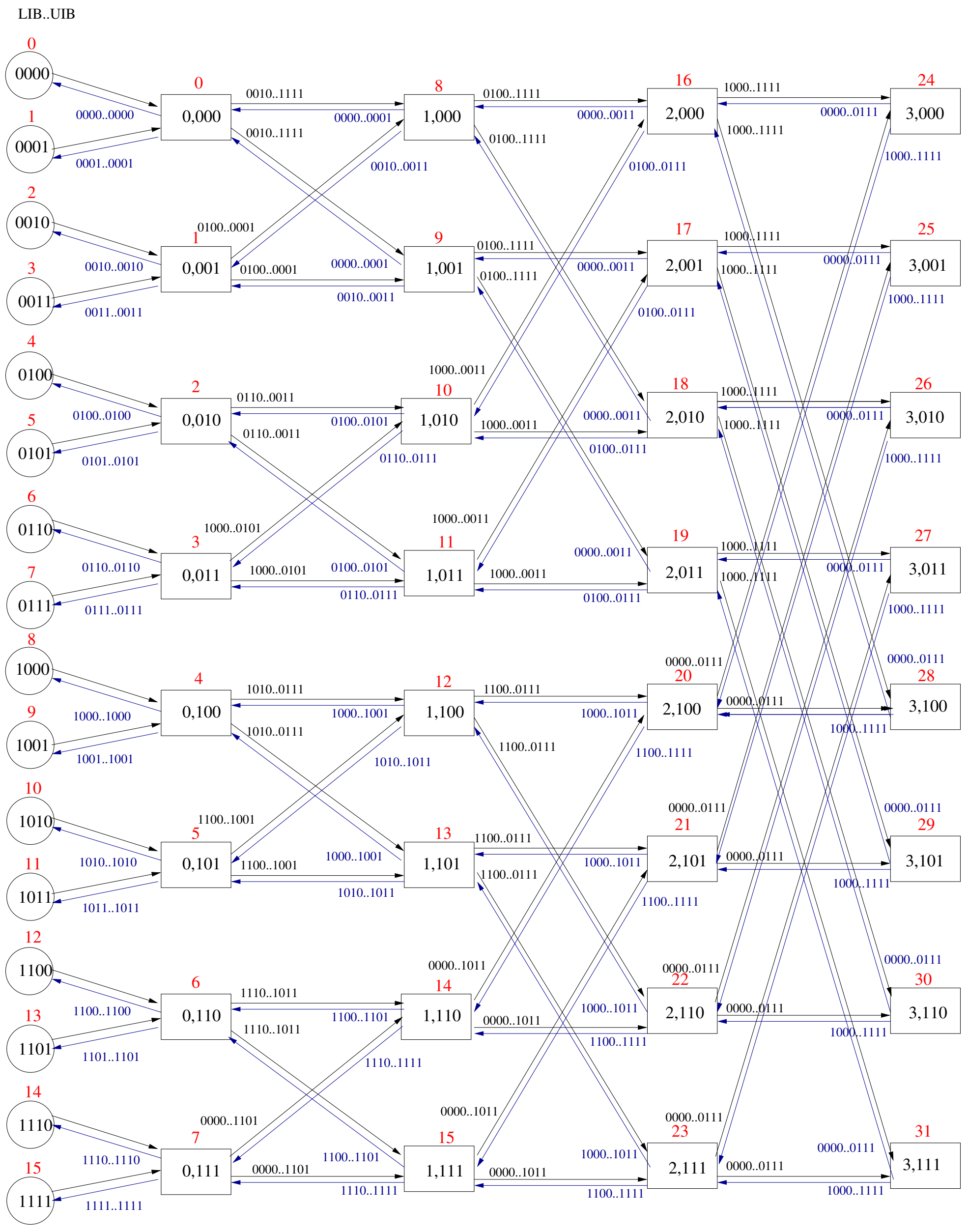


Figure 3.4: Adaptive routing with IR in a 2-ary 4-tree.

methodology in this section is focused only in the computation of the fault-tolerant routing information. Fault detection can be done as proposed in [124], where the faults are detected using timers. Concerning checkpointing, any of the checkpointing techniques proposed in the literature can be suitable, for example the ones proposed in [125].

As commented, our proposal uses Interval Routing (IR) for adaptively routing packets in k-ary n-tree networks. Our basic idea is to extend IR with *exclusion intervals* for the purpose of providing fault-tolerance. The original IR uses an interval per output port to indicate the reachable destinations through that output port. We associate a new interval called exclusion interval to each output port. The exclusion interval indicates the destinations that become unreachable through that output port after a fault. It contains some nodes that belong to the inclusion interval, but due to a fault, they must be excluded. Therefore, the nodes that are reachable through an output port are the ones that are in its inclusion interval but are not in its exclusion interval. For instance, assume that the routing interval associated to a given output port is [0..11]. Assume also that, due to a fault, the interval [4..7] must be excluded in this output port. Then, the set of reachable nodes is reduced to $[0..3, 8..11]$. In this way, we can avoid sending packets through paths that use faulty links. In Section 3.2.1 we show an example of computing the exclusion intervals for a faulty network. In order to maintain a notation similar to the one used in IR, we refer to $ELIB$ and $EUIB$ as the registers that contain the bounds of the exclusion interval associated to the output ports of the switch. As our fault-tolerant mechanism relies on the use of exclusion intervals, we will refer to it as FT^2EI (Fat-Tree Fault-Tolerant routing with Exclusion Intervals).

Figure 3.5 shows the hardware associated to each output port to implement FT^2EI. As it can be seen, it is an extension of the one presented in Figure 3.1. Like in the implementation of IR, this hardware operates in parallel in all the output ports of the switch. In the figure, the top half corresponds to the comparison of the destination with the routing interval of IR. The bottom half of the figure corresponds to the hardware required to perform the comparison of the destination with the exclusion interval in order to exclude some nodes from the routing interval. To perform this comparison, the destination address is compared with $ELIB$ and $EUIB$ to know whether the destination is inside the exclusion interval or not. Formally, if $(ELIB \leq EUIB) \wedge (Dest \geq ELIB) \wedge (Dest \leq EUIB)$ or if $(ELIB > EUIB) \wedge ((Dest \geq ELIB) \vee (Dest \leq EUIB))$ then the destination is inside the exclusion interval. Notice that both comparisons (inclusion and exclusion) are performed in parallel, so the delay of the logic for FT^2EI is equal to the one from IR plus the delay of one logic gate. As it can be seen, the hardware ensures that the output port is only selected if the destination is inside the interval indicated by LIB and UIB, and is not inside the interval formed by $ELIB$ and $EUIB$. Formally, an output port of a switch can be selected for routing a packet with destination d only if $d \in [LIB..UIB] \wedge d \notin [ELIB..EUIB]$. Finally, as the exclusion interval only has to be considered if there are faults in the network, a bit is used to enable the use of the exclusion interval (labeled as *Exclusion interval enable* in Figure 3.5). As it can be seen, this implementation of fault-tolerant routing in fat-trees is quite simple. It scales with the network size, since only two sets of registers that store the inclusion and exclusion intervals plus some logic to check whether the destination address is inside these bounds are required per output port[2].

Now that we have the basic infrastructure that allow us to perform the routing algorithm taking into account the exclusion intervals, we have to describe the way to properly compute and update the exclusion intervals after a fault, in order to avoid using those paths that traverse the fault.

3.2.1 Computing the Exclusion Intervals

In order to compute and distribute the routing intervals, we have to define the element that has to start the reconfiguration process. In our proposal, the element that has to detect and take the appropriate actions to tolerate the fault, is the faulty switch. In order to identify this switch, we have to remember that every link in the network goes from the output port of a switch to the input port of a neighbor switch. From these two switches, we refer to *faulty switch* as the switch that is connected to the link that fails through its output port. So, the switch that detects that one of its output ports is faulty is the

[2]In fact, exclusion intervals are only required at up output ports (see section 3.2.1).

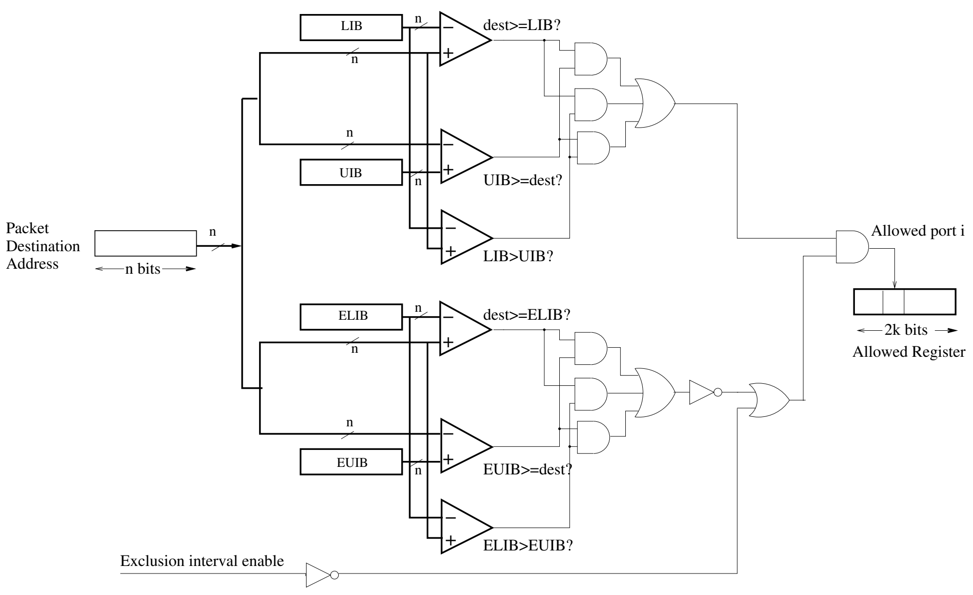


Figure 3.5: FT^2EI hardware associated to each output port.

responsible of starting the reconfiguration process.

The first step of the reconfiguration process consists on classifying the fault. In a k-ary n-tree, we can consider two different types of link faults from the point of view of the switch that will handle the fault: ascending (up) link faults and descending (down) link faults, which correspond to links that forward packets to upper or lower stages, respectively. The management of both types of faults is completely different.

On one hand, the up link fault management is extremely easy. As all nodes reachable through an up link of a switch can also be reached by any of its ascending links, we just nullify the faulty up link. This is done by associating to the faulty link an exclusion interval that includes all the nodes in the network. That is, in the faulty link, we set $ELIB$=0 and $EUIB$=$N-1$, $N = k^n$ being the number of nodes in the network. As the upwards phase is fully-adaptive, all the network traffic that was to be sent through the faulty up link, will be sent through any of the other up links of the switch. For instance, Figure 3.6 presents a 2-ary 3-tree where the switch 4 has a fault in one of its ascending links highlighted in red. To tolerate this fault, switch number 4 has set an exclusion interval that contains all the nodes of the network in the

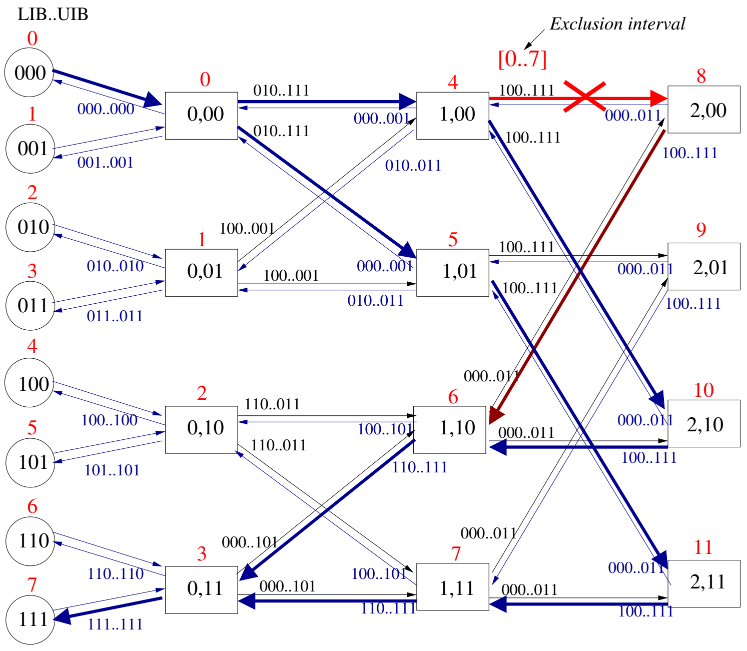


Figure 3.6: Example of ascending fault handling.

output port corresponding to the faulty link. As it can be seen in the figure, if a packet from node 0 whose destination is node 7 arrives to switch 4, it can forward it through the other output port. Moreover, the figure highlights in blue all the healthy paths from node 0 to node 7, and in dark red the links that can not be longer used due to the fault.

On the other hand, a fault in a down link of a switch requires the updating of several exclusion intervals. Moreover, it involves updating the exclusion intervals of switches that are not directly connected to the faulty switch. This is due to the fact that in a k-ary n-tree the down routing phase is deterministic. Once a packet has arrived to one of the common ancestors between source and destination nodes, it has only one possible down path. So, our methodology must avoid that packets arrive to any common ancestor whose down path to destination goes through a faulty link. This requires updating the exclusion intervals of the switches traversed in the ascending routing phase that can deliver the packet to one of these common ancestors.

We will explain how to identify the switches whose exclusion intervals

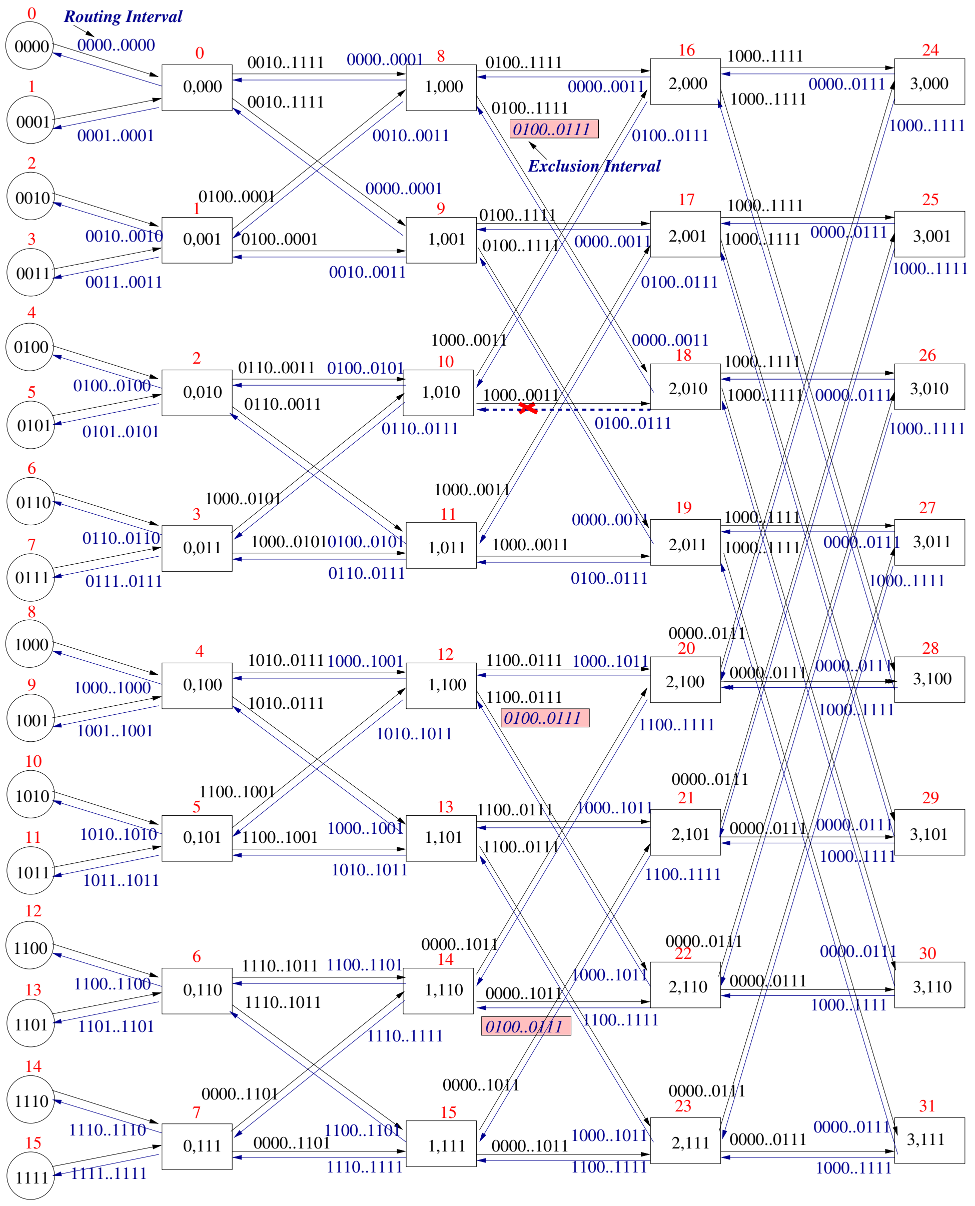


Figure 3.7: A 2-ary 4-tree with a faulty link at switch 18.

should be updated when a descending link fault is detected by using the example shown in Figure 3.7. When a fault appears in a down link of a switch, the nodes that were reachable through the faulty link become unreachable from the switch, since the destinations that are reachable through a down link of a switch cannot be reached through any other link of the switch. Figure 3.7 assumes that link 1 of switch 18 has failed and, as a consequence, nodes 4 to 7 are not longer reachable from switch 18. Therefore, packets with destination 4, 5, 6 or 7 should not arrive to switch 18 in their descending phase. In order to identify the switches where the exclusion intervals should be updated, we travel the tree upwards from switch 18. The first switches that we meet are switches 26 and 30. As it can be seen, switches 26 and 30 should not receive packets destined to nodes 4, 5, 6 and 7, since their unique path to reach these nodes is through switch 18. To make this possible, switches that can directly forward packets to switches 26 and 30 through, should exclude nodes from 4 to 7 in their associated output ports. Therefore, switches[3] 18 and 22 should update the exclusion interval of their up output ports to exclude nodes 4, 5, 6 and 7. Switches 18 and 22 cannot send packets to destinations [4..7] through any of its output ports after updating the exclusion interval, so switches from the previous stage should avoid forwarding packets destined to those nodes to switches 18 and 22. The switches from the previous stage that are directly connected to switches 18 and 22 are switches 8, 12 and 14, which should exclude nodes 4 to 7 in the exclusion interval associated to output port 3, since this is the output port that connect them to either switch 18 or switch 22. Switches 8, 12 and 14 can route packets with these destinations through output port 2. So, it is not necessary to continue the process, since we have reached a set of switches that can forward packets to destinations [4..7] without using the faulty link.

Notice, though, that as long as the exclusion intervals of switches 8, 12 and 14 have been set, it is not longer necessary to update the exclusion intervals of switches located at stage 2 (i.e. switches 18 and 22) since packets destined to nodes 4, 5, 6 and 7 will never reach these switches, because switches 8, 12 and 14 already exclude those nodes in output port 3. Figure 3.7 shows the

[3]Notice that switch 18 will not actually send any packet to nodes 4-7 through its up links.

exclusion intervals that should be updated in the network. Notice that the switches whose exclusion intervals has to be updated (switches 8, 12 and 14) belongs to the previous stage to the one the faulty switch (switch 18) belongs to. This property is always accomplished in the reconfiguration process of descending link faults due to the connection pattern of the fat-tree topology.

Let's check if the faulty link from Figure 3.7 can be avoided by updating to [4..7] the value of the exclusion intervals of output port 3 from switches 8, 12 and 14. For this, we travel along the network from several source nodes to the nodes included in that exclusion interval. First, we focus on nodes included on the exclusion interval. Nodes from 4 to 7 never use the faulty link to send packets among them. It can be seen in the figure that the highest stage that these packets reach is stage 1, concretely these packets can only reach switches 10 or 11. Once these packets arrive to either switch, they are forwarded downwards, so they cannot arrive to switch 18 and use the faulty link.

Now, let's deal with packets from nodes 0 to 3 that are delivered to nodes 4 to 7. As it can be seen, these packets have to reach the stage number 2 in order to begin their downwards routing phase. If any of these packets reaches switch 18, they would have to use the faulty link in their path to the destination, since that is the only path that the switch provides to arrive to nodes from 4 to 7. So, we have to check that switches from the previous stage cannot forward those packets to switch 18. These switches are the switches 8 and 9. Switch 9 forwards packets to nodes from 4 to 7 through switches 17 and 19. From these two switches, those packets would reach switch 11. Thus, packets that reach switch 9 never use the faulty link. On the other hand, switch 8 forwards packets to those destinations through switches 16 and 18. If a packet to those destinations reaches switch 16, it would reach switch 10, and the faulty link would not be used. However, if the packet reaches switch 18, it would use the faulty link. But this is avoided by the exclusion interval set in the output link 3 from switch 8. This exclusion interval forces that all the packets destined to nodes 4 to 7 that reach switch 8 are sent through link 2, thus reaching switch 16.

The same procedure can be followed from packets for nodes 8 to 15 whose destinations are nodes that belong to the affected destinations by the fault. In this case, the exclusion intervals from switches 12 and 14 avoid that packets sent to those destinations reach switch 22, that would forward them through switch 18 and the faulty link.

Next, we show how to compute, in a general case, the switches and links whose exclusion intervals should be updated, and the values to update them, when a new fault appears in the network. Let $\langle e, v_{n-2}, ..., v_1, v_0 \rangle$ and l be the faulty switch and link, respectively. As it can be seen, this switch belongs to the e stage. As shown in the previous example, only some switches at the stage $e-1$ should be updated. The switches to be updated can be classified according to their connection to the faulty switch as directly or indirectly connected to it.

First, we identify the switches that are directly connected to the faulty switch. In Figure 3.7, these switches are 8 and 10 (but switch 10 routes packets to the affected destinations through its down links, so it is not actually necessary to update it). According to Section 2.1.3, the identifier of the switches that are directly connected to the faulty switch can only differ from the faulty switch identifier by the $e-1th$ digit of the second component of their identifiers. Hence, the identifier of these switches is given by $\langle e-1, v_{n-2}..v_e, X, v_{e-2}, .., v_1, v_0 \rangle$, being X any value between 0 and $k-1$. The port to update in them is the ascending one that is connected to the faulty switch and is given by v_{e-1} (the value of the e less significant digit) from the second component of the identifier from the switch that contains the fault. According to our link numbering (where up links start from k), the link to be updated is the $k + v_{e-1}$ one. As it was stated above, there is a switch (switch 10 in the example) from the previous stage that does not need to be updated, despite the fact of accomplishing the previous equation. This switch does not need to be updated because it never forwards through its up links packets whose destination is included in the exclusion interval. This switch is the one that is directly connected to the faulty switch through the link in the opposite direction of the faulty link, and its identifier is given by $\langle e-1, v_{n-2}..v_e, l, v_{e-2}, .., v_1, v_0 \rangle$.

The rest of the switches of the $e-1$ stage to be updated are the ones that are indirectly connected to the faulty switch, since they may forward packets to the excluded destinations through the faulty link following any of the possible paths provided by the routing algorithm. The identifier of these switches is given by $\langle e-1, X, .., X, v_{e-2} ..., v_1, v_0\rangle$. Notice that this expression also includes the switches directly connected to the faulty switch. Again, the link to update in those switches is also given by $k + v_{e-1}$.

Finally, the nodes to include in the exclusion interval should be computed, that is, we have to compute the values that will be stored in the ELIB and EUIB registers of the switches that should be updated. The exclusion interval should contain the nodes included in the routing interval associated to the faulty link, since those nodes are the ones that have become unreachable from the faulty switch. This interval is given by $[v_{n-2}, ..., v_{n-e}, l, 0...0..$ $v_{n-2}, ..., v_{n-e}, l, (k-1)...(k-1)]$.

Let us apply these equations to the example shown in Figure 3.7 in order to check if we get the same results as before. The link 1 of switch 18 $\langle 2, 010\rangle$ has failed. The switches to be updated are the ones whose identifier follows the pattern $\langle 1, XX0\rangle$, that is, switches 8 $\langle 1, 000\rangle$, 10 $\langle 1, 010\rangle$, 12 $\langle 1, 100\rangle$ and 14 $\langle 1, 110\rangle$. The link to be updated in these switches is $k + v_{e-1}$, that is, $k + v_1 = k + 1 =$ link 3. The values to fill in the exclusion intervals are given by [0100..0111], which is the interval that includes nodes 4, 5, 6 and 7. Finally, the switch 10 $\langle 1, 010\rangle$ may not be updated, since it would never forward packets to the excluded destinations through the faulty switch. These are the same switches and intervals that where previously obtained on Figure 3.7.

Notice that in both, up and down link faults, it is only required to associate exclusion intervals to up links, since what we do is to eliminate the ascending paths that cross the faulty link. We modify the paths in the ascending routing phase, no matter if the fault is in an ascending or in a descending link, since this phase is the only one that provides alternative routes of minimal length. Also, notice that only the paths that make use of the faulty link are eliminated, allowing adaptive routing through all the remaining healthy paths. For example, in Figure 3.7 up link 3 from switches 8, 12 and 14 can be adaptively used by any packet whose destination is not in the exclusion interval, since they would not cross the faulty link. As shown, by using one exclusion interval per output link, the network is able to tolerate at least 1 fault (see Section 3.4.2 for more details). Nevertheless, a mechanism that is based on a model in which only one fault can be present in the network is very unrealistic. For this, in the next section, we extend our proposal to a model that considers more than one fault in the network.

3.2.2 Extension to more than one fault

In this section, we analyze the consequences of having more than one fault in the network and we will explain how our methodology deals with them. When considering multiple faults, we have analyzed two alternative approaches. First, we will consider the existence of only one exclusion interval per output port, as assumed up to this point. Second, we consider the possibility of associating multiple exclusion intervals to each ascending output port.

The methodology could be directly applied to tolerate multiple link faults, provided that these faults affect exclusion intervals associated to different output ports. The only problem arises when two different faults require to update the exclusion interval associated to the same output port. In order to tolerate multiple faults regardless of the exclusion intervals to be updated, we have designed a methodology to merge two exclusion intervals, the one previously stored that avoids the previous faults, and the one required to tolerate a new fault. The resulting merged exclusion interval should contain all the nodes contained in both exclusion intervals, but it may also contain some nodes that do not belong to any of the initial intervals. We will refer to these nodes as *victim nodes*. Victim nodes are destination nodes that are actually reachable through an output port, but are included in the exclusion interval of the port due to the process of merging several exclusion intervals into a single interval[4].

The inclusion of victim nodes is necessary if a single exclusion interval is associated to each output port and the old and new intervals do not intersect. As an example, consider the network status shown in Figure 3.7, with a faulty link in switch 18. If, at this point, a new fault is found in link number 1 of switch 22, the interval [12..15] should be excluded in link number 3 of switches 8, 10 and 12. These links already have their exclusion interval registers set to [4..7]. The only possible way of avoiding the use of the faulty links for both set of destinations is to merge both exclusion intervals to obtain a new one

[4]Notice that these nodes are still reachable through other paths, but not using this output port. These nodes are not eliminated from the system.

[4..15], which will include victim nodes from 8 to 11. As the inclusion of victim nodes in the exclusion interval reduces the number of paths in the network to these nodes, care must be taken to minimize the number of victim nodes in the exclusion interval.

The methodology to obtain a single exclusion interval must try to minimize the number of victim nodes taking into account that the exclusion intervals may be cyclic intervals. For this, several cases must be considered when merging two exclusion intervals. First, there is a straight-forward case: when the exclusion intervals that have to be merged overlap. In this case, the resulting interval will merely be the union of the two intervals. Otherwise, either a cyclic or non-cyclic interval can be chosen. The resulting interval is chosen trying to minimize the number of victim nodes. The algorithm that allows doing this is the following:

```
if $[LIB_1..UIB_1] \cap [LIB_2..UIB_2] \neq \emptyset$ then
    $[LIB_{res}..UIB_{res}] = [LIB_1..UIB_1] \cup [LIB_2..UIB_2]$
    break
end if
if $(LIB_1 < UIB_1) \wedge (LIB_2 < UIB_2)$ then
    $LIB_{temp1} = min(LIB_1, LIB_2)$
    $UIB_{temp1} = max(UIB_1, UIB_2)$
    $LIB_{temp2} = max(LIB_1, LIB_2)$
    $UIB_{temp2} = min(UIB_1, UIB_2)$
    if $|[LIB_{temp1}..UIB_{temp1}]| \leq |[LIB_{temp2}..UIB_{temp2}]|$ then
        $[LIB_{res}..UIB_{res}] = [LIB_{temp1}..UIB_{temp1}]$
    else
        $[LIB_{res}..UIB_{res}] = [LIB_{temp2}..UIB_{temp2}]$
    end if
    break
end if
$LIB_{temp1} = min(LIB_1, LIB_2)$
$UIB_{temp1} = min(UIB_1, UIB_2)$
$LIB_{temp2} = max(LIB_1, LIB_2)$
$UIB_{temp2} = max(UIB_1, UIB_2)$
if $|[LIB_{temp1}..UIB_{temp1}]| \leq |[LIB_{temp2}..UIB_{temp2}]|$ then
```

```
        $[LIB_{res}..UIB_{res}] = [LIB_{temp1}..UIB_{temp1}]$
    **else**
        $[LIB_{res}..UIB_{res}] = [LIB_{temp2}..UIB_{temp2}]$
    **end if**
```

In the previous pseudo-code, *break* instructions stop the algorithm. First, the algorithm checks whether the exclusion intervals to merge are disjoint or not. In case they are not disjoint, the resulting exclusion interval is the union of both intervals and the algorithm ends. In case that the exclusion intervals are disjoint, it checks if they are both non-cyclic intervals. In that case, the algorithm must construct an exclusion interval that includes both intervals, but it can be done in two ways, so the algorithm builds two auxiliary intervals. The first one is built by using a non-cyclic interval, and the second one is built by using a cyclic interval. Finally, the algorithm decides to choose one of both auxiliary intervals by comparing the number of elements of each interval. The one that has less number of elements would be chosen, and the algorithm ends. Notice that the resulting merged intervals contain all the nodes from the initial intervals by definition. So, the minimum number of elements in the resulting interval is equal to the number of different nodes included in the initial intervals, thus any additional node included in the resulting interval is a victim node. Therefore, by minimizing the number of elements on the resulting interval, we choose the option that introduces the lowest possible number of victim nodes.

The last case is only reached when the exclusion intervals are disjoint and one of them is a cyclic interval. In this case, the algorithm follows a similar procedure to the one used in the previous case, but changing the way to build the auxiliary intervals. Notice that if both exclusion intervals are cyclic, they are not disjoint and the algorithm would enter in the first case.

Furthermore, after updating each exclusion interval (either by merging with a previous one or by setting a new one), it must be checked if there is a set of nodes that are now unreachable from the updated switch. These nodes were reachable through some of its up links before updating the exclusion interval, but due to multiple link faults they may have been excluded in all the up links of the switch, and, therefore, the switch can not reach this set of nodes. If this is the case, the switches of the previous stage that connect to the affected switch should also exclude that set of nodes in the links that connect to it, in order to avoid sending to it packets whose destinations cannot be reached by crossing the affected switch. Once the switches from the previous stage are updated, they must perform the same checking, and so on. This process is iterative, and it stops if the propagation of the exclusion interval reaches the lowest stage of the network, or if the switch detects that it can reach all the destinations of the network. For instance, using the scenario shown in Figure 3.7, if a new fault is detected at link 1 of switch 16, link 2 of switches 8, 12 and 14 should exclude the interval [4..7]. Now, these switches would have the interval [4..7] excluded in all their up links, hence switches of the previous stage directly connected to them should not forward packets with destination between 4 and 7 to them. So, this interval should be also excluded at link number 2 of switches 0, 1, 4, 5, 6 and 7. No more iterations are needed, because these switches have other ports that can forward packets to nodes [4..7]. Anyway, once the lowest stage of the network is reached, no more iterations are possible.

The task of minimizing the number of victim nodes in a network with several faults can be improved if more than one exclusion interval could be associated to each output port. In this way, if merging two intervals introduces some victim nodes, we could instead store them into two different physical exclusion intervals associated to the same output port, thus, not excluding any victim node. The routing function will now exclude those nodes that are included in any of the exclusion intervals associated to the output port. So, no healthy path is sacrificed and therefore we should get a better performance. However, if we consider the possibility of having a higher number of faults than the number of physical exclusion intervals associated to each output port, victim nodes could be again included in the exclusion intervals associated to output ports, but it is expected that the number of victim nodes will be smaller than the one obtained with just one exclusion interval per output port. In order to fill exclusion intervals in the output ports, the methodology follows these steps when after a fault, a new exclusion interval must be associated to an output port:

- Check if the exclusion interval associated to the new fault intersects with any of the exclusion intervals already stored at the output port. If so, a new exclusion interval is obtained by merging them. In this case, it is not necessary to use another physical exclusion interval, as there are no victim nodes.

- If the new exclusion interval does not intersect with any of the intervals already stored in the output port, and there are physical exclusion intervals available, then a new one is used to store the exclusion interval associated to the new fault.

- If there are not physical exclusion intervals available to assign to the new non-overlapping one, it is merged with the exclusion interval that leads to the minimum number of victim nodes following the algorithm presented above.

As stated in the case with only a physical exclusion interval per output port, once a physical exclusion interval has been updated at a switch, it must be checked if there are nodes that have become unreachable from that switch, accordingly updating the exclusion intervals in the previous stage. This process should be repeated as many times as required.

Notice that the exclusion intervals do not induce new dependencies in the channel dependency graph (CDG) associated to the routing function [40]. In fact, every time a exclusion interval is updated, the number of channel dependencies is reduced, as the number of possible paths is reduced. Therefore, the resulting fault-tolerant routing algorithm is deadlock free, since the routing algorithm in which it is based on is deadlock-free and our methodology removes some existing channel dependencies.

3.3 Dynamic Fault-tolerance Routing with Exclusion intervals

In this section, a mechanism to dynamically spread the fault-tolerance information in the network without stopping its activity is going to be presented. Specifically, we enhance the fault-tolerant routing strategy presented in the previous section to support a dynamic fault model. With a static fault-model, we did not care about how the information is distributed in order to reconfigure the network after a fault, we just identified the switches and links that must be updated after a fault, and the information to update them. In this section, we take on the challenge of designing the dynamic version of the methodology to reach the adequate switches and deliver them the fault-tolerance information.

Since a dynamic model is used, when a new fault is detected, the system continues running the applications, and the mechanism that handles the fault progresses simultaneously in the network with the traffic from the applications. In this way, a dynamic fault model implementation of the methodology avoids the disadvantages of the static fault model. These disadvantages are the need of stopping the system while the reconfiguration is in progress, and the need of checkpointing techniques. Checkpointing is undesirable because it usually requires great quantities of system resources, such as processor cycles, network bandwidth, storage capacity and so on. Moreover, applications are stopped while reconfiguration is in progress, and later they are restarted from the last performed checkpoint. Depending on checkpointing frequency, a lot of computing made by these applications may be lost.

3.3.1 Informal Description

We are going to intuitively describe how the exclusion intervals can be dynamically updated by using the same example used in the previous section (see Figure 3.7). Remember that a fault at link 1 of switch 18 has appeared, so we must avoid packets to take the descending paths that traverse it. We assume that a switch has the capability of realizing that one of its links has failed as in [124]. So, in our methodology, switch 18 begins the exclusion interval update, since the responsible of starting the reconfiguration process is the switch that detects the fault. Switch 18 triggers the mechanism by notifying either switch 26 or 30 that they must prevent sending it packets destined to nodes 4 to 7. These are the switches of the upper stage that are connected directly to the faulty switch. Since they can not route those packets through other switches, they indicate to every switch of the previous stage connected to them that they can no longer continue routing packets whose destination is between 4 and 7. Notice that it does not matter which switch is chosen between switch 16 and switch 20, since they are connected to the same switches, switches 18 and 22 in this case.

Switches 26 and 30 will notify switches 18 and 22, that they cannot forward packets destined to nodes 4 to 7. Switches 18 and 22 can not route those packets through any of its output ports, so they must notify switches of the previous stage[5] that they must avoid sending packets with those destinations to them. These switches are 8, 12 and 14, which already have an alternative path to reach the conflictive destinations. So, these switches must update the exclusion interval of the output port corresponding to the input port through which the notification has arrived. By doing that, no packets destined from 4 to 7 will arrive to the switches that are not able to reach those destinations. Switches 8, 12 and 14 will route packets to these destinations through the other up link, as these destinations are included in their inclusion intervals. These switches will not longer propagate the notification to the previous stage. Notice that these switches are the same switches where configuration was done in the previous section when using the static fault model. Furthermore, the link whose exclusion interval must be updated is the same as in the previous section.

3.3.2 Formal Description

Now, we make a more formal description of our dynamic fault-tolerant mechanism. As stated above, we assume that the switches of the network have the capability to detect link failures. In addition, we assume that switches have the capability of generating fault-tolerance control packets and in case they receive one of these packets, they are also able to interpret it and take the appropriate actions. In order to spread the reconfiguration among the required switches, we define a new control packet that includes all the required information to identify the switches at where reconfiguration must be done, the exclusion intervals of which output port should be updated, and the value to update them. We refer to this kind of packets as fault-tolerance control packet and they are formed by three fields that contain the required information for the reconfiguration process: $FaultySwitch$, $ELIB$ and $EUIB$. In the $FaultySwitch$ field, the identifier of the switch that detects the faulty link is indicated, and in $ELIB$ and $EUIB$, the interval to exclude is indicated. As it

[5]Notice that, as switch 18 is the one that triggers the reconfiguration process, it will not wait for upper switches to propagate the reconfiguration downwards.

was shown in the static fault-model mechanism, to identify where to perform the reconfiguration we do not need more information than the one provided by these fields.

As commented, the dynamic mechanism begins when a switch detects that one of its output links has failed. If it is an up link, its exclusion interval corresponding to that up link is set to $[0..N-1]$ and no more actions are needed to tolerate the fault. If the faulty link is a down link, the switch generates a fault-tolerance control packet. The interval to exclude corresponds to the inclusion interval of the link that has failed. In other words, the switch fills the fields $ELIB$ and $EUIB$ from the control packet with the routing interval of the faulty link. The switch also fills the $FaultySwitch$ field with its own identifier. As observed in Figure 3.7, the fault-tolerance control packet must arrive to the upper stage of the network, and then, it has to be forwarded downwards to all the switches that are reachable by using only down links till the previous stage to the one at which the fault is located. To do that, it is enough that the switch that has detected a fault in one of its down links generates a fault tolerance control packet that is sent only through one of its up links. The choice of a given up link does not have any impact on the result[6]. While the control packet is on its upwards routing path, the switch that receives it, at each stage, must forward it through any of their healthy up links. It is enough to reach one of the switches of the last stage in order to reach all the switches that must be updated in the previous stage to the one where the fault is located. Once the packet reaches the last stage of the fat-tree, it begins its downwards path. In this down path, all switches that receive a control packet must re-send it through all their down links, stopping the process at the previous stage to the one where the fault was detected. The stage at which the fault has been detected can be easily obtained from the faulty switch identifier indicated in the control packet. The switches of this stage that receive a fault-tolerance control packet will exclude the interval indicated in the control packet in the up output port that corresponds to the down input port through which the packet is received. Moreover, they could obtain it by using the identifier of the faulty switch.

[6]In Section 3.3.3, we explain that this is not always true when multiple faults are considered.

Therefore, the algorithm works as follows:

- A switch that detects a fault in some of its output links:
  - If it is an up link, its exclusion interval is set to $[0..N-1]$.
  - If it is a down link, it generates a control packet containing its identifier and the inclusion interval of the faulty link.
    1. If the switch is in the last stage, it sends the control packet through all its down links.
    2. If the switch is not in the last stage, it sends the control packet through one of its up links, it does not matter which one is selected.
- A switch that receives a fault-tolerance control packet:
  - If it arrives in the upwards direction through one of its input up links:
    1. If the switch is not in the last stage, it re-sends the control packet through just one of its up output links. It does not matter which one is selected.
    2. If the switch is in the last stage, it re-sends the control packet through all its down output links.
  - If the packet arrives in the downwards direction through one of its input links:
    1. If the switch is located at a stage different to the previous one of the switch indicated in the packet (the faulty switch), then it re-sends the control packet through all its down output links.
    2. If the switch is in the previous stage of the switch indicated in the control packet, then it updates the exclusion interval of the up link corresponding to the input link through which the packet is received with the interval indicated in the control packet.

Notice that fault-tolerance control packets must be received completely in their downwards phase before routing them since the switch decides to forward it or to update its intervals depending on the value of the fields of this packets. Hence, independently of the switching technique used for the rest of the packets, fault-tolerance control packets have to use store and forward switching. By following this algorithm, all the remaining routes are fault-free. This mechanism is designed for tolerating one fault. In the following section, we present how to adapt it to tolerate multiple faults.

3.3.3 Multiple Faults Considerations

As in the static fault-model, when multiple faults are considered, it is possible that several faults provoke to update the exclusion interval of the same output port of the same switch. To deal with this issue, the techniques presented in Section 3.2.2 to merge multiple intervals into one or to implement several exclusion intervals per output port can be used in the dynamic model without any modification, and providing the same fault-tolerance characteristics than in the static fault-model.

Also, as in the static fault-model, every time an exclusion interval is updated, the switch must check if it can still forward packets to all destinations. In case that the switch can not reach some set of nodes, it should notify all the switches of the previous stage connected to it to prevent that packets to the unreachable destinations are forwarded to the switch. Dealing with this case is easy with the dynamic fault model by considering that a new fault have happened and it affects to the up links of the switch. Thus, the affected switch generates a fault-tolerance control packet to update the switches of the previous stage connected to it. The packet should contain as faulty switch the current one and as exclusion interval the set of nodes that it can no longer reach. As a result, the switches from the previous stage directly connected to the switch will exclude that interval in the up link that connect them to the switch that sent the control packet, and they should also check if there is a set of unreachable nodes. This process must be repeated as many times as needed (see Section 3.2.2 for more details).

Despite that multiple faults can be handled in a similar way to the one used for the static fault-model mechanism, the dynamic mechanism requires a temporal restriction. Our dynamic mechanism can only guarantee the correctness of the reconfiguration process provided that a new fault does not appear before the reconfiguration of a previous fault has been completed. This can be easily understood by the example presented in Figure 3.8 which represents an incorrect reconfiguration process caused by the simultaneous appearance of two faults in the network. In Figure 3.8(a), we can observe that two faults have been detected simultaneously: one in the link number 1 of switch 6, and the another one in the link number 0 of switch 8. The destinations that have become unreachable due to these faults are nodes 6 and 7 in switch 6, and nodes from 0 to 3 in switch 8. Therefore, as shown in Figure 3.8(b), these switches generate a fault-tolerance control packet with the corresponding exclusion interval following the algorithm presented above. In this example, both switches send the fault-tolerance control packet to each other. Finally, Figure 3.8(c) shows that switch 6 receives the fault-tolerance control packet from switch 8, and updates the exclusion interval of the link that connects it to switch 8. Therefore, switch 6 will not send packets to nodes 0 to 3 through by crossing switch 8. The reconfiguration for the fault detected at switch 8 is completed correctly. Nevertheless, this does not happen with the reconfiguration for the fault detected at switch 6. Switch 8 receives the fault-tolerance control packet from switch 6. As it belongs to the last stage of the network, it spread the fault-tolerance control packet by forwarding it by its down links. However, its link number 0 is faulty, and the fault-tolerance control packet is lost without reconfiguring the exclusion intervals of the required switches. Therefore, packets whose destinations are 6 or 7 may reach switch 6 and would be lost.

As it can be seen, our mechanism cannot ensure that the fault-tolerance control packets update the required exclusion intervals if the reconfiguration due to another fault is still in process. To avoid this problem, the mechanism requires that the mean time between faults (MTBF) of the system is several orders of magnitude smaller than the time required to complete the reconfiguration process. This assumption is very realistic since fault handling time is several orders of magnitude smaller than real systems MTBF[7] (see Section 3.4.3).

[7]According to [126] the network failure rate is 0.129 failures per week.

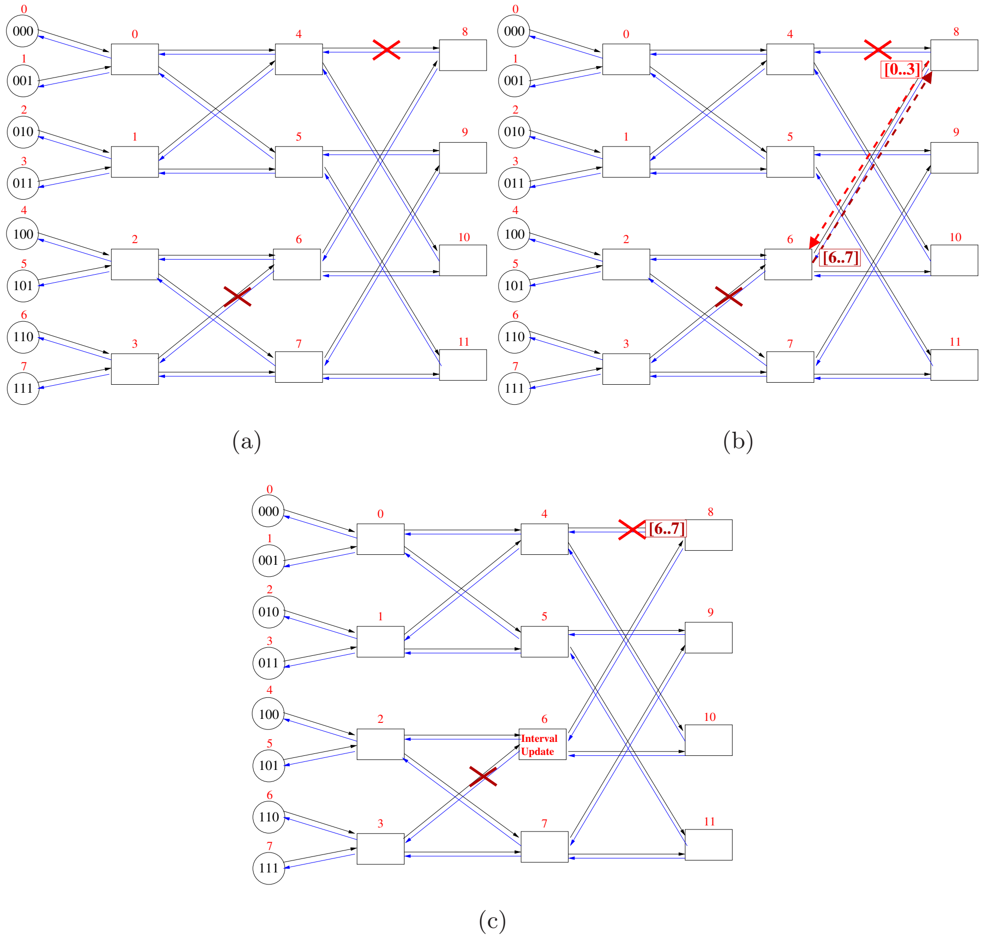


Figure 3.8: Step-by-step example of an incorrect reconfiguration due to two simultaneous faults.

Now, we want to show that our dynamic mechanism can ensure the correctness of the reconfiguration process even in the presence of previous faults if the reconfiguration process due to these faults has ended completely. Figure 3.9(a) presents the initial scenario, in which switch 8 has detected a fault in its link 0 and the reconfiguration process has ended by excluding the interval of nodes [0..3] from port number 2 of switch 6. Later, switch 6 detects a new fault in its link number 1 (Figure 3.9(b)). So, switch 6 starts the reconfiguration process by generating a fault-tolerance control packet and tries to forward it through its up links. At this point, switch 6 knows that through its link 2, the nodes 0 to 3 are excluded due to a previous fault, so it decides to avoid that fault by sending the fault-tolerance control packet through its link number 3, as it can be seen in Figure 3.9(c). Once the fault-tolerance control packet has arrived to switch 10, it continues with the reconfiguration without any problem, as shown in Figure 3.9(d).

However, in order to perform the reconfiguration properly, switch 6 has to be able to decide not to use the link with the exclusion interval. The algorithm to spread the fault-tolerance packets has to be adjusted for the case of multiple faults, since fault-tolerance control packets must avoid the different faulty links in order to complete the reconfiguration process. Now, a switch that must forward a fault-tolerance control packet during its upwards phase has to guarantee that the packet is able to reach all the destination nodes of the network. For this, when forwarding a fault-tolerance control packet, a switch tries to use an up link whose exclusion interval registers are not used. In case that the switch has several links whose exclusion intervals are not being used, the switch can choose any of the up links to forwards the control packet. If the switch does not have any up link whose exclusion intervals are disabled, then the switch has to find a subset of up links that allows the control packet to reach all the destinations. For example, if a switch with arity 2 whose two up links have exclusion intervals set to $[0..(N/2)-1]$ and to $[N/2..N-1]$, respectively, the switch would have to forward the fault-tolerance control packet by both up links to reach all the destinations of the network. Notice that, if this is not possible, then the switch has a subset of destinations that are not reachable from it. In other words, the switch has lost the connectivity with those destinations and therefore the fault combination

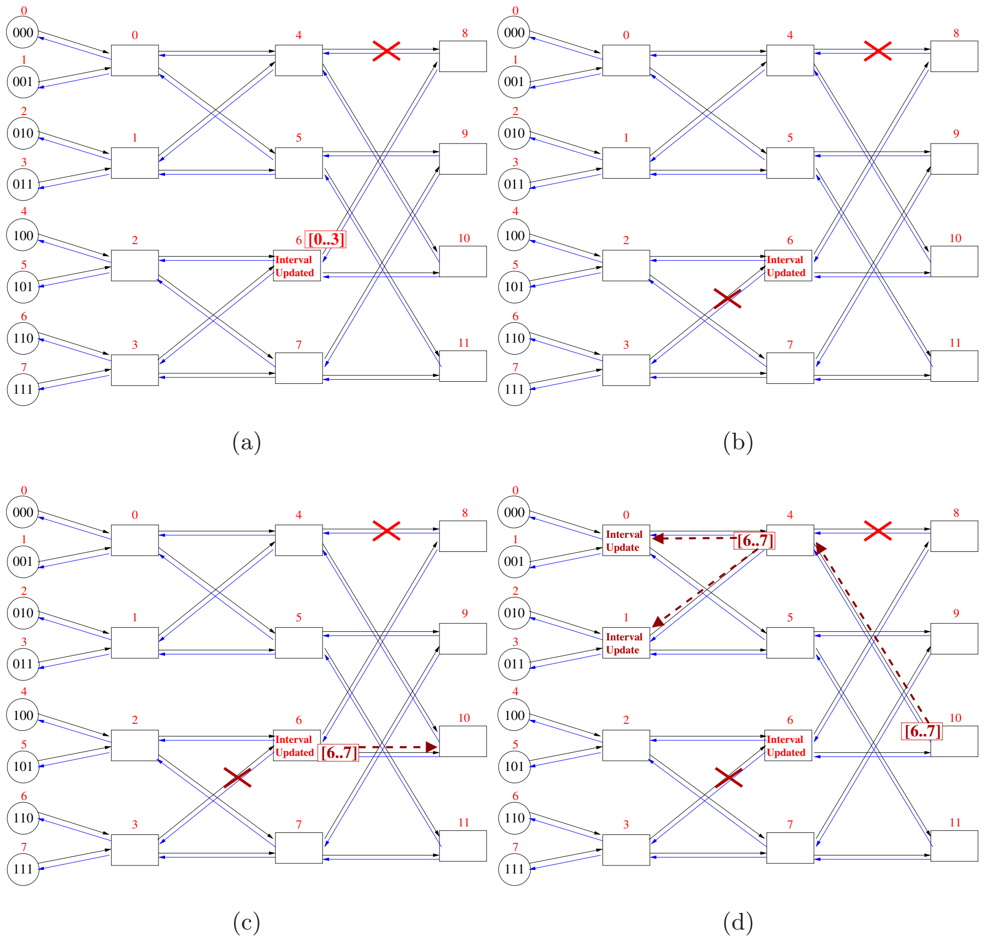


Figure 3.9: Step-by-step example of the correct reconfiguration of a new fault despite the presence of previous faults.

is not tolerated by FT^2EI, so reconfiguration can not be completed. Notice that if the switch has an exclusion interval set in all its up links, then the network has suffered at least k faults, and our strategy can not tolerate such a number of faults either on a static or a dynamic fault-model. More details about these two issues are described on Section 3.4.2.

Additionally, instead of considering the bidirectional links as two separate links in opposite directions, we assume that when a fault occurs in a link, this link can not be used in any of its two directions and the fault is detected as an ascending link fault by the switch of the lower stage and as a descending link fault by the switch from the higher stage. This assumption was used in [61] and it is a more realistic one, as bidirectional links are actually implemented by using a single wire [33, 40]. In this way, we allow the mechanism to complete the reconfiguration even when several down link faults appear in the same sub-tree.

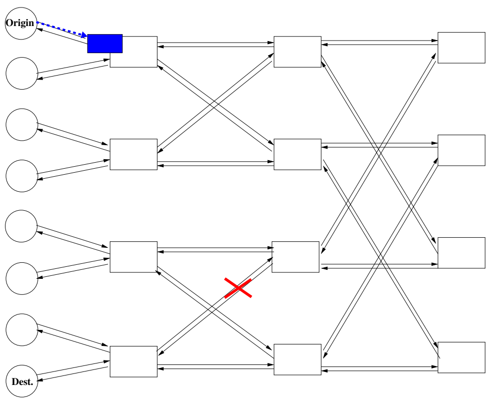
3.3.4 Avoiding Losing Packets during Reconfiguration

Despite that the mechanism is capable of reconfiguring the fault-tolerance information of the network avoiding to use the faulty links, thus preventing that packets are lost once the exclusion intervals have been updated, it can happen that some packets are lost during the reconfiguration of the network, since the computer activity is not stopped and the reconfiguration is still on progress. The packets that are lost are directed to the destinations contained in the exclusion interval and cross the output links where the exclusion intervals should be updated.

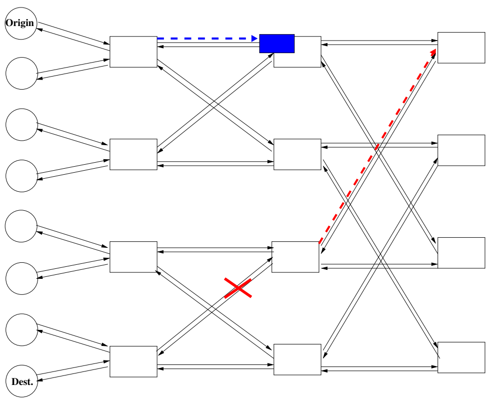
This can be more easily understood with the example shown in Figure 3.10, which represents how a packet can be lost during reconfiguration. In Figure 3.10(a), the node 0 sends a packet to the node 7. Concurrently, switch 6 detects that its link 1 is faulty. Later, as presented in Figure 3.10(b), switch 0 has routed the packet from node 0 and forwards it to the next switch of its route. At the same time, switch 6 has started the reconfiguration process and sends upwards the fault-tolerance control packet. Next, as presented in Figure 3.10(c), the packet from node 0 reaches the last stage of the network and starts its deterministic downwards phase. The reconfiguration process progress simultaneously. Nevertheless, notice that the reconfiguration for the packet sent from node 0 has arrived too late, since the packet would be forced to cross the faulty link, as it can be seen in Figure 3.10(d). In this figure, we can see that the reconfiguration process ends correctly, but the packet from node 0 reaches the faulty switch, and the only path to the destination of the packet is through the faulty switch, as shown in Figure 3.10(e).

To deal with these packets, there are two alternatives. In the first approach, no action is taken. So, these packets will be lost, and should be recovered by a higher level protocol. Notice that fault-tolerance control packets should have a higher priority than data packets to speed-up reconfiguration and minimize the number of lost packets during reconfiguration. For the priority, we have used a non-preemptive priority mechanism. That is, fault-tolerance control packet have a higher priority than data packets when selecting an output port. However, fault-tolerance control packets cannot use a port that is being used by another packet, despite of the priority of the packet. For more details, in section 3.4.3, we analyze the effect of such traffic priorization on the time that takes to complete the reconfiguration.

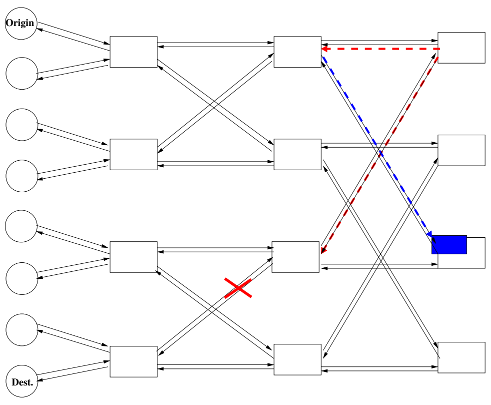
As a better approach, we propose the use of an emergency path. An emergency path can be used when a switch detects that the routing function returns as unique output link a faulty link. With this technique, when a packet is going to traverse a faulty link in a switch, the packet is deviated to any of the other non-faulty down links of the switch. Notice that the chosen down link will not include the destination of the packet in its routing interval. Once in the next switch, the packet is forwarded one stage up by the Interval Routing algorithm, and takes another down path to its destination, thus avoiding the failure. In this way, when a switch is no longer able to reach the destination of the packet, it provides a non-minimal path to the destination of the packet. An example is presented in Figure 3.11. This figure provides an alternative ending to the figures shown in Figure 3.10. Concretely, it starts from the scenario presented in Figure 3.10(d), in which the reconfiguration is completed, but the packet from node 0 has reached the faulty switch, and it is going to cross the faulty link to reach its destination. Nevertheless, by using an emergency path, the switch deviates the packet through its healthy down link as shown in Figure 3.11(a). Once in the first switch from the emergency path, the packet is routed normally by the routing intervals to its destination



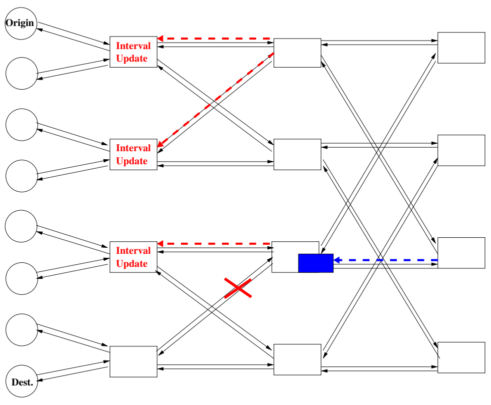
(a)



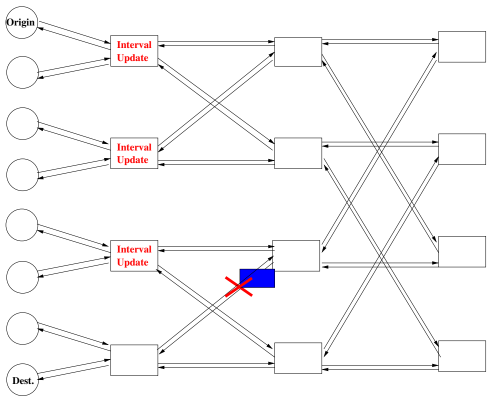
(b)



(c)



(d)



(e)

Figure 3.10: Step-by-step example of packet lost during reconfiguration.

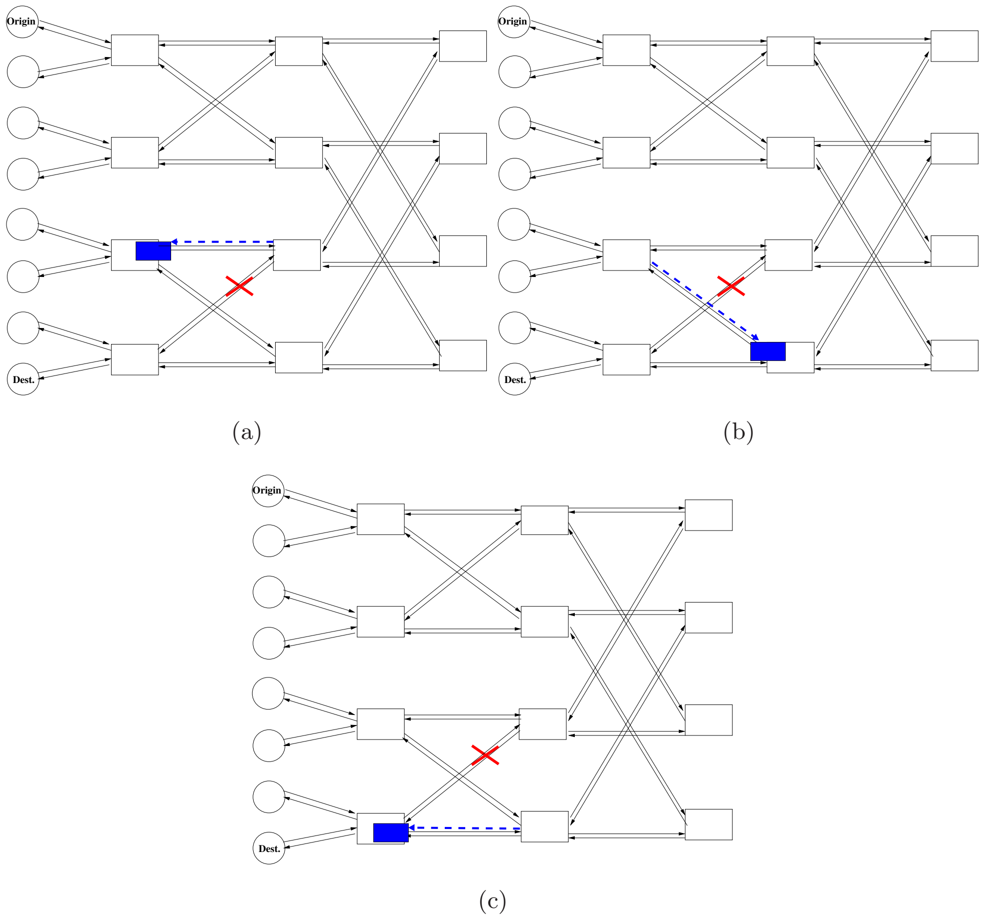


Figure 3.11: Step-by-step example of a deviated packet through an emergency path.

as shown in Figures 3.11(b) and 3.11(c).

The deviated packets will go through a non-minimal path; but with this simple technique we can prevent the loss of any packet while the reconfiguration is in progress. Although the use of a down link followed by an ascending one introduces a new channel dependency, it does not introduce a cycle in the dependency graph, since in order to introduce a cyclic dependency, the faulty link is needed. Therefore, the mechanism of the emergency path is deadlock-free.

A similar mechanism to the emergency path was introduced in [115]. In that paper, the authors proposed to misroute packets down several hops as unique fault-tolerant mechanism. That mechanism degrades throughput considerably and increases latency due to the use of non-minimal paths continuously after a fault appears. By using our emergency path, the paths of the packets that should cross the faulty link are increased in two hops, which increases their latency, but allow them to reach their destination. Nevertheless, we do not consider it a good permanent strategy to deal with faults. We only apply it temporarily, between the fault detection and the end of the reconfiguration period, which it is a short period of time, as it will be shown. Notice that, when reconfiguration is completed, the packets that would use the emergency paths if they reach the faulty switch are deviated through other switches thanks to the exclusion intervals.

3.4 Evaluation

In this section, we evaluate FT^2EI. First, we focus on the fault–tolerance properties of the proposal. Then we focus on some dynamic issues, such as reconfiguration time. Next, we study the performance degradation of the network, in both latency and throughput, when faults have been handled and the exclusion intervals have been updated. Finally, we show a comparison of the amount of memory required by our proposal against the memory required by a system based on forwarding tables.

Before moving to the evaluation, we want to remark several important concepts that we use during this section. We will say that the proposal is $n-$fault tolerant, if it is able to tolerate any combination of n faults. Indeed, we say that a given combination of n faults is tolerated if the methodology is able to provide at least one path to communicate every source-destination pair in the network completely avoiding the faults. A fault combination is not tolerated, if there is at least a source-destination pair for which the methodology can provide a path that traverses a faulty link. When possible, we have analyzed all the possible combinations for a given number of faults. This has been possible for 2-ary 2-trees and 2-ary 3-trees. For networks with a larger number of links, we have analyzed up to 10,000 randomly selected fault combinations. The error caused due to not analyzing all the cases is obtained by a statistical method that considers the total number of fault combinations and the number of analyzed ones, and represents the maximum probability that a fault combination is not tolerated. This probability is quite low (i.e. fault combinations are likely to be tolerated). As an example, in the 4-ary 3-tree, this probability is 0.0081 for 2 faults and 0.0098 for 15 faults.

3.4.1 Simulation Environment

For the fault–tolerance evaluation, we have developed an application that generates all or up to 10000 random fault combinations depending on the network under testing. For each fault combination, it checks if our proposal can provide at least a fault-free path for each source-destination pair for each fault combination. On the other hand, the dynamic issues that we want to analyze are mainly the reconfiguration time, that is, how long it takes to update all the exclusion intervals from the fault detection, and the number of packets lost during the reconfiguration period. To evaluate these dynamic issues and the performance of the network before and after reconfiguration, a detailed event-driven simulator has been developed.

The simulator models a k-ary n-tree virtual cut-through network with point-to-point bidirectional serial links. Routers has a non-multiplexed crossbar with queues only at the input ports. The queues can store up to five packets. Packets are adaptively routed using the mechanism proposed in this paper. Routing, switching and link time are assumed to be 1 cycle. We assume that the packet generation rate is constant and the same for all the nodes. The destination of a message is randomly chosen with the same probability for all the nodes. The packet length is set to 16 bytes. We assume that the time needed by a switch to detect that one of its links has failed is equal to 10 cycles. The time to generate and process a fault-tolerance control packet by a switch is 1 cycle. The same value is used for updating the exclusion intervals of an up link. Finally, the simulator ensures that each node of the network receives an average of 100000 packets before ending each simulation to ensure the stability of the results.

By using these tools, we have simulated a wide range of fat-trees for different network loads. Concretely, we have evaluated from 2-ary 3-tree to 2-ary 6-tree, from 4-ary 3-tree to 4-ary 6-tree, and from 8-ary 3-tree to 8-ary 5-tree.

The results for the topologies that are not shown follow the same trend than the ones presented in here, but we only present the most representative ones for the sake of simplicity.

3.4.2 Fault-Tolerance Results

In order to analyze the fault-tolerance capabilities of our mechanism, we analyze, for each network, the routes provided by FT^2EI after provoking several combinations of randomly-chosen faults. As commented previously, for each network and for each number of faults, we have analyzed 10000 fault combinations. First, we analyze the capabilities of our mechanism by using only one exclusion interval per output port. Next, the advantages of using multiple exclusion intervals per output port are analyzed.

One exclusion interval

Table 3.1 summarizes the maximum number of tolerated faults with a single exclusion interval per output port for the analyzed networks. As it can be observed, the number of tolerated faults is $k-1$. By using FT^2EI, the networks whose switches have $k = 2$ tolerate 1 fault, the networks whose switches have $k = 8$ tolerate up to 7 faults, and so on. As commented previously, notice that $k - 1$ is the maximum number of faults that can be physically tolerated in the network without adding extra resources to the network. For example, in Figure 3.12, we show two examples of a combination of two faults that breaks the connectivity of a 2-ary 3-tree ($k = 2$). For example, Figure 3.12(a) shows that if the k faults affect to the k up links of a switch of the first stage, then the nodes attached to this switch will be isolated and the fault combination is not tolerated. Moreover, notice that it is not needed that the faults affect to all the up links of a switch from the first stage to lost the network connectivity. For instance, in Figure 3.12(b), connectivity is lost since the faults cut all the paths that go from node A to nodes B and C, but affecting links that belong to different switches. As it can be seen, k faults can be enough to break the connectivity of the network, and therefore, more than $k - 1$ faults cannot be tolerated without adding new resources to the network.

Concretely, the non-tolerated fault combinations for more than $k-1$ faults

Table 3.1: Number of tolerated faults for different k-ary n-trees.

Network	# of tolerated faults
2-ary 2-tree	1
2-ary 3-tree	1
2-ary 4-tree	1
3-ary 3-tree	2
3-ary 4-tree	2
4-ary 2-tree	3
4-ary 3-tree	3
8-ary 3-tree	7
8-ary 4-tree	7
8-ary 5-tree	7

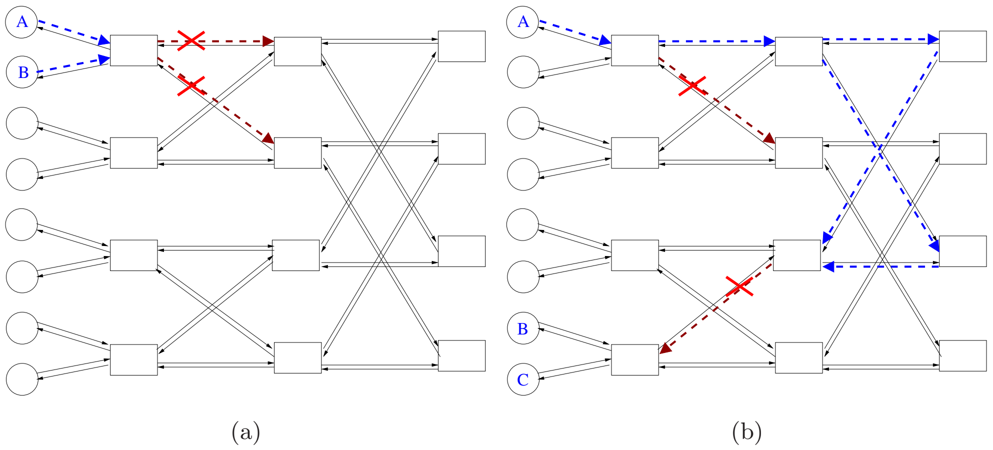


Figure 3.12: 2-ary 3-tree with two faults that break the network connectivity.

in our methodology correspond to the fault combinations where there is at least a switch at the first stage (stage 0) whose up links exclude a common subset of destination nodes. In this way, source nodes attached to that switch have no available path to the excluded set of nodes. When this happens in higher stages different to stage 0, it is solved by propagating the exclusion interval to the switches of the previous stage. However, this is not possible for stage 0. Next, we show that this case is the only one that our mechanism cannot tolerate for combinations of k faults.

As can be seen in Section 3.2, when a single fault occurs in the network the switches that have to update its exclusion intervals only have to update the one associated to one of its up output ports. Therefore, the other $k-1$ up output ports of these switches can still forward packets to the destination nodes included in the exclusion interval. If one fault can only affect to one output port of a switch, $k-1$ faults would at most affect to $k-1$ up output ports of a switch, that is, $k-1$ output ports of the switch would have to set their exclusion intervals. In the worst case, the $k-1$ exclusion intervals of a given switch may share a set of nodes, but even in that case, there is still an up output port that does not exclude any node. Therefore, the switch still can forward packets to all destinations and, therefore, the combination of faults is tolerated. On the other hand, if we consider combinations of k faults (or more), two scenarios can be identified:

- The k faults have affected a subset, but not all, of the up output ports of a given switch. This will not be a problem for our methodology. The exclusion interval of these output ports will be updated, but the switch can still forward packets destined to the nodes affected by the exclusion intervals through any of the other up output ports of the switch that have not excluded them due to the faults.

- The k faults have affected all the k up output ports of a switch. In this case, there are two possible sub-cases:

  – The k exclusion intervals that have been set in the switch do not have any common node. Despite the fact that some destinations cannot be reached through some of the up output ports, for each destination there exists at least an up output port at which the destination is not excluded.

– The k exclusion intervals that have been set in the switch have some common nodes. In this case, the switch cannot reach these destinations through any of its up output ports. In this situation, FT^2EI tries to tolerate the fault by spreading the exclusion information to the switches of the previous stage that are connected to the affected switch. This is possible for all the stages, but not for the switches of the lowest stage (stage 0) of the network. Therefore, some subset of nodes become unreachable from the switch, and the k faults are not tolerated.

Therefore, despite that the mechanism cannot tolerate more than $k - 1$ faults, it can tolerate certain combination of k faults or more. Indeed, Figure 3.13 shows the percentage of non-tolerated fault combinations for two network sizes (2-ary 3-tree and 4-ary 3-tree) when varying the number of faults. As it can be seen, in a 2-ary 3-tree, all the combinations of 1 faults are tolerated, and for 2 faults there is a 6% of non-tolerated combinations. In a 4-ary 3-tree, all the combinations of 3 faults are tolerated, and for 4 faults, there is only a 0.23% of non-tolerated combinations. Notice that, for more than $k - 1$ faults, the percentage of non-tolerated combinations decreases as we increase the network size. If the fat-tree has more stages (n), there is a larger percentage of fault combinations that can be tolerated by propagating to the previous stages the exclusion interval information before arriving to stage 0. If the switch arity (k) is increased, a larger number of faults are tolerated by design. Nevertheless, as the network size increases, either by having more stages or a higher switch arity, more possible paths are available in the network, so it is easier to provide alternative paths between a source-destination pair no matter the number of faults.

Despite that this analysis have been performed for a static model, the same degree of fault-tolerance can be achieved when considering a dynamic fault–model. However, it must be taken into account that reconfiguration of each fault must be finished before the appearance of the following one, as stated on Section 3.3.3.

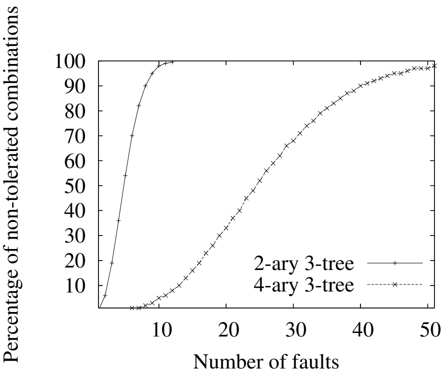


Figure 3.13: Percentage of non-tolerated combinations versus number of faults.

Multiple exclusion intervals

The fact of having multiple exclusion intervals per output port does not increase the number of tolerated faults as can be seen in Table 3.2. Table 3.2 shows the number of tolerated faults for the same networks presented in Table 3.1 varying the number of exclusion intervals per output port. That is, the methodology remains $k - 1$ fault-tolerant. Using multiple exclusion intervals helps to reduce the number of victim nodes. However, reducing the number of victim nodes does not increase the number of tolerated faults by our mechanism, since with k faults or more the worst-case scenarios are those in which the network is not connected, as commented before.

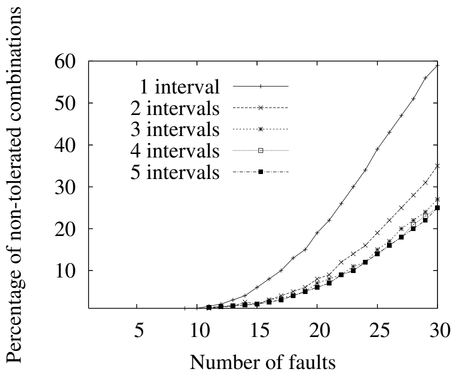
Nevertheless, using multiple exclusion intervals helps to reduce the number of non-tolerated combinations for k and more faults. That is, the percentage of non-tolerated combinations gets smaller as the number of exclusion intervals increases. Therefore, fault combinations that do not disconnect any node (no worst-case fault combinations) are easier to be tolerated, since there is a lower number of victim nodes. As an example, Figure 3.14(a) shows the percentage of non-tolerated fault-combinations for a 4-ary 3-tree for different numbers of exclusion intervals per output port. As it can be seen, adding a second exclusion interval has a large influence, specially when more than 15 faults are present in the network. This magnification on the advantage of using several exclusion intervals when the number of faults is increased is logical, since as the number of faults increases, the probability of including victim nodes also increases and, therefore, the usefulness of having multiple

Table 3.2: Number of tolerated faults for different k-ary n-trees varying the number of exclusion intervals per output port.

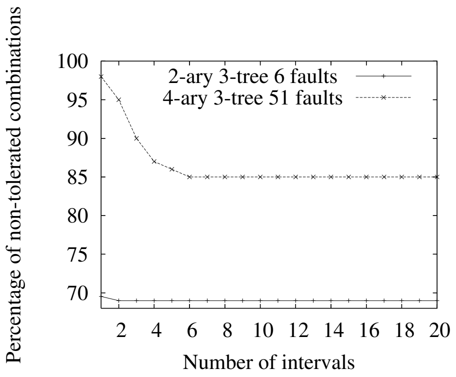
	Tolerated faults		
Network	1 excl. int.	2 excl. int.	3 excl. int.
2-ary 2-tree	1	1	1
2-ary 3-tree	1	1	1
2-ary 4-tree	1	1	1
3-ary 3-tree	2	2	2
3-ary 4-tree	2	2	2
4-ary 2-tree	3	3	3
4-ary 3-tree	3	3	3
8-ary 3-tree	7	7	7
8-ary 4-tree	7	7	7
8-ary 5-tree	7	7	7

exclusion intervals per output port. The availability of a third exclusion interval also gets improvements but adding more exclusion intervals has a lower impact. For small networks (2-ary 3-tree) and a limited number of faults, as it can be seen in Figure 3.14(b), it is not interesting to support more than two exclusion intervals. Nevertheless, for large networks (4-ary 3-tree) and considering a very high number of faults (see Figure 3.14(b)), it can be worth to have more exclusion intervals (up to 6 in this case). Hence, we can state that two exclusion intervals are enough, except for large systems with a high number of faults in the network.

In order to show that having multiple exclusion intervals per output port actually helps in reducing the number of victim nodes, we show in Table 3.3 the average number of victim nodes for 10 faults in a 2-ary 3-tree and in a 4-ary 3-tree varying the number of exclusion intervals per output port. Each row shows the mean number of victim nodes in the network for 1000 random fault combinations. As it was expected, when we increase the number of exclusion intervals, the number of victim nodes is reduced. Adding the second exclusion interval strongly reduces the number of victim nodes, but adding more intervals gives diminishing returns. This confirms the results shown in



(a) Varying the number of faults and exclusion intervals in a 4-ary 3-tree



(b) Varying the number of intervals in two different networks.

Figure 3.14: Percentage of non-tolerated combinations.

Table 3.3: Number of victim nodes for various k-ary n-trees with 10 faults.

Network	# of exclusion intervals	mean # of victim nodes
2-ary 3-tree	1	40.76400
2-ary 3-tree	2	34.51600
2-ary 3-tree	3	33.72000
4-ary 3-tree	1	134.79199
4-ary 3-tree	2	54.27200
4-ary 3-tree	3	46.52800
4-ary 3-tree	4	46.33600

Figure 3.14(a).

3.4.3 Dynamic Issues

In this section, we focus on the issues concerning the dynamic fault model. As commented, the dynamic issues are mainly the reconfiguration time, that is, how long takes to update all the exclusion intervals starting to count from the time at which the fault is detected, and the number of packets lost during the reconfiguration period. Remember that reconfiguration time is an important issue because, in our mechanism, another fault can not occur during the reconfiguration due to a previous fault. In the cases where we have decided to use emergency paths, we have analyzed how many packets are deviated through those paths in addition to analyze the number of lost packets, and the path length increase.

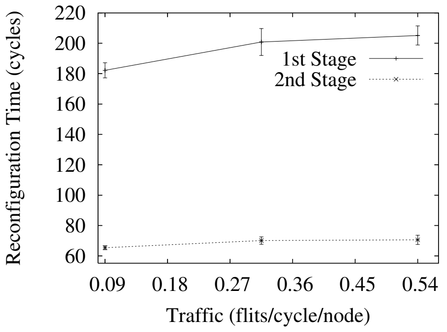
For analyzing these issues, for each network, we found its throughput saturation point (the maximum traffic accepted by the network). Then, we have chosen three different traffic points: the one that saturates the fat-tree (high traffic), its sixth part (low traffic), and the mean value between both of them (medium traffic). For each of those traffic load points in each network, we have introduced a link fault considering a significant number of different faulty link locations. Moreover, as we expected that the stage of the fault could also have some influence on the results, these fault combinations have been grouped depending on the stage in which the faulty link appears. In particular, we made 25 different simulations for each network stage, each of them corresponding to a different single down link fault combination[8].

We do not show any result concerning the number of control packets required by the strategy because it is extremely low compared to the number of simulated packets and their lifetime is very short, as it will be shown in this section. Notice that FT^2EI only uses one control packet during its upwards reconfiguration phase, whereas in the downwards phase, the affected switches forward only one control packet through all their down links. Switches receive just one control packet, no matter they belong to the upwards or the downwards reconfiguration phase. At worst, FT^2EI will generate as much control packets as half the number of switches plus one, which is negligible compared with the number of packets that have been simulated.

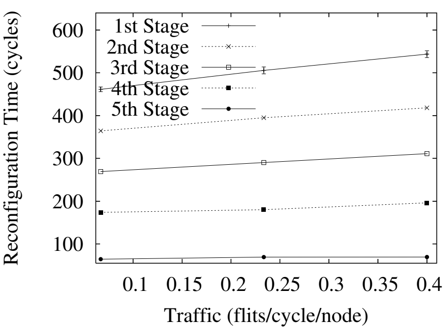
Reconfiguration Time

Figures 3.15 and 3.16 show how the reconfiguration time is affected by the injected traffic and the stage at which the fault is located for four networks sizes. In both figures, we show the average reconfiguration time classifying the faults by the stage at which they are located. Reconfiguration time is expressed in network cycles. As it was expected, reconfiguration time grows with the injected traffic. When the traffic is high, there are more packets in the network, so it is more likely that a link is busy when a control packet

[8]Since some of the networks have less than 25 links per stage (2-ary 3-tree and 2-ary 4-tree), we repeated some fault links appearing in the network at different times.



(a) 4-ary 3-tree.

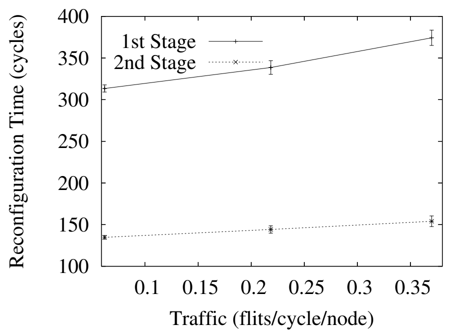


(b) 4-ary 6-tree.

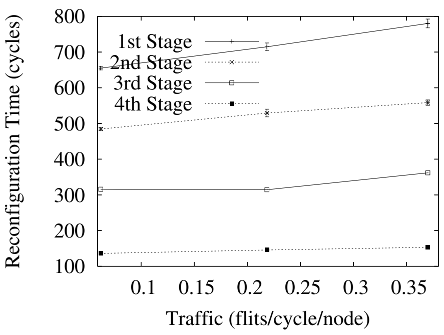
Figure 3.15: Reconfiguration time classifying the faults by the stage at which they are located for different traffic loads.

needs to go through it. Also, it can be observed that the reconfiguration time is higher when the fault is located in lower stages of the fat-tree and it is lower when the fault is located in the higher stages. As we have explained before, the fault-tolerance control packet must go up from the stage where the fault was discovered to the highest stage of the network, and then it must go down to the previous stage to the one where the fault was discovered. So, it is justified that the reconfiguration of faults in the lower stages of the network takes more time, since fault-tolerance control packets have to cross more stages. The reconfiguration time in the worst case is around 760 cycles. This time is noticeably smaller than the meantime between faults (according to [126] the network failure rate is 0.129 failures per week.).

Furthermore, it can be observed by comparing both figures that the arity of the switches also influences on the reconfiguration time. For instance, in Figure 3.15(a), the worst reconfiguration time is approximately 210 cycles, whereas the worst reconfiguration time is close to 375 cycles in Figure 3.16(a). The networks used in these figures have the same number of stages, but the arity of the switches of the second one is doubled with respect to the switches of the first one. This seems logical if we consider that in the downwards phase of the proposed mechanism, to spread the fault-tolerant information, the switches has to send a copy of the packet through all their down links. Therefore, as we increase the number of down links by increasing the switch arity, the



(a) 8-ary 3-tree.



(b) 8-ary 5-tree.

Figure 3.16: Reconfiguration time classifying the faults by the stage at which they are located for different traffic loads.

probability of finding that one of these links is being used by another packet is also increased. Nevertheless, notice that the increase of the reconfiguration time increases in a sub-linear way, so it is expected to be several orders of magnitude lower than the MTBF of large clusters no matter the arity of the switch.

Finally, we have performed an approximation for the availability that our mechanism would reach on the worst case (where reconfiguration takes 760 cycles) in the large-scale high-performance computers presented in the Table 2.1. The results of such approximation were obtained applying Equation 2.5, and are presented in Table 3.4. In the table, the network frequencies were obtained from [5, 13, 18, 120, 126]. In the two cases where we could not find documentation about the network frequency, we have approximated it by the frequency of the most common network technology according to [12], which is Gigabit Ethernet. For Gigabit Ethernet, we have used the working frequency of the implementation from Cisco [95]. As it can be seen, our mechanism can keep those systems available most of the time.

Number of Lost Packets

We have also analyzed the number of packets that are lost during the reconfiguration period. As commented previously, these packets are lost because they traverse the switches where the exclusion intervals will be set after the

Table 3.4: Availability of our mechanism applied in several large-scale high-performance computers. Computers marked with a ∗ are approximated by using Gigabit Ethernet characteristics.

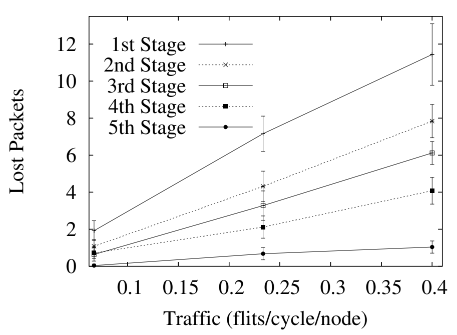
Computer	#CPUs	MTBF	Net. Freq.	Availability
ASCI Q	8192	6.5 hours	250 MHz	0.99999999916
ASCI White	8192	40 hours (2003)	500 MHz	0.99999999958
Seaborg	6656	14 days	125 MHz	0.99999999831
Lemieux∗	3016	9.7 hours	1.062 GHz∗	0.99999999980
Google	more than 100k	20 reboots/day	100 MHz	0.99999999789
Abe∗	9600	6 hours	1.062 GHz∗	0.99999999980
BlueGene	65535	6.16 days	1.4 GHz	0.99999999985

appearance of the fault, but before the control packet has arrived, so in their down phase, they would have as unique path the one that traverses the faulty link.

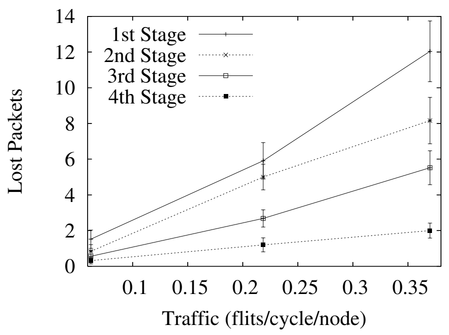
Figure 3.17 presents the number of lost packets for the simulations from Figure 3.15(b) and Figure 3.16(b). As it was expected, the number of lost packets grows with the reconfiguration time. Hence, the number of lost packets increases directly with the traffic load of the network and the switch arity. Additionally, it is increased as the stage where the fault is located at decreases. In the worst case, we only lost an average of 12 packets. This value is negligible if it is compared with the total number of packets that have been simulated.

Results with Emergency Paths

We have also performed simulations enabling the emergency path to test the effectiveness of this technique to avoid packet losing during the reconfiguration of the exclusions intervals. Contrary to the previous section, packets that have to cross a faulty link before the exclusion intervals have been updated are sent through a non-faulty down link of the faulty switch following an emergency path. As it was expected, the reconfiguration time keeps almost equal to the case where the emergency path is not used, but there are no lost packets. All the packets that would be lost are deviated through a non-minimal path that allows them to reach their destination without encountering a fault during



(a) 4-ary 6-tree.



(b) 8-ary 5-tree.

Figure 3.17: Number of lost packets classifying the faults by the the stage at which the fault is located for different traffic loads.

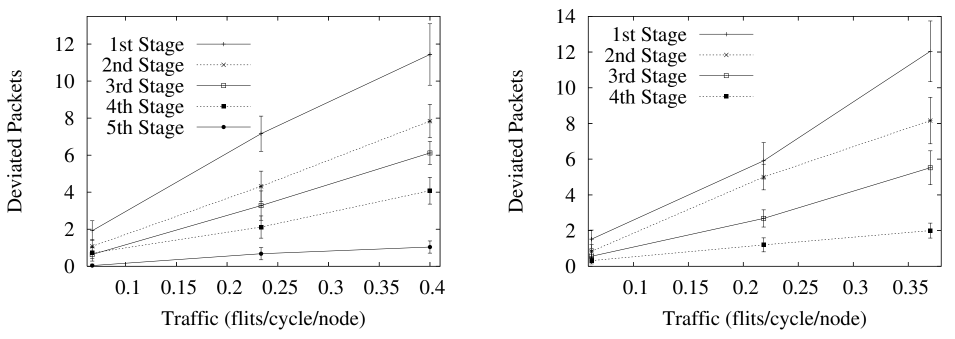
their travel along of the network. This can be seen in Figure 3.18, where the number of deviated packets is almost the same as the number of lost ones in Figure 3.17.

We have also measured the increase of the path length for the packets that are deviated. The deviated packets suffer a path size increase of two hops, since they are deviated sending them to a switch of the previous stage to the one at where the fault is located, increasing the length in one hop, and then they have to go up another stage to return to their path to the destination node, increasing in another hop the total length of the path.

Impact of control Packets Priority

Finally, we have also run several simulations to analyze the impact of not giving priority to control packets on the reconfiguration time, since reconfiguration time is a critical issue in our mechanism.

The most representative results can be seen in Figure 3.19. It shows the results for the same simulation whose results where shown in Figures 3.15(b) and 3.17(a), but without giving priority to fault-tolerance control packets over data packets. As expected, the reconfiguration time grows more quickly with the injected traffic. When the traffic load of the network is increased, the collisions between packets are more frequent. If priority is given to fault-tolerance control packets, these collision would be resolved favoring to the fault-tolerance



(a) 4-ary 6-tree.

(b) 8-ary 5-tree.

Figure 3.18: Number of deviated packets depending on which stage the fault is located at for a 4-ary 6-tree with emergency path.

control packets. Hence, by removing the priority of the fault-tolerance control packets, the waiting time due to collisions of these packets rises directly with the network traffic load. Nevertheless, reconfiguration time keeps being several orders of magnitude smaller than the failure rate occurrence in a supercomputer [126]. As reconfiguration time increases, the number of lost packets also increases, if no emergency path is used.

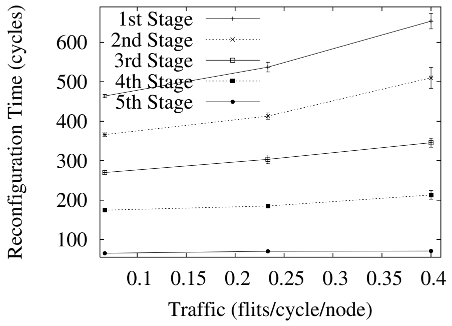
3.4.4 Impact on Network Performance

Now, we focus on the performance analysis of the network once the exclusion intervals have been updated, for a single exclusion interval and multiple exclusion intervals per output port.

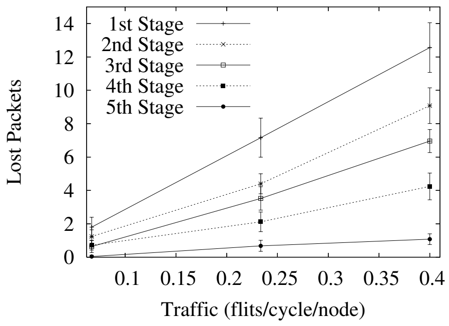
One exclusion interval

It is expected that network throughput after reconfiguration would be lower than the one before the fault appearance, since some paths of the network have been lost due to the fault. Due to this fact, we analyze how network performance is degraded when some faults are produced and are solved by FT^2EI using only one exclusion interval. Figure 3.20 shows the throughput degradation for several k-ary n-trees[9] when varying the number of faulty links

[9]For a 2-ary 2-tree and 5 faults, there are not tolerated combinations.



(a) Reconfiguration Time.



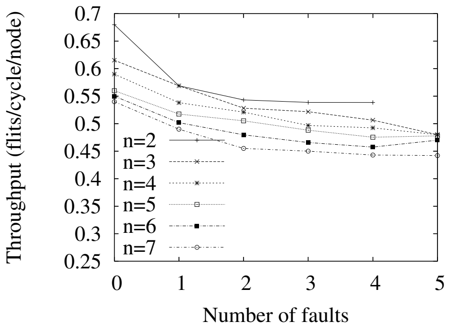
(b) Lost packets.

Figure 3.19: Results for a 4-ary 6-tree without giving priority to the fault-tolerance control packets.

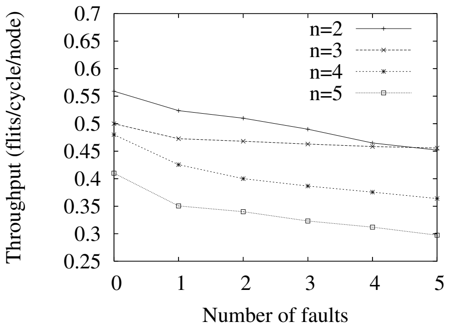
from 0 to 5. Each point represents the mean of 500 simulations, each one of them corresponding to a different randomly selected fault combination tolerated by our methodology (if there is a smaller number of tolerated fault combinations, a simulation is performed for each one of them). The error bars are always smaller than 0.05, so, for the sake of clarity, they are not shown. For one fault, the performance degradation varies from 6% to 14%[10]; for five faults the performance goes down between 8% and 25%. Again, the larger the network, the lower performance degradation due to the higher number of alternative paths[10]. Furthermore, as it can be seen, in the networks with lower arity ($k = 2$) the first introduced fault has a higher impact over performance than the following faults. Nevertheless, in the networks with $k = 4$, the performance degradation of each consecutive fault is very similar.

In an intuitive way, we can say that the lower stage the fault is located, the higher number of paths that are affected by the fault. For this, we have also analyzed the throughput degradation depending on the stage at which the fault appears. The results can be seen at Figure 3.21. We represent some networks with different number of stages and a faulty link, varying the stage at which the fault appears. In networks with low arity (k=2, Figure 3.21(a)),

[10]Notice that this is the smallest performance degradation that can be achieved since with one exclusion interval; no victim node is excluded and only the physical paths invalidated by the fault are not used.

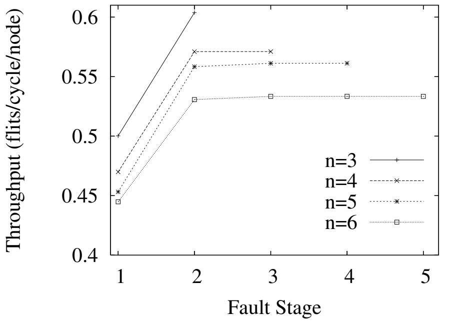


(a) k=2

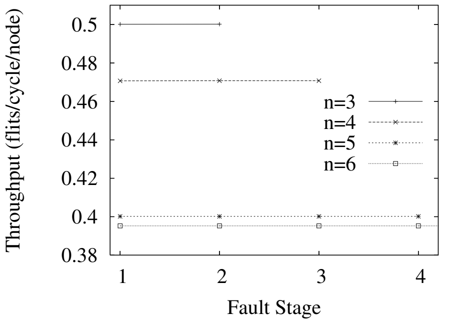


(b) k=4

Figure 3.20: Network throughput degradation for one exclusion interval.



(a) k=2



(b) k=4

Figure 3.21: Network throughput degradation depending on the which stage the fault is located at for several networks.

throughput is strongly affected when a fault happens at the first stage, but it keeps constant in the rest of the stages. In networks with larger arity (k=4), throughput degradation keeps nearly constant, independently of the stage at which the fault happens, even for stage 0.

This not-expected behavior can be actually explained by the number of paths that have been nullified when a fault is handled. Table 3.5 shows the number of paths that have been nullified and the percentage that they represent over the total number of them, depending on the stage at which the fault is located. We observe that, for networks that keep a constant performance degradation wherever the fault is located at, the percentage of nullified paths

Table 3.5: Number of lost paths due to a fault in some k-ary n-trees depending on the stage at which the fault is located.

Network	Stage	# of paths	# of lost paths	% of lost paths
2-ary 3-tree	first	168	20	11.90%
2-ary 3-tree	second	168	16	8.52%
4-ary 3-tree	first	52416	816	1.56%
4-ary 3-tree	second	52416	768	1.46%

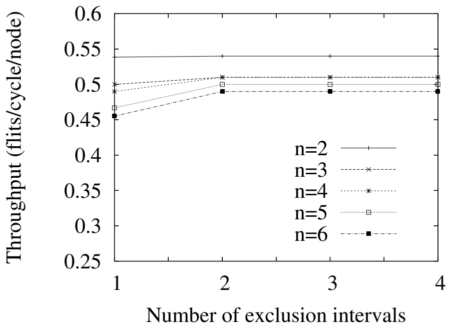
relative to the number of total paths is quite constant (see the 4-ary 3-tree). On the other hand, for the networks in which the performance degradation is higher in low stages, these percentages are quite different (see the 2-ary 3-tree).

Multiple exclusion intervals

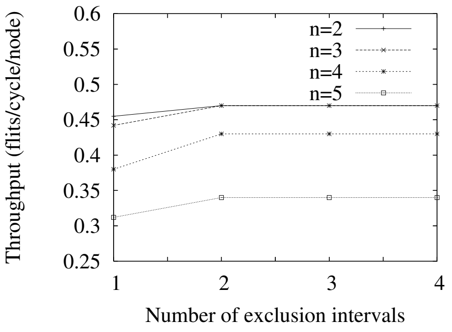
Just to end with the evaluation of the performance of the FT^2EI mechanism, we also consider the case of having multiple exclusion intervals per output port. Figure 3.22 shows, as an example, the performance degradation of several networks whose k is 2 or 4 for 4 faults[11] when considering several exclusion intervals, from 1 to 4. As it can be seen, from 1 to 2 exclusion intervals, the performance degradation is reduced in almost all the cases, but for more intervals, no additional improvements are obtained. This could be expected from the results shown in Table 3.3, as the percentage of lost paths due to the faults was really small in most of the cases. Notice that with 4 faults and 4 exclusion intervals no victim node is excluded and, therefore, the performance degradation is the lowest that can be achieved without adding new links, switches or virtual channels to the network.

Nevertheless, again, when considering a larger network and a very high number of faults (see Figure 3.23), additional improvements are obtained when increasing the number of exclusion intervals to tolerate faults. In this case, as it can be seen, up to 3 exclusion intervals per output port is worth to be used. Notice that as we increase the number of faults, we increase the number of victim nodes that are introduced in the exclusion intervals of the network. As

[11]The error bars are again smaller than 0.05, so they are not shown for the sake of clarity.



(a) k=2



(b) k=4

Figure 3.22: Network throughput degradation for 4 faults and multiple exclusion intervals.

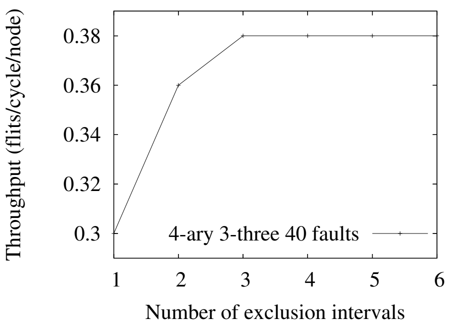


Figure 3.23: Network throughput degradation for multiple exclusion intervals. 4-ary 3-tree with 40 faults.

it was shown in Table 3.3, having more exclusion intervals helps in reducing the number of victim nodes. So, the extra exclusion intervals help in reducing this increase in the number of victim nodes, and therefore less healthy paths are eliminated from the network.

3.4.5 FT^2EI Memory Requirements

Finally, we analyze the amount of memory required at each switch by FT^2EI and by a routing algorithm based on forwarding tables.

On one hand, routing based on forwarding tables requires a table with as many entries as the number of destination nodes (N). Each entry must contain a node identifier and the port, or ports for adaptive routing, returned by the routing function. Switches have $2 \times k$ ports. The length of the identifiers of the destination nodes is $log(N)$, and the number of bits required to represent the output is $log(2 \times k)$. Hence, the cost of this alternative is $C_{FT_{det}} = N \times (log(N) + log(2 \times k))$ bits in each switch, for deterministic routing. In adaptive routing, the routing function of a fat-tree can return up to k results. Therefore, the routing table must be able to store k output ports for each destination of the network. The cost of implementing adaptive routing in fat-trees with forwarding tables is $C_{FT_{adap}} = N \times (log(N) + k \times log(2 \times k))$ bits in each switch. Clearly, both, for adaptive and deterministic routing, cost is $O(N)$, which is not scalable with the network size. Indeed, the fault-tolerant mechanism used in a table-based network could require additional memory.

On the other hand, we have FT^2EI that is based on IR. As before, we assume that the network is composed by N nodes, built with switches with $2 \times k$ ports. IR associates two routing registers to each output port, each of size $log(N)$. So the total number of bits per switch for IR intervals is $2 \times k \times 2 \times log(N)$. Additionally, FT^2EI needs to associate additional registers to each ascending output port for the exclusion intervals. In particular we need two registers per each exclusion interval. So, being n_{ei} the number of exclusion intervals, our proposal needs $2 \times n_{ei} \times k \times log(N)$ bits per switch for the exclusion intervals. Therefore, considering also the required inclusion interval (2 registers), the total amount of memory per switch is $C_{FT^2EI} = (2 + n_{ei}) \times 2 \times k \times log(N)$, which is $O(log(N))$ and is clearly scalable with the network size. Moreover, FT^2EI relies on a low number of exclusion intervals per output port, since having more intervals only makes sense in very non-reliable large networks.

Finally, Figure 3.24 shows the required total memory per switch as the number of nodes in the network is increased, for a routing scheme based on forwarding tables and one that is based on FT^2EI with 2 exclusion intervals. We show the results up to 1M nodes to show how both schemes could handle the expected number of nodes in next-generation supercomputers. For this, we set $k = 32$ to reflex the trend of using high-radix switches. Despite this high value of k, which increases the memory required by FT^2EI, it stays several orders of magnitude smaller than the one required by forwarding tables. For

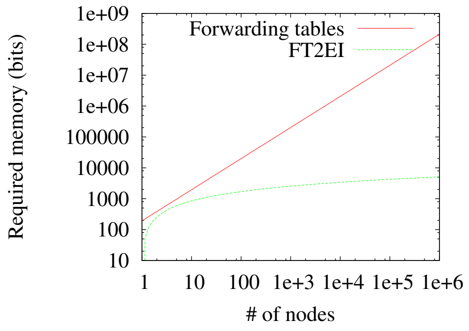


Figure 3.24: Memory requirements per switch for a routing scheme based on forwarding tables and one based on FT^2EI with 2 exclusion intervals in a network with $k = 32$.

this value of k, forwarding tables are not worth for any system size in adaptive routing. Furthermore, notice that the high-radix trend on high-performance interconnects enhances the fault-tolerance provided by our mechanism, as it would be able to tolerate a higher number of faults.

3.5 Conclusions

In this chapter, we have proposed an efficient fault-tolerant distributed adaptive routing strategy (FT^2EI) for k-ary n-tree interconnection networks. The FT^2EI methodology does neither need complex hardware, nor replicating any network component, nor a large amount of memory. It is based on enhancing the well-known Interval Routing scheme with exclusion intervals. Each output port is provided with an exclusion interval, that represents the set of nodes that become unreachable from the output port after a fault. We have also proposed a simple algorithm to calculate the exclusion intervals of the affected switches, avoiding penalizing those nodes that are not affected by the fault and allowing adaptive routing in all the healthy paths. Also, a very efficient mechanism to dynamically spread the exclusion interval information without stopping the network is proposed, thereby avoiding the disadvantages of using a static fault model. Moreover, we have proposed a mechanism to avoid packet losing while reconfiguration is in progress.

The evaluation results show that the FT^2EI methodology is able to completely tolerate all the fault combinations of $k-1$ links. Moreover, for medium and large networks, FT^2EI is able to tolerate a large number of fault combinations with a very high probability (98.05% of probability for 8 faults in a 4-ary 3-tree network). The price paid is some performance degradation, which depends on the number of faults and the network size. We have also analyzed the impact of using several exclusion intervals per output port to improve our mechanism. Although associating several exclusion intervals per output port does not help in increasing the number of tolerated faults for a given topology, the percentage of non-tolerated combinations of faults is strongly reduced, and also the performance degradation of the network. We have obtained that two exclusion intervals per output port is enough for reliable networks. More than two exclusion intervals only makes sense for networks in which the faults have eliminated a high percentage of the paths of the network.

Finally, the time required to make the reconfiguration of the exclusion intervals is several orders of magnitude lower than real systems MTBF, which assures the correct behavior of the proposed dynamic fault handling mechanism.

Chapter 4

DESTRO: Effective Deterministic Routing in Fat-Trees

"You step into the Road, and if you don't keep your feet, there is no knowing where you might be swept off to."

Frodo Baggins, quoting Bilbo Baggins, Lord of the Rings.

Routing is a key design issue in interconnection networks, since network performance is greatly influenced by the chosen routing algorithm. There are several classifications for routing algorithms, the most common one differentiates between deterministic and adaptive routing algorithms. Deterministic routing algorithms always provide the same unique predefined path for the packets from a given source node to a given destination node. On the other hand, adaptive routing algorithms can adapt the paths that they provide to the traffic conditions of the network.

Adaptive routing usually obtains a higher performance than deterministic routing. However, it may introduce out–of–order packet delivery, if no additional mechanisms are used. In-order delivery of packets is mandatory for several applications, like some cache coherence protocols [84], and network technologies, like Infiniband [7]. The only way of keeping the delivery order of packets in adaptive routing is by using specific mechanisms. However, these mechanisms increase the complexity of the network and may reduce significatively its performance.

Furthermore, the required logic to implement adaptive routing is composed of a routing function that provides a set of output ports for each routed packet, and a selection function that selects the one that would be used as output port for the packet that is being routed. In deterministic routing algorithms, the selection function is not necessary since a deterministic routing function always returns only one output port. Therefore, the complexity of the implementation of an adaptive routing algorithm is further increased if it is compared to the one of a deterministic routing algorithm, since they require the use of selection functions and in-order delivery mechanisms. Nevertheless, deterministic routing algorithms usually achieve a lower performance compared with an adaptive routing algorithm, since they cannot react to the changes in the conditions of the traffic in the network.

Concerning topology, as previously commented, most of the commercially available interconnects are based on fat-trees. In the previous chapter of this dissertation, we took advantage of the huge path diversity of the fat-tree topology to provide several alternative routes to use when a fault is detected in the network. In this chapter, we take on the challenge of proposing a deterministic routing algorithm that is able to efficiently exploit the available rich connectivity of the fat-trees. In this way, we pretend to propose a new deterministic routing strategy with low complexity that preserves in–order delivery of packets and at the same time providing a similar performance as the adaptive routing algorithms. The proposed routing mechanism is very simple but at the same time very powerful, since it is able to balance network traffic. Indeed, opposite to previously–proposed approaches, it allows to retain contention as much as possible, leading to a deterministic routing algorithm that can outperform previously–proposed deterministic routing algorithms and the commonly–used adaptive routing algorithm while keeping all the advantages of deterministic routing such as in–order packet delivery and a simple implementation.

The rest of the chapter is organized as follows. Section 4.1 briefly introduces the motivation of the chapter, and explains several basic points about adaptive and deterministic routing. Our proposed deterministic algorithm is presented in Section 4.2, providing in addition an efficient way of implementing it. Next, Section 4.3 provides an exhaustive evaluation of our proposal against adaptive routing using several selection functions and other deterministic routing algorithms proposed for fat-trees. Finally, some conclusions are drawn in Section 4.4.

4.1 Introduction

As commented, the main difference between adaptive and deterministic routing algorithms is the number of possible routes for the packets. In deterministic routing schemes, an injected packet traverses a fixed predetermined path between source and destination; while in adaptive routing schemes, the packet may traverse any of the different alternative paths available from the packet source to its destination.

As adaptive routing algorithms can provide several different routes for the packets, they are very flexible and can adapt to changes in the conditions of the traffic in the network to provide better routes for the packets that are traversing the network. For example, Figure 4.1 presents a network using an adaptive routing algorithm. In this network, the node labeled Origin sends a packet to the one labeled Destination, as seen in Figure 4.1(a). This packet is sent to a switch that is heavily congested, and gets stopped, see Figure 4.1(b). At the same time, the same node sends a second packet to the same destination node. However, as depicted in Figure 4.1(c), the adaptive routing algorithm has detected that the switch at which it forwarded the first packet is heavily congested, and it deviates this second packet, avoiding in this way that the packets traverse the congested zone that would likely speed it down, increasing its latency. In this way, the packet can reach its destination using a less congested path, see Figure 4.1(d). This flexibility is the great advantage of adaptive routing algorithms over deterministic ones, since it usually allows them to achieve better performance results than deterministic routing algorithms.

However, this flexibility increases the complexity of the implementation of the adaptive routing algorithms, since they require to use a selection function in addition to the routing function [40]. In adaptive routing algorithms, the routing function supplies a set of one or more routing options, so there must be another function that selects one of these routing options according to the conditions of the network traffic to provide the best possible routes for the packets, and this is the selection function.

Moreover, the flexibility of adaptive routing algorithms can further increase the complexity of the implementation of the network, because adaptive routing algorithms cannot ensure to preserve packet ordering by themselves due to this adaptivity. An example of packet disorder provoked by the adaptivity of the network is shown in Figure 4.1. As commented, in this example, there is a switch in the network that is suffering from heavy congestion. Any packet that tries to cross this switch would likely be stopped for a long time before being able to continue its travel. As commented, in the first figure, the node labeled as Origin sends a packet to the node labeled as Destination. This packet is routed by the switch attached to the node, that decides to forward the packet through its first up link, directly to the congested switch. Immediately after the first packet, the same node sends another packet to the same destination. But, this time, the switch attached to the node decides that the second packet is going to be forwarded through its second up link, avoiding the congested zone, as explained previously. As it can be seen, the first packet would take a long time to traverse the congested switch, while the second one has avoided the congested zone and continues its travel to the destination node. At the end, the second packet reaches the destination before the first one. Thus, the delivery order of the packets is broken. For this reason, when in-order delivery of packets is mandatory like in certain cache coherence protocols [84] and some communication technologies [7], adaptive routing algorithms require to use complementary techniques to keep the packet delivery order, increasing the global complexity of the network.

On the other hand, deterministic routing algorithms are considered to achieve a lower performance than adaptive routing algorithms due to their lack of adaptivity. However, thanks to their lack of flexibility they are more simple to implement than adaptive routing algorithms. For example, they do not require the use of a selection function, because the routing function always return only one routing option.

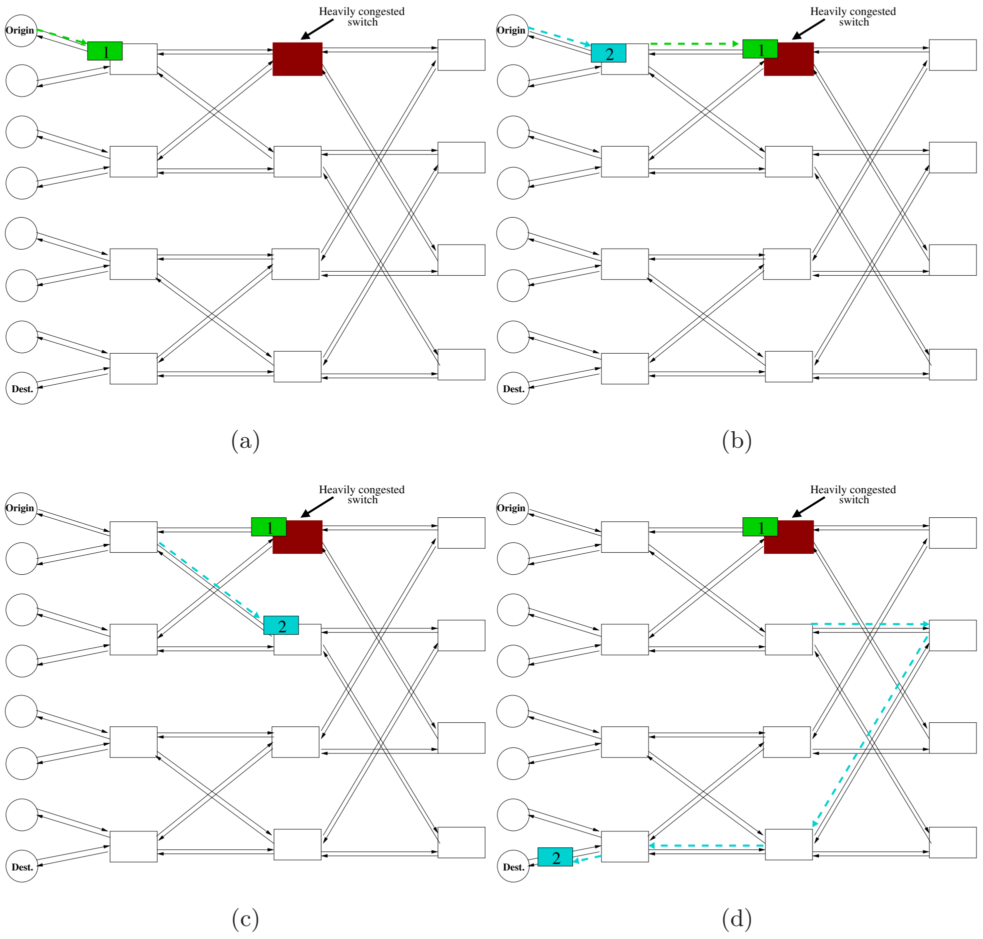
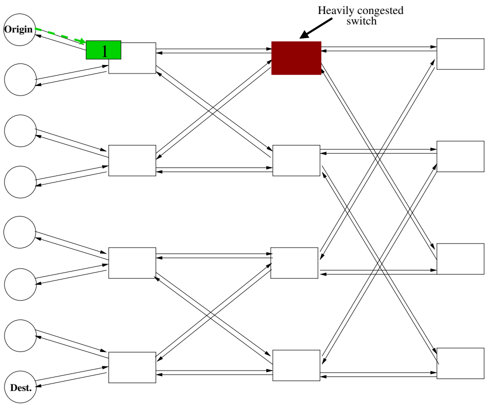


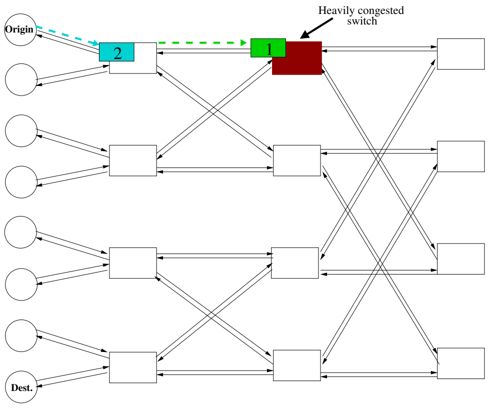
Figure 4.1: Step-by-step example of two consecutive packets that follow different paths in a fat-tree with adaptive routing.

Furthermore, deterministic routing algorithms always preserve packet delivery order, independently of the traffic conditions of the network. Let's check this with the same example as before. Figure 4.2 presents the same scenario as before, but using deterministic routing instead of adaptive routing. As before, we have a switch that is heavily congested, and the node labeled as Origin sends a packet to the node labeled as Destination. Again, the switch attached to the node forwards the packet through the link that connects it to the congested switch. As in the previous example, the same node sends a second packet to the same destination immediately after the first packet. However, in deterministic routing the paths are fixed, and cannot be changed. So, if the first packet has been forwarded to the congested switch, the second packet would also be forwarded to that switch. After being stopped in the congested switch, both packets would reach their destination in the same order that they were injected into the network. As it can be seen, deterministic routing algorithms preserve the packet delivery order no matter the traffic conditions of the network. However, in deterministic routing algorithms, if the path traverses a congested zone like in the example, all the packets would be delayed by the congestion. Nevertheless, in adaptive routing algorithms, at least one of the packets was able to reach the destination avoiding the congestion. Notice that this situation with just one switch that is heavily-congested is very unrealistic, since congestion is usually spread around the congested switch. That is, in adaptive routing algorithms the deviated packets would also be delayed by the congestion and the differences in the increase of packet latency due to congestion between deterministic and adaptive routing algorithms are not as important as they can seem.

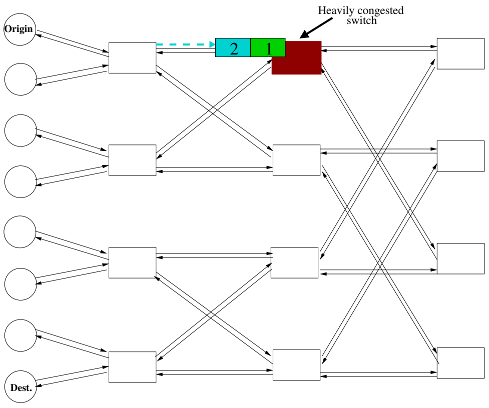
As commented in Chapter 2, routing strategies can also be classified according to the place where the routing decisions are made as source or distributed routing. Concretely, in Section 2.1.6, we explained that distributed routing is usually implemented either by a fixed hardware, specific to a given routing function on a given topology which lacks of flexibility; or by using forwarding tables that are very flexible but suffer from a lack of scalability. In the previous chapter, we presented a scalable distributed implementation of the commonly used adaptive routing algorithm in fat-trees based on Interval Routing (IR). In this chapter, we use the implementation showed in Section



(a)



(b)



(c)

Figure 4.2: Step-by-step example of two consecutive packets that preserve their order despite the network conditions with deterministic routing.

3.1 to implement the routing function for adaptive distributed routing.

4.2 Description of the Deterministic Routing algorithm

Our work to propose a new deterministic routing algorithm for fat-trees is based on the aforementioned adaptive routing algorithm. We have decided to use this algorithm as the base of our study since it is widely-used and it is established as the default routing algorithm in fat-trees. As commented in the previous chapter, this adaptive routing algorithm is composed of two phases. The first one is an upwards phase that it is fully adaptive, followed by a deterministic downwards phase. In the upwards phase, at each traversed switch, there are k output ports returned by the routing function. A selection function is used to select the output port finally used. This phase is fully adaptive since all the ascending ports can be used in order to reach the destination of the packet. In this phase, the packet travels upwards from the nodes located at the leaves of the fat-tree to the root of the tree. When the packet reaches a switch that is one of the nearest common ancestors between source and destination nodes, the packet enters in the downwards phase. In this phase, the packet travels downwards from the switches of the fat-tree to its destination node located at the bottom of the tree. This phase is deterministic, that is, only one path is provided. This downwards path is completely determined by the switch reached in the upwards phase, as it can be seen in Figure 4.3. The figure shows all the possible routes from the source node labeled as Origin to the node labeled as Destination, each highlighted in a different color. As it can be seen, each ascending adaptive path leads to a different common ancestor switch, and the descending path towards the destination is different for each ancestor switch. That is, the path of the downwards phase depends completely by the path that was used during the upwards adaptive phase. So, notice that the decisions taken during the upwards phase may be critical later during the downwards phase.

In order to propose a deterministic routing algorithm from the adaptive one, we have to reduce all the routes between the nodes of the network to just one route per source–destination pair. In the adaptive routing algorithm

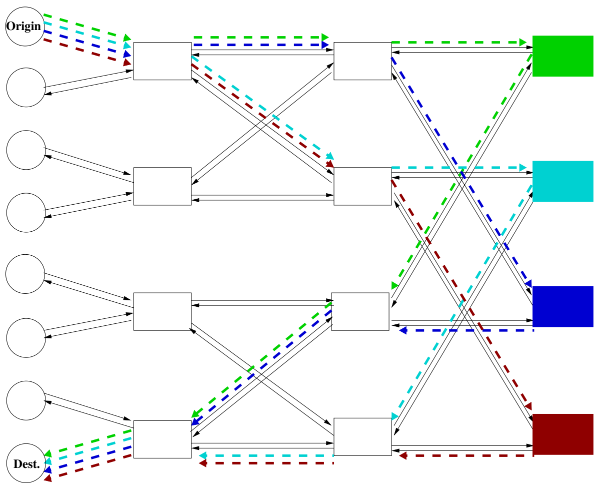


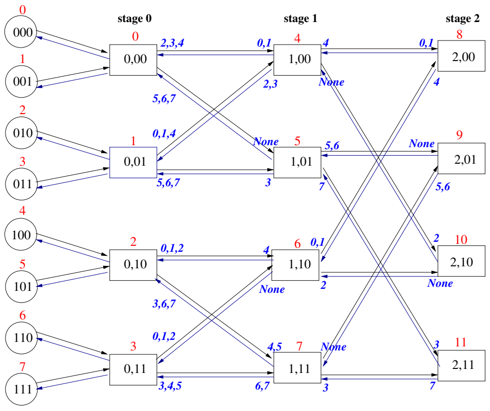
Figure 4.3: The different adaptive routes from node 0 to node 7 in a 2-ary 3-tree highlighted in different colors.

for fat-trees, since only one of its phases is adaptive. The downwards routing phase is deterministic, so nothing must be done in this phase to reduce the number of paths for each pair of nodes. Therefore, we have to reduce all the possible paths on the upwards phase to a unique path for each pair of nodes, but this reduction must be made with the aim of making an efficient use of the available rich connectivity of the topology. In order to avoid saturating some links that would limit the performance of the network, the selection of the paths for the deterministic routing algorithm should be done trying to balance network link utilization. That is, all the links of the same stage should be used by a similar number of paths. This is easy to obtain in the ascending phase. For example, an alternative to do that is to divide the set of reachable destinations through the k up links of a switch into k sub-sets of the same size. Each of these sub-sets should be associated to each up link to determine which one is used to reach each destination. However, the way to create these sub-sets is not straightforward, since balancing the number of paths that use a link in the ascending phase does not ensure that the utilization of the links of the down phase is also balanced. Remember that the paths used during the upwards routing phase determines the one used in the downwards routing phase. So, we must balance the paths that use each links in both stages. A first approach to group the destinations is to associate consecutive destinations to each up link in the first stage. Nevertheless, as it can be seen in Figure 4.4(a)[1], this approach is very naive since it balances the number of routes in the up links of the first stage, but it cannot balance the number of routes in the up links of the switches of the second stage. Anyway, these same groups of destinations per port can be arranged in a proper way, as shown in Figure 4.4(b). As it can be seen, with this second grouping, upwards paths are equally used by the same number of paths, but down links utilization is still unbalanced. As all the down links route packets to the same number of destinations, it may seem that they are used by an equal number of routes, but that is not true. For instance, in Figure 4.4(c), we have highlighted all the paths that lead to node 0. In the figure, the down link that connects switches 4 and 0 is used by two routes; on the other hand, the down link that connects switches 5 and 0 is used by four routes. As it can be seen, one of the down links is used by twice the number of routes than the other. Notice that those two links are only used by routes to nodes 0 and 1, and routes to both nodes suffer from the same unbalance. Furthermore, the same fact happens with the routes to the rest of nodes, not only with the ones that lead to node 0 and 1. This is an example of even utilization in the up links, but not in the down links.

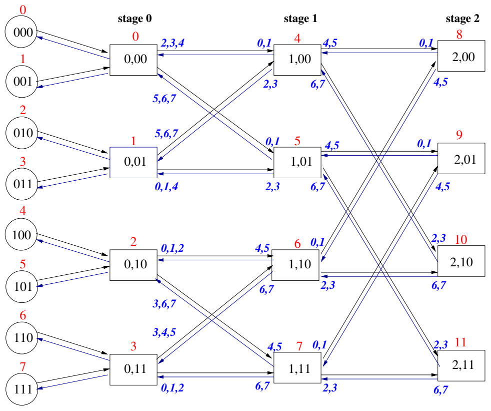
We have analyzed several approaches, trying to balance both routing phases, and found that a good alternative is to shuffle, at each switch, consecutive destinations in the ascending phase. In other words, consecutive destinations are distributed among the different ascending links, reaching different switches in the next stage. In this way, both routing phases are balanced. Moreover, the down phase becomes exclusive for each destination avoiding the Head-of-Line blocking effect and collisions between packets directed to different destinations.

In order to help to understand the mechanism, we will explain our proposed deterministic routing algorithm by using an example. Figure 4.5 shows the destination node distribution in the ascending and descending links of a 2-ary 3-tree using our proposal. In the first stage, consecutive destinations are shuffled between the two up links. To do that, the least significant component

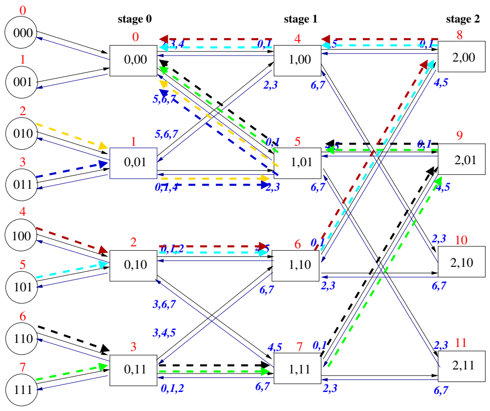
[1]In the figure, each link has been labeled (in dark blue bold–italic font) with the destinations whose packets will be forwarded through it.



(a) Grouping consecutive destinations in the first stage.



(b) Balancing the number of routes in the upwards phase.



(c) Paths to node 0 are highlighted.

Figure 4.4: Two possible groupings of destinations for deterministic routing.

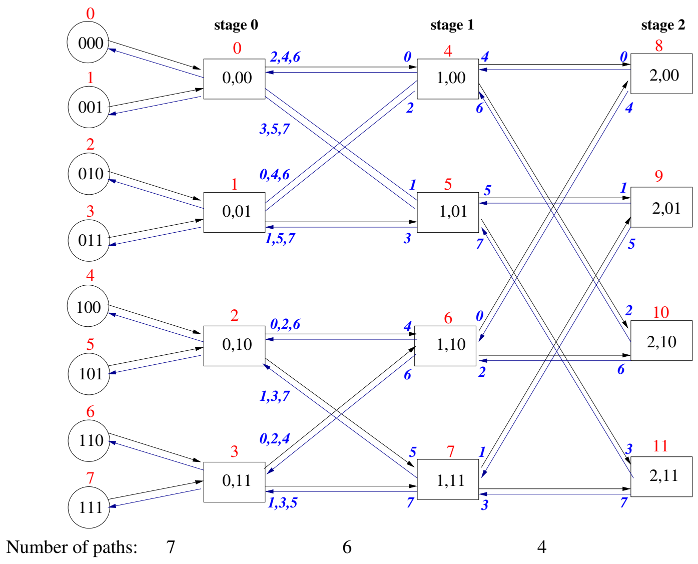


Figure 4.5: Reachable destinations through each output port in a 2-ary 3-tree with our proposal.

of the packet destination address (the least significant bit in this example) is used to select the ascending output port. That is, packets that must be forwarded upwards select the ascending output port indicated by the least significant component of the packet destination (p_0). Therefore, consecutive destinations are sent to different switches in the next stage. For example, switch 0 forwards through its up links packets to the destinations from 2 to 7. This switch sends through its first up link the packets whose destination identifier ends in 0, that is, packets with destinations 2 $\langle 010 \rangle$, 4 $\langle 100 \rangle$, and 6 $\langle 110 \rangle$; and it sends through its second up link the packets whose destination identifier ends in 1, that is, packets with destinations 3 $\langle 011 \rangle$, 5 $\langle 101 \rangle$, and 7 $\langle 111 \rangle$.

At the second stage, we cannot use the least significant component of the destination address to route the packets since the destinations of all the packets that reach a given switch in this stage share the least significant component. Let's check this in Figure 4.5. For example, switch number 5 can only receive packets in the upwards phase from switches 0 and 1. As it can be seen in the figure, switches 0 and 1 only forward packets with destinations 1, 3, 5, and

7 to switch number 5. The least significant component of these destinations is 1. As the least significant component can not be used in this stage, the component to consider in the selection of the up output port is the next one from the destination address. For instance, as shown at switch 5, only packets destined to nodes 1, 3, 5 and 7 reach that switch, and only packets destined to nodes 5 and 7 must be forwarded upwards. Packets destined to node 5 ($\langle 101 \rangle$) select the first up link since its second least significant component is 0, and packets destined to node 7 ($\langle 111 \rangle$) the second one since its second least significant component is 1.

Considering all the switches that belong to stage 1 (the second stage), packets destined to nodes 0, 1, 4 and 5 use the first up output port of the switches and those packets destined to nodes 2, 3, 6 and 7 use the second output port. That is, the second least significant component of the packet destination is used in the second stage to route packets.

In the general case, the selection of the output port made by our mechanism is based on the destination identifier and the stage of the switch that is routing the packet. For this, we will refer to this deterministic routing algorithm as Deterministic dEstination and STage based ROuting in fat-trees (*DESTRO*). In particular, *DESTRO* considers the component of the packet destination corresponding to the stage at which the switch is located at, (i.e., a switch located at stage s considers the s^{th} component of the destination address). That is, at the switch $\langle s, o_{n-2}, ..., o_1, o_0 \rangle$, the selected output port for a packet that must be forwarded upwards whose destination is given by $p_{n-1}, ..., p_1, p_0$ will be $k + p_s$. k is added to the component of the destination address because up links start at k in our link numbering. In this way, we use the up port corresponding to the component of the destination address.

Notice that we only change the allowed paths during the upwards phase from the adaptive routing algorithm, therefore we do not make any change in the routing of the downwards phase as this phase was already deterministic. Also, notice that the packets would transit from the upwards phase to the downwards phase in the same stage than in the adaptive routing algorithm. However, instead of being able of starting it in any of the nearest common ancestors between the source node and the destination node of the packet, the packets can only reach one of these nodes since the upwards routing phase is

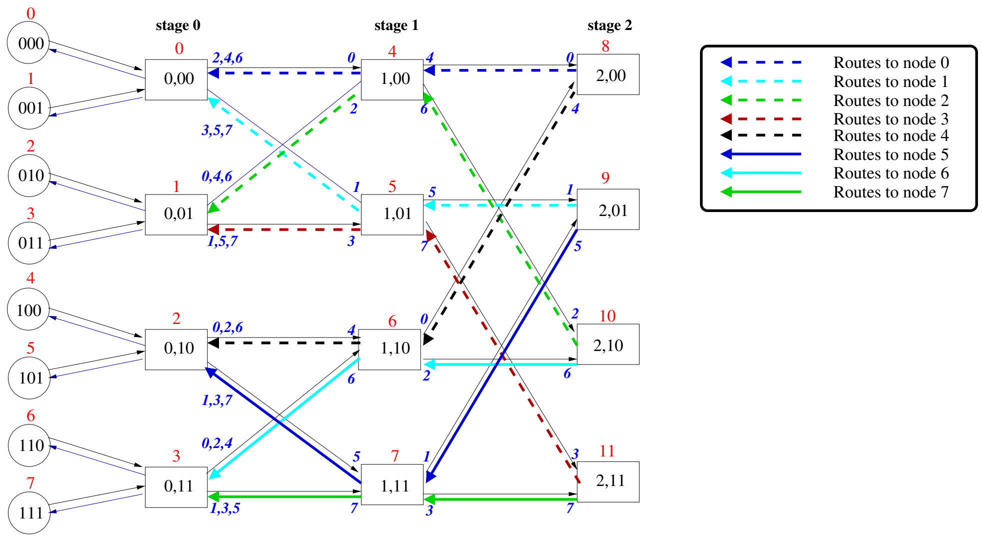


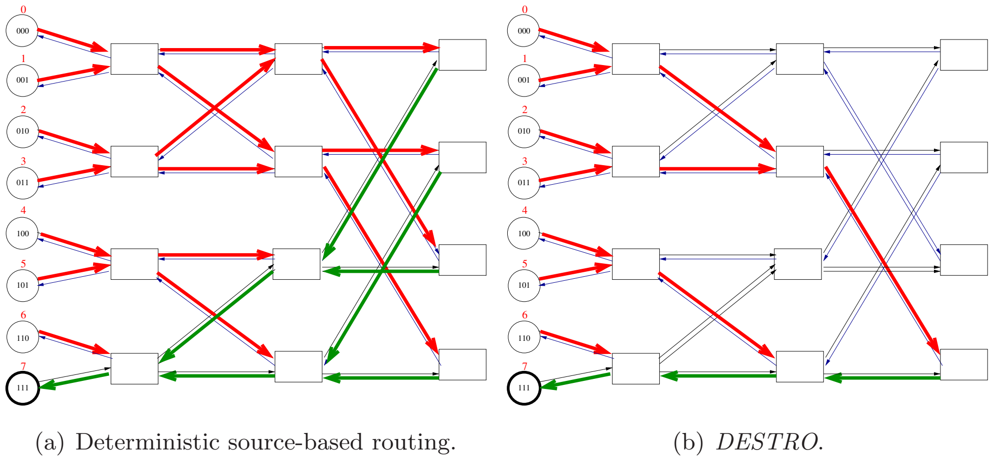
Figure 4.6: *DESTRO* downwards routing paths for each destination are highlighted in different colors in a 2-ary 3-tree.

no longer adaptive.

As it can be seen, the mechanism is very simple and easy to implement, since in the upwards phase a single component of the destination address indicates the output port to use. Despite this simplicity, the resulting deterministic routing algorithm has two very important features. First, *DESTRO* evenly distributes the traffic destined to different nodes, as shown in Figure 4.5. The bottom of Figure 4.5 shows the number of paths (source–destination pairs) that use each link at each stage. Both, the ascending and descending links of a given stage are used by the same number of paths. So, traffic in the network is completely balanced. This traffic balance is also achieved by another previously-proposed routing algorithm [78].

The difference between our proposal and previous proposals like [78] is that our proposed routing algorithm retains the congestion into the congested paths without blocking traffic that is not directed to the congested point, opposite to what is done in [78]. In *DESTRO*, as it can be seen in Figure 4.5, packets whose paths from their source node to the destination have the same length and are destined to the same node begin their downwards routing phase in the same switch, sharing the same descending path to their destination. This can be easily seen in the switches of the last stage. Each switch of the last stage receives packets addressed only to destinations whose identifiers differ only by the most significant component, and those packets are forwarded through a different descending link depending on their most significant component. In this way, each destination has a different descending path that does not share any link with packets with other destinations. Figure 4.6 highlights all the down links of a 2-ary 3-tree coloring the links depending on the destination to which they forward packets to in *DESTRO*. As it can be seen in the figure, all the down links are only used by packets belonging to one destination, completely avoiding the HOL blocking effect, avoiding merging packets destined to different nodes. Furthermore, in order to reach the same switch to start the downwards routing phase, in *DESTRO*, the paths to the same destination during the upwards phase become grouped inside a sub-tree, converging to the switch at which they begin their downwards routing phase, as it can be seen in Figure 4.7(b). As a consequence, the congestion of one destination does not affect packets destined to other destinations.

On the contrary, in [78] packets destined to the same destination that have different source nodes use almost disjoint ascending and descending paths, collapsing the whole network if a single destination node becomes congested. This can be seen in Figure 4.7, where all the paths to destination 7 from all the other nodes (nodes 0 to 6) are shown, for a source-based routing like the one proposed in [78] (Figure 4.7(a)) and for our proposed mechanism (Figure 4.7(b)). As can be seen in Figure 4.7(a), 16 network links are used to communicate only with destination node 7 and a single hot-spot can saturate almost all the network links; whereas in Figure 4.7(b), 6 network links are used, and the traffic of a single hot-spot only saturates one of the sub-trees of the network, not the whole network. In the routing algorithm proposed in [78], if a destination becomes congested, the packets to this destination would be spread over most of the switches of the network, and packets directed to non-congested destination would be blocked by the Head-of-Line blocking effect. Notice that the congestion is extended all over the network in the same way with adaptive routing, as adaptive routing algorithms try to balance the traffic load by using the output links that are free when their preferred one can not



(a) Deterministic source-based routing. (b) *DESTRO*.

Figure 4.7: Routes from all the source nodes to destination node 7 in source-based routing and *DESTRO*.

be used.

4.2.1 Implementation of *DESTRO* by using Flexible Interval Routing

In the previous chapter, we showed how the commonly-used adaptive routing algorithm for fat-trees can be implemented by using IR in a very simple and compact way. Now, in this section, we show how *DESTRO* can be also implemented in a very simple way by using Flexible Interval Routing (FIR) [62], an extension of IR that allows using different routing algorithms by adding only two additional registers per output port. In order to allow the reader to understand FIR without any additional document, we briefly first introduce FIR.

FIR

FIR was initially proposed to implement the most commonly–used routing algorithms in meshes and tori. Most of these routing algorithms could not be implemented by using IR due to its lack of flexibility in the interval definition. In FIR, as in IR, each output port has an associated interval that can be cyclic, which is implemented with the LIB and UIB registers, like in IR. But,

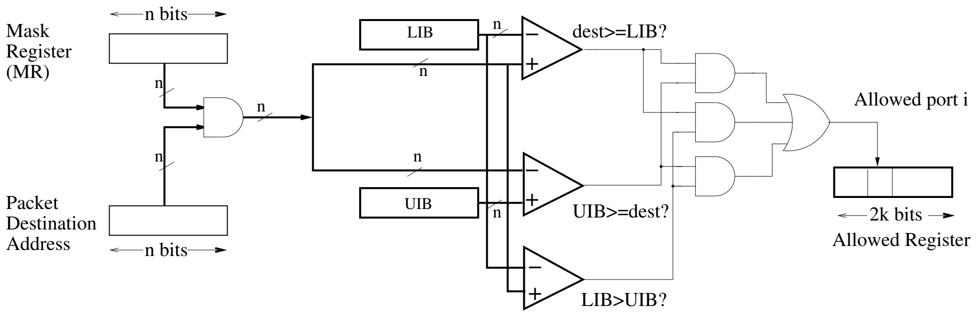


Figure 4.8: FIR hardware associated to each output port for masking the direction address of the packet and comparing it with the routing interval.

in order to add flexibility, additional registers are associated to each output port. In particular, each output port has a Mask Register (MR) that indicates which bits of the packet destination address will be actually compared with the routing interval (provided by the LIB and UIB registers). An implementation of this logic is depicted in Figure 4.8, in which the destination of packet is masked with the MR and then compared with the LIB and UIB registers. As in IR, the result of this comparison is stored in the allowed output port register (AR), that stores which output ports can be used by packet that is being routed. As can be seen, this implementation is very similar to the IR implementation shown in Figure 3.1. The main difference is the presence of the MR in Figure 4.8.

Additionally, FIR allows to apply some routing restrictions by means of an additional register associated to each output port, the Routing Restrictions Register (RRR), which defines, for each output port, which other output ports of the switch should be selected prior to this one. This register has one bit per output port. For a given output port i, the j bit in the RRR of the port i indicates whether the output port j has more preference (bit set to 1) or not (0) than output port i. Thus, the final routing decision for an output port i is obtained not only by comparing the masked destination with the interval bounds, but also by checking the bits in its RRR. The logic for the RRR checking is shown in Figure 4.9. The RRR was introduced in FIR to introduce an order in channel usage, thus ensuring deadlock-freedom of the

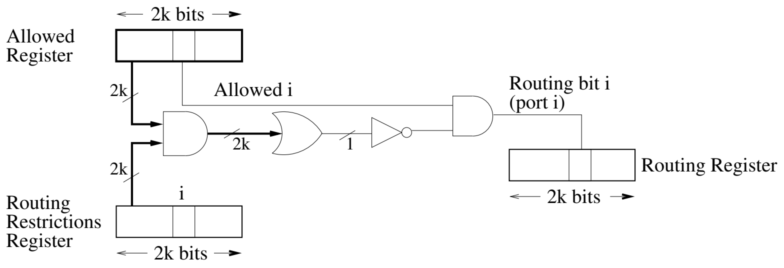


Figure 4.9: FIR hardware associated to each output port to prioritize output ports.

routing algorithm. However, the up/down routing algorithm used in the fat-trees is deadlock-free by default. Nevertheless, we still use this register for other purposes, as it can be seen below.

Implementation of ***DESTRO***

With these basic notions about FIR, we can show how these few registers per switch can be configured to route packets in fat–trees following the proposed deterministic routing algorithm. Notice that the adaptive routing algorithm described in Section 3.1 can be also implemented with FIR, since IR is a subset of FIR.

We begin explaining how to configure the ascending links registers. They are configured in a very different way than in the adaptive routing case, since the number of ascending paths is reduced to one for each source–destination pair. At each switch, the ascending link to use is obtained from the packet destination component corresponding to the stage at which the switch is located at. A given packet can only be forwarded upwards through that up link. To obtain the proper component from the destination identifier corresponding to the switch stage, we use the Mask Register (MR). The MR of each ascending link sets to 1 the bits corresponding to the component associated to the switch stage. Figure 4.10 shows the FIR register configuration for a 2–ary 3–tree. As k is 2, the components of the destination address are composed by only one bit. In the figure, each output port shows three lines of numbers: the first one (in red italic) shows the destination nodes that are reachable through that output port, the second one shows the values of the LIB and UIB registers of that output port, and the third one shows the value of the MR associated to the output port. As it can be seen, the MR is set to 001 in the up links of all the switches of the first stage, as only the least significant bit is selected and compared with LIB and UIB in this stage. Packets are forwarded through the ascending output port depending on the least significant component of its destination. For this, the LIB and UIB registers are set to their order inside the up links of a switch. In other words, the LIB and UIB of the first up link are set to route packets whose component that correspond to the stage at where the switch is located at is 0, and the ones that belong to the second up link of the switches only route packets whose component is 1. At the second stage, the next bit or component is considered, so MR is set to 010, but LIB and UIB values are equal to the ones from the first stage shifted one component to the left. The same procedure is used for filling the routing intervals of the switches from the rest of stages. In k–ary n–trees with $k > 2$, the components will have more than one bit and, thus, more than one bit will be set to 1 in the MR to select the component corresponding to the stage at which the switch is located at.

Descending links have the same values stored in LIB and UIB than the ones used with adaptive routing in Section 3.1, since the path reduction is only done in the upwards phase. As the MR is not used in the downwards phase, we set to 1 all the bits belonging to a down link, in order to select all the bits in the destination address. Notice that, despite the same downwards paths valid in the adaptive routing case are also valid in the deterministic case, only one is actually used because the packets only reach one of the nearest common ancestors switches between source and destination during the upwards routing phase.

In Figure 4.10, the reachable destinations through the descending links of a switch are also included in the routing intervals of the up ports. For instance, destination 0 in switch 0 can be reached through link 0 and link 2 according to the information obtained from their LIB, UIB and MR registers. So, packets destined to nodes that should only be reachable through the down links of the switch could be incorrectly forwarded through the upwards links. To avoid this problem, the RRR register is used to give preference to the

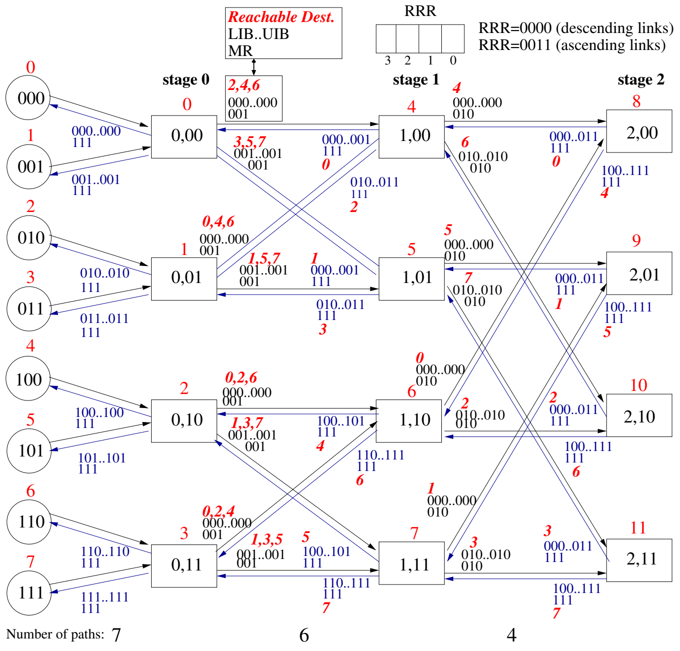


Figure 4.10: *DESTRO* routing in a 2-ary 3-tree with FIR registers.

descending links over the ascending ones and guarantee a minimal path. In the RRR, the half lowest significant bits correspond to the descending links and the half most significant ones to the ascending links. Therefore, in the ascending links (links 2 and 3) the RRR stores 0011, to give preference to links 0 and 1 over the ascending links. In this way, as an example, when routing a packet to destination 0 ⟨000⟩ at switch 0, both output ports, 0 and 2, may be allowed, since this destination, after being masked, is included in the intervals associated to both output ports. In link 0, LIB and UIB are equal to 000, as the MR of this port is 111, we compare the whole identifier of the destination with the routing interval. As it can be seen, 000 is included in the routing interval, thus link 0 can be used to forward packets to destination 0. On the other hand, in link 2, LIB and UIB are also set to 000. However, MR is set to 001. Hence, we just compare the least significant component of the destination identifier with the routing interval, and the destination is again included on it. Nevertheless, as output port 2 gives preference to output port 0, output port 2 is not finally returned. As commented above, we use here the RRR

```
RRR=$\{0\}^{k}\{1\}^{k}$
stage s
    (* selection of the bits of the packet destination corresponding to the current stage *)
    MR=$\{0\}^{n*r-(s+1)*r}\{1\}^{r}\{0\}^{s*r}$
    link L
    LIB..UIB=$\{0\}^{n*r-(s+1)*r}$ L−k $\{0\}^{s*r}$ .. $\{0\}^{n*r-(s+1)*r}$ L−k $\{0\}^{s*r}$
        (* the bits of the registers corresponding to the current stage are given by
           the link identifier *)
        (* they must be equal to L−k, being represented by r bits *)
        being
            k = arity of the switch
            n = number of stages
            r = log(k); (* bits used by each stage in the destination addresses *)
            n*r = bits in destination identifiers
```

Figure 4.11: Register configuration in the ascending links with deterministic routing.

register not to provide deadlock-freedom, we use it to restrict the possible paths provided by the masked routing interval, therefore providing minimal routing.

Figure 4.11 presents an algorithm to configure the FIR registers of the ascending links. First, the algorithm sets the half least significant bits of the RRRs to 1 and the half most significant bits to 0, to give preference to the descending links. Following, the algorithm sets the MR registers. MRs are always set as a sequence of zero or more zeros, at least a one, and a final sequence of zero or more zeros. Notice that, with $k \neq 2$, the MRs will have as many bits set to 1 as the bits needed to represent a component in the destination address, that is $log(k)$ bits. These 1s must be located at the position corresponding to the switch stage. For this, these ones are displaced $s \times log(k)$ bits to the left, s being the stage of the switch. The rest of the MR will be all set to 0.

Finally, the LIB and UIB registers of an ascending link (L) will select the destinations that have the identifier of the ascending link in the position of the component given by the switch stage (s in Figure 4.11). Since ascending links are labeled from k to $2k-1$, the value in the destination identifier component will be $L-k$, L being the link identifier. The configuration for the descending links is not shown because it is very simple. The LIB and UIB registers are the same as the adaptive routing case, and the MR is set all to 1s to select all the bits in the destination address for being compared with the LIB and UIB registers. The RRR is set to all 0s, since no preference is given to the other output ports in the down links of the switches.

Despite that we actually use this implementation, *DESTRO* can be implemented by using any routing scheme. It can implemented by using forwarding tables, hardwired specific logic, source routing, centralized routing, and so on.

4.3 Evaluation

In this section, we perform the evaluation of *DESTRO* focusing mainly on network performance. To perform this evaluation, we compare our deterministic routing algorithm with the adaptive routing algorithm commonly-used on fat-trees with different selection functions and with the deterministic routing algorithm proposed in [78]. First, we perform this evaluation without enforcing packet in-order delivery. However, as packet in-order delivery is mandatory on several systems, we also perform the comparison by enhancing the network with a reordering mechanism that preserves packet delivery order.

4.3.1 Adaptive Routing Issues

In adaptive routing, each time that a switch routes a packet, after applying the routing function, the selection function has to choose which of the routing options provided by the routing function is going to be actually used. With the proposed implementation (Figure 3.1), the routing function stores all the possible output ports in the AR register, and from this register, the selection function chooses the output port that will use the packet that is being routed. The selection functions can be classified in two main groups: the ones that consider the status of all the possible output links to choose the final output port, and the ones that prioritize one output port and depending on the status of this output port they use it or use another possible output port. The selection functions from the first group try to equitably use the output ports of the switches, while the ones from the second can unbalance the utilization of the network links, creating bottlenecks, if they are not designed carefully. However, as the selection functions from the first group require the global status of the switch to take their decision, their implementation is more complex. That is, they have a higher implementation cost and may be slower than the ones from the second group.

The selection functions of the second group are implemented in two steps, as it can be seen on Figure 4.12. The first step calculates the preferred output port for the packet that is being routed. This step is usually performed in parallel with the routing function, therefore, this step does not slow down the switch. The second step takes the result from the first one and the results of the routing function stored in the AR register and performs the selection of the output port. This logic first check if the preferred output port is available, that is, it is included in the AR register, and then checks if this output port meets the criteria for selecting the output port. This criteria may be implemented in several ways, for example, it can check if the available space in the buffer of the port or the number of credits is higher than certain threshold, or the last time that the port was used, or any other criteria that can be built up from the switch status. Independently of the used criteria, if the preferred output port meets it, then it is chosen. On the other hand, if the preferred output port does not match it, the second step tries to use another output port from the AR that meets it. This is usually implemented by means of a cyclic rotator that begins to search in the next output port to the preferred one till it completes a round or till it finds a suitable output port. For example, let's suppose that the preferred output port for a given packet is the port number i. When the second step checks the availability of this port, it notices that the port is not free as it is being used by another packet. Then, the cyclic rotator begins to search from the port $i + 1$ for a suitable output port.

As commented, we are going to compare our deterministic routing algorithm against the adaptive routing algorithm based on up/down that is commonly used on fat-trees. In adaptive routing, it is necessary to use a selection function. In order to perform a fair comparison, we are not going to compare our deterministic routing algorithm against a single selection function. Indeed, we perform such comparison against several selection functions explained below, some of which are presented and compared in [50]. All these selection

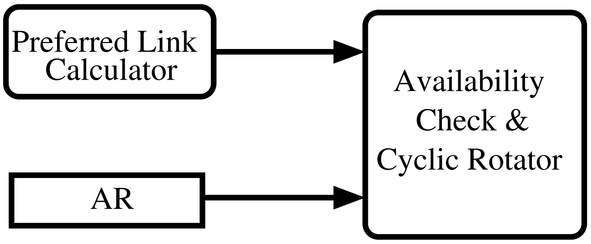

Preferred Link Calculator
AR
Availability Check & Cyclic Rotator

Figure 4.12: Basic implementation of a selection function with prioritization.

functions, but the last one, can be implemented as indicated above. That is, they can be implemented by a circuit that calculates the preferred choice and a rotative search starting in that link:

- *First Free (FF).* The FF selection function selects the first physical link which has free space. It uses a linear search, starting at the first ascending physical link (the k^{th}, link according to our notation). In other words, the preferred link is always the link labeled as k.

- *Static Switch Priority (SSP).* At a given stage of the fat-tree, the SSP selection function assigns the highest priority to a different ascending link at each switch of that stage. So, a given switch has always the same preferred link, regardless the packet destination. The idea is to create a disjoint high priority ascending path for each switch of the first stage. Hence, packets coming from different switches at the first stage will reach different switches at the last one, thus balancing the traffic in the upwards phase. The highest priority physical link for the switch $\langle s, o_{n-2}, ..., o_1, o_0 \rangle$ is the ascending link labeled as $k + o_s$.

- *Static Destination Priority (SDP).* The SDP selection function assigns priorities to physical links at each switch depending only on the packet destination. The preferred physical link is given by the least significant component of the packet destination. That is, a packet sent to processing node $p_{n-1}, ..., p_1, p_0$ has as the preferred link the $k + p_0$ link in all the stages of the fat–tree.

- *Static Origin Priority (SOP).* The SOP selection function assigns priorities to physical links depending only on the packet source. The preferred physical link is given by the least significant component of the packet source. That is, for a packet sent from processing node $p_{n-1}, ..., p_1, p_0$, it is $k + p_0$.

- *Cyclic Priority (CP).* The CP selection function uses a round robin algorithm to choose a different physical link each time a packet is forwarded. In this case, the preferred link is the link that was provided by the selection function the last time it was executed.

- *Stage And Destination Priority (SADP).* The SADP selection function takes into account both the stage at which the switch belongs to and the component of the packet destination corresponding to that stage, (i.e., a switch located at stage s considers the s^{th} component of the destination address). That is, at the switch $\langle s, o_{n-2}, ..., o_1, o_0 \rangle$, the highest priority physical link for a packet with destination $p_{n-1}, ..., p_1, p_0$ will be $k + p_s$. As it can be seen, this selection function has as preferred link the link that would be used in our deterministic routing algorithm.

- *More Credits (MC).* Since we use credits to implement the flow control mechanism, the MC selection function selects the link which has the highest number of credits available. MC is the most complex one, as it needs several comparators to select the link with more available credits.

- *Adap-Lin.* We also compare our proposal against the selection function resulting from using as preferred link the one that would be used by the deterministic routing algorithm proposed in [78], explained in Section 2.2.2. That is, the highest priority physical link for a packet with origin $p_{n-1}, ..., p_1, p_0$ will be $k + p_s$.

Notice that the preferred links for all the selection functions are only up links, because selection functions are only used in the upwards routing phase, since the downwards one is deterministic.

Additionally, when in-order delivery is required, we have implemented a basic approach to enforce in-order delivery with adaptive routing algorithms. It is similar to the one proposed in [86]. This approach uses a reorder-buffer (different buffer sizes have been used) at the destination node Network Interface Card (NIC) to store packets that have arrived out of order. Every time a packet is sent, its sequence number is included in its header. When a packet arrives out of order at the destination, it is stored in the reorder-buffer to wait for all packets with smaller sequence number. If a packet is received out-of-order and it has no free space in the reorder-buffer, it should be discarded and retransmitted. On the other hand, to prevent these unnecessary packet re-transmission, the source node does not inject packets if the destination buffer does not have enough free space to store all the packets that have been sent previously and have not been delivered to the processing node. For this, an end-to-end flow control mechanism is implemented.

As it can be seen, guaranteeing in-order delivery of packets complicates the adaptive routing implementation, as a reorder-buffer is required at the destination and an end-to-end flow control is needed. Our implementation of the reorder mechanism for adaptive routing is ideal, as it does not consider any reordering costs. For example, we assume that the acknowledge control messages do not consume network bandwidth and cannot be affected by packet contention or network congestion; the bandwidth between the reorder buffer and the node is unbounded, thus the reorder buffer can forward any number of packets to the node in just one clock cycle; the source nodes know the available free space of all the reorder buffers of all the other nodes without any access time and without consuming network bandwidth. Thus, the obtained performance results could be considered as optimistic and the differences that we obtained in the results are produced only by the delay introduced by out-of-order arrived packets, not by the reordering mechanism itself. Notice that deterministic routing do not require the use of additional mechanisms, since packet delivery order is preserved by design.

4.3.2 Traffic Patterns

In order to make a deep analysis of our proposed routing algorithm, we have performed the evaluation using a wide range of traffic patterns. The used traffic patterns can be classified in two categories: synthetic traffic patterns and I/O traces.

Synthetic Traffic Patterns

In the evaluation with synthetic traffic patterns, four different traffic patterns have been used: uniform with and without hot-spot traffic, bit-reversal and complement traffic.

- In uniform traffic, message destination for each message is randomly chosen among all the processors.
- In hot-spot traffic, a percentage of the total traffic of the network is sent to a single node and the rest of the traffic is distributed like in the uniform traffic pattern.
- In bit reversal traffic, each node sends packets to the destination obtained by taking its components but in reverse order. That is, the node whose identifier is $\langle 100 \rangle$ would send all its packets to node $\langle 001 \rangle$.
- In complement traffic, each processor sends all its messages to the opposite node. Thus, in a network with N processors, the processor i sends messages to the processor $N - i - 1$. The complement traffic pattern has two interesting properties in fat-tree networks. The first one is that all the packets have to reach the upper stage in order to arrive to their destination, hence, the up link selection mechanism must be applied several times. The second one is that each processor node only sends messages to one destination, so the network should not be congested due to hot-spots.

Packet size has been fixed to 8 KB for all the traffic patterns.

We have evaluated a wide range of k–ary n–tree topologies, k being 2, 4, 8, 16 and 32 and n being up to 8. For the sake of simplicity, we show here only a subset of the most representative simulations.

I/O Traces

In order to analyze the performance of our proposal using a more realistic traffic pattern, we have used a set of I/O traces provided by Hewlett Packard Labs. They include I/O activity generated in the early 1999 at the disk interface of the cello system. The cello system is a timesharing system with a storage subsystem of 23 disks. They provide information for the requests generated by the hosts and the answers generated by the disks. A detailed description of similar traces of 1992, collected in the same system, can be found in [110]. We will use packets with a payload equal to the size specified in the trace for the I/O accesses, but if the message is larger than 8 KB, we split it into several packets generated at the same time with a payload of at most 8 KB.

In the evaluation with traces, we have simulated a 2–ary 7–tree. As the *cello* system, it has a storage subsystem with 23 disks that we have attached to 23 randomly selected leaves of the 2–ary 7–tree. The remaining 105 leaves have been used to attach processing nodes.

4.3.3 Simulation Environment

To evaluate our deterministic routing algorithm and compare it against the routing algorithms and selection functions proposed above, a detailed event-driven simulator has been implemented. The simulator is a modification of the base simulator used in most of the works of the Parallel Architecture Group (GAP [6]) of the Technical University of Valencia (Universidad Politécnica de Valencia), as the one used in the previous chapter.

The used simulator models a k-ary n-tree with FIR routing and virtual cut-through switching. Each switch has a full crossbar with queues of two packets both at the input and output ports. We assumed that it takes 20 network clock cycles to apply the routing algorithm; switch and link bandwidth has been assumed to be one flit per network clock cycle; and fly time has been assumed to be 8 network clock cycles. We apply these values because they were used to model Myrinet networks in [44]. Finally, the simulator uses credit-based flow control. When in-order delivery is required, the simulator uses the in-order delivery mechanism proposed above.

The simulator allows to work with auto-generated traffic patterns or with external traces. If the simulator is working in trace mode, it generates the packets at the times and with the sizes indicated in the used trace file. However, if the simulator is working with synthetic traffic then the simulator generates the packets on its own. To do this, the simulator allows to change the traffic pattern with all the required parameters for each traffic patter, if any, and the average injection rate. In this way, we can simulate several traffic loads with several traffic patterns.

4.3.4 Performance Results

In order to perform the performance evaluation of our proposed deterministic routing algorithm, we compare it with the aforementioned adaptive routing algorithms with and without preserving the delivery order of packets. When in order delivery of packets is not preserved, we want to analyze the relative performance of *DESTRO* against the best adaptive routing algorithms, taking only into account the selection function alone. We also want to analyze the relative performance of *DESTRO* versus the best adaptive routing algorithms for fat-trees when packet in-order delivery is a must, and the impact on performance of enforcing in-order delivery of packets in adaptive routing algorithms.

Results without enforcing in-order delivery of packets

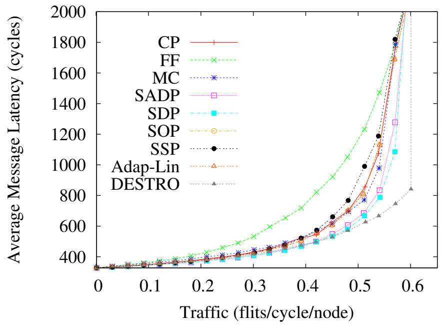
First, we compare *DESTRO* with the adaptive routing algorithm commonly used in fat-trees with all the aforementioned selection functions. First, we perform the comparison by using only synthetic traffic patterns. In this way, we want to explore the impact of network size, varying the k and n parameters of k-ary n-trees for different traffic patterns [33, 40]. Next, we compare the different routing algorithms with a more realistic traffic pattern by using the aforementioned traces.

Synthetic Traffic Patterns. Figure 4.13 shows the performance results for several networks with $k = 4$ and $k = 8$ for the uniform traffic pattern. Figure 4.13(a) presents the average packet latency versus traffic for a very small network (a 4-ary 2-tree that has 16 nodes). The behavior of the selection functions is not very different, with the exception of FF, which achieves a higher average latency than the other selection functions. FF always returns the same preferred ascending link, therefore it saturates an ascending path before selecting another one. Hence, FF tends to make an unbalanced link utilization, as those links that belong to the preferred ascending paths always have a higher utilization than the others, resulting in a larger latency.

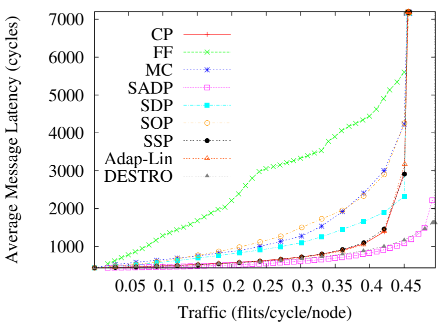
On the other hand, due to the fact that there are only 2 stages in the network, the rest of selection functions have almost the same performance, because there is a low number of different paths that can be chosen to reach any destination. In fact, with only two stages, the selection function is only performed once. Indeed, it is not performed for all the packets, it is only performed for the packets that require to travel upwards in the first stage. Therefore, the influence of the selection function for networks with such a small number of stages is heavily lessened. A similar behavior can be observed in Figure 4.13(d) where a network with a larger k and two stages is analyzed. Nevertheless, in this case, the differences in network latency are larger. Notice that, with k equal to 8, there is a higher percentage of packets that require to go up to the last stage of the network, so the influece of the selection function is higher than in a network whose k is 4. Thus, SADP and SDP obtain a higher throughput than the other selection functions.

Notice that with two stages SOP and Adapt-Lin provide the same routes for all the packets, so the results for such selection functions are almost the same, the same happens with SDP and SADP. Concerning *DESTRO*, the results are very positive, since it obtains a slightly higher throughput than the adaptive routing algorithms. Moreover, at the highest injection rates, average packet latency in *DESTRO* is lower than the one from adaptive routing algorithms.

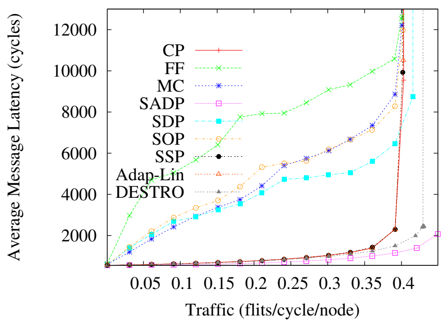
Figures 4.13(b) and 4.13(c) show results for a higher number of stages (4 and 6) and $k = 2$. As it can be seen, the more stages in the network, the differences among the average packet latency of the different selection functions get more significative. This is due to the fact that, with more stages, there are more different ascending paths to choose from. Therefore, there are more opportunities to balance traffic since the selection function is applied more times. Despite SOP, MC and SDP achieve a better performance than FF, they still have a high network latency due to the fact that they do not balance link utilization as much as possible. Moreover, since they select paths based only on the p_0 component of the packet origin (SOP) or destination (SDP) or on the number of available credits at the output ports, the probability of obtaining disjoint paths for different destinations and, thus, retaining the



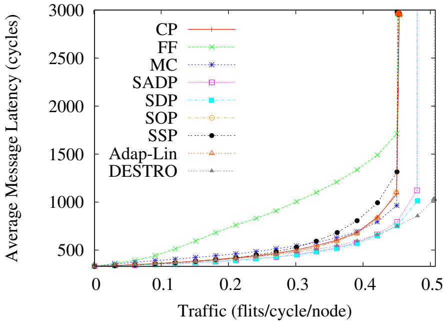
(a) 4-ary 2-tree.



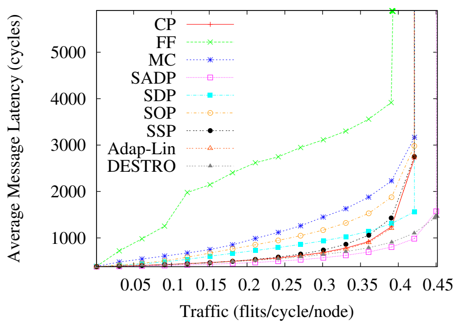
(b) 4-ary 4-tree.



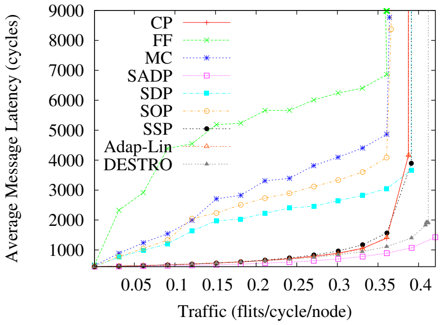
(c) 4-ary 6-tree.



(d) 8-ary 2-tree.



(e) 8-ary 3-tree.



(f) 8-ary 4-tree.

Figure 4.13: Average packet latency versus accepted traffic for uniform traffic.

congestion due to congested nodes is very low. Notice that the difference of SDP and SOP with respect to FF is that FF always prioritizes the same output port for all the packets, so this port tends to be overloaded, whereas in SSP and SDP, the preferred link depends on the packet destination or origin, so they are more prone to distribute the packets between the ascending links of the switches than FF. On the other hand, CP and SSP achieve almost the same performance. However, in these selection functions, there is not any mechanism that tries to avoid that the ascending paths of different destinations have disjoint links to retain congestion. Adap-Lin obtains also similar latencies that the previous selection functions, but SADP achieves the best performance because, assuming that the preferred ascending path for each packet is free, the links shared by different destinations are as few as possible.

For this same reason, our deterministic routing algorithm can achieve almost the same performance than SADP. *DESTRO* classifies the packets according to their destination, concentrating packets to the same destination in the same links and only in those links. In this way, when the network reaches saturation, packets to different destinations do not have to compete between them, reducing the large penalization on latency provoked by the HOL blocking effect.

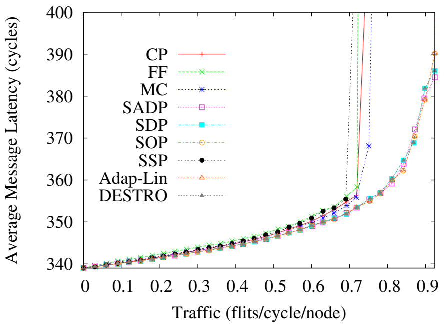
We have also analyzed the impact of the arity (k) of the fat-tree on performance and we have observed that the number of stages (n) is the most important parameter in the differences among the studied routing algorithms. In the case of the selection functions, this is due to the fact that the impact of any selection function is greater when there is a higher number of stages, since it is applied more times as packets need more hops to reach their destination. Figures 4.13(d), 4.13(e), and 4.13(f) show results for three networks with $k = 8$. Comparing the figures with $k = 4$ and the ones with $k = 8$, it can be observed that increasing k tends to increase the differences among selection functions, but the relative behavior remains the same. Notice that the best adaptive routing algorithm (SADP) tends to obtain a slightly higher throughput than *DESTRO* as the number of stages is increased, that is, as the number of different routing options is increased. However, notice that *DESTRO* obtains a higher throughput than most of the adaptive routing algorithms despite being a deterministic one. Furthermore, SADP only gets

6,25% higher throughput in the best case, which is not a big difference, making *DESTRO* competitive in terms of throughput. In terms of latency, *DESTRO* always has a higher latency than the best adaptive routing algorithm for low and medium injection rates, but has a lower latency for high injection rates in most of the cases. This can be explained by the fact that in low and medium injection rates the adaptivity of the adaptive routing algorithms allows packets to be deviated from more used paths to less loaded paths; however, for high injection rates, this adaptivity suppose a drawback since there is not any path that is not heavily loaded, increasing in this way the effects of HOL blocking.

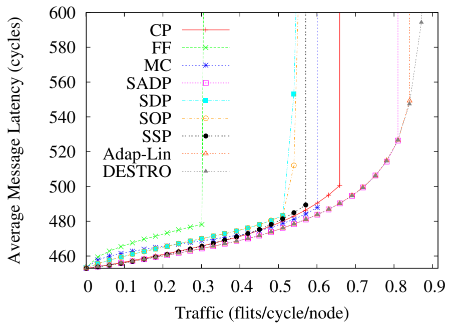
Figure 4.14 shows the performance of the selection functions for the same networks, but using the complement traffic pattern. Concerning the selection functions, the results remain similar to the ones obtained with the uniform traffic pattern. Again, FF has the worst performance. But, in this case, Adap-Lin and SADP have the best performance. Both selection functions have a similar behavior with complement traffic because each source node only sends packets to the same destination node. Remember that Adap-Lin tries to have disjoint paths for different source nodes and SADP tries to do that for different destinations, which for this traffic pattern is the same.

Concerning *DESTRO*, it obtains similar results than SADP, which can be easily explained since SADP tries to use the same routes than *DESTRO*. Taking into account that a source only sends packets to a single destination, both algorithms classify the packets avoiding that packets to different destinations collide. However, when the injection rate is close to the saturation point of the network, SADP begins to use alternative routes to forward packets, since it is an adaptive routing algorithm, which makes that the number of collisions in the network completely saturates the network due to the high injection rates. Nevertheless, *DESTRO* keeps using the same routes in this situation, contenting the congestion, and outperforming the adaptive routing algorithms in all the studied topologies. In the best case, *DESTRO* obtains 10,7% higher throughput than the best adaptive algorithm. Notice that with complement traffic the number of routing options tends to reduce the performance of adaptive routing algorithms due to the effects of HOL blocking, so *DESTRO* outperform them as we increase k or n.

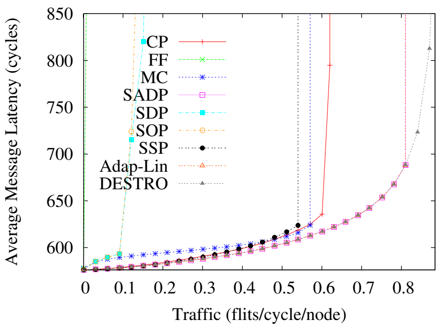
As we have evaluated a traffic pattern that unintentionally favors *DE-*



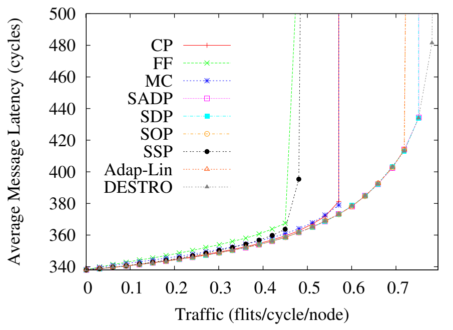
(a) 4-ary 2-tree.



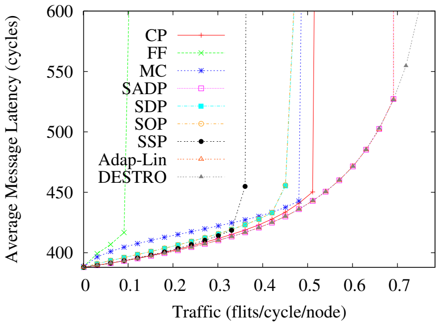
(b) 4-ary 4-tree.



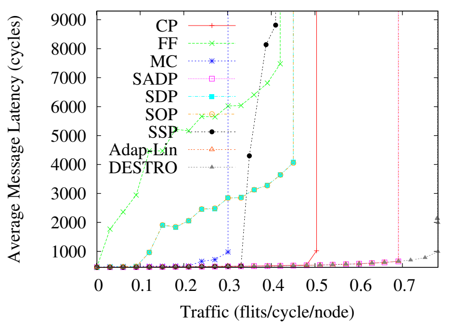
(c) 4-ary 6-tree.



(d) 8-ary 2-tree.

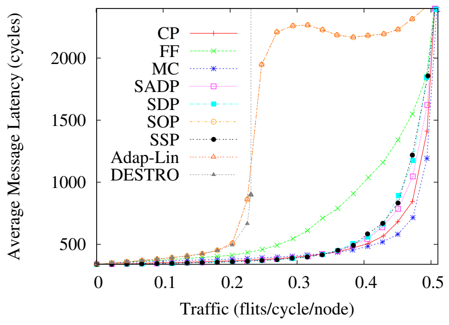


(e) 8-ary 3-tree.

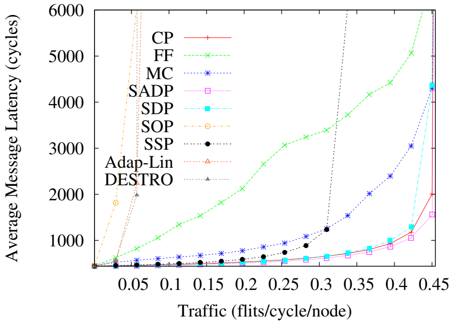


(f) 8-ary 4-tree.

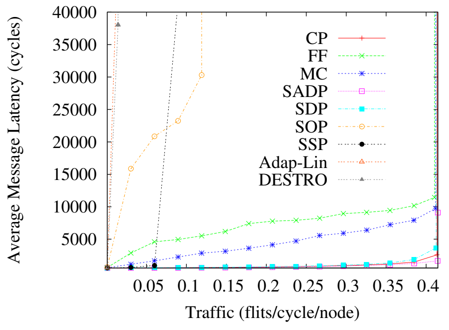
Figure 4.14: Average packet latency versus accepted traffic for complement traffic.



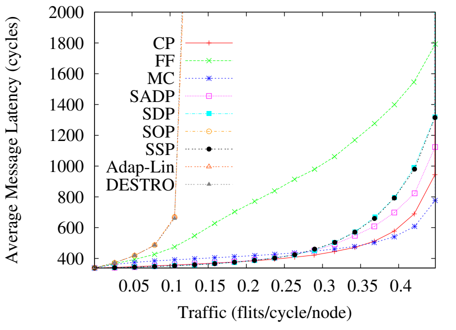
(a) 4-ary 2-tree.



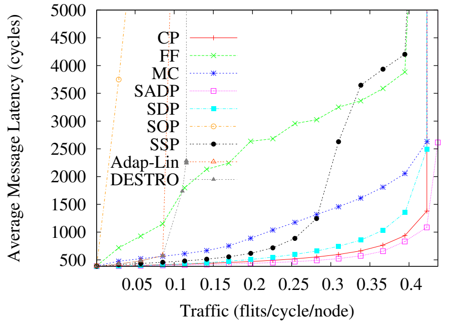
(b) 4-ary 4-tree.



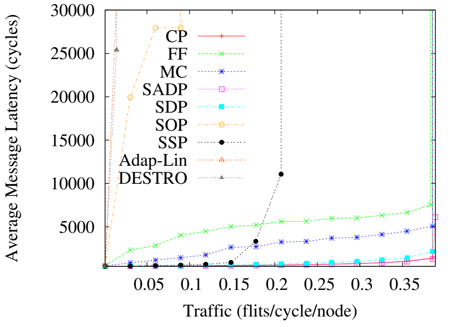
(c) 4-ary 6-tree.



(d) 8-ary 2-tree.



(e) 8-ary 3-tree.



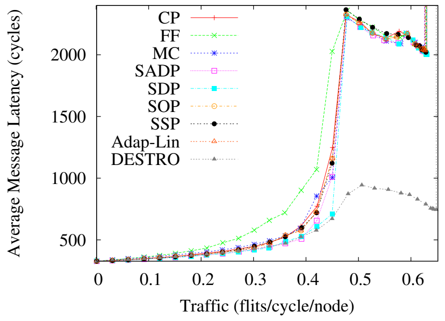
(f) 8-ary 4-tree.

Figure 4.15: Average packet latency versus accepted traffic for bit reversal traffic.

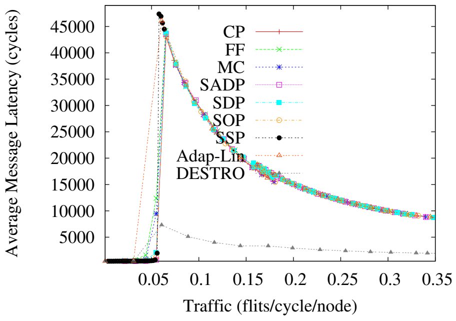
STRO, we want to analyze the worst traffic pattern for *DESTRO*. In that traffic pattern, messages from all the nodes attached to the same switch of the first stage are sent to processing nodes whose id has the same least significant component. For instance, in a 2-ary 2-tree, if all the nodes from a switch want to send their packets to destinations that have the last component equal to 0, the possible destinations should be 0 and 2. For this traffic pattern, the effective bandwidth is half the total bandwidth with our deterministic routing, because only the links labeled as 2 (k in general) will be used in the ascending phase. In a general case, the effective bandwidth of the network would be $1/k$, since only one up link of the switches of the first stage would be used. An example of such traffic pattern is the bit reversal traffic pattern. For instance, in a 8-ary 3-tree with bit reversal traffic, the nodes attached to the switch 0, nodes $\langle 0,0,0\rangle$, $\langle 0,0,1\rangle$, $\langle 0,0,2\rangle$, ..., $\langle 0,0,7\rangle$, send all their messages to nodes $\langle 0,0,0\rangle$, $\langle 4,0,0\rangle$, $\langle 2,0,0\rangle$, ..., $\langle 7,0,0\rangle$, respectively. As it can be observed, all the destinations share the two last components, so packets will share the same link during all their ascending paths. So, the performance of *DESTRO* is expected to very low compared to the adaptive routing algorithms.

Figure 4.15 shows the performance of the same networks than before with bit reversal traffic. As it was expected, the performance of the deterministic routing algorithm is poorer than the one obtained by all the adaptive algorithms, since it only uses one of the k up links of the switches of the first stage. However, as it can be seen, the two selection functions that define their preferred link based on the source node of the packets (SOP and Adap-Lin) obtain also a lower performance than the other selection functions. Bit reversal for this selection functions behaves similar to a hot-spot traffic pattern obtaining that they can not exploit the adaptivity of the network since most packets get congested in the lower stages of the network.

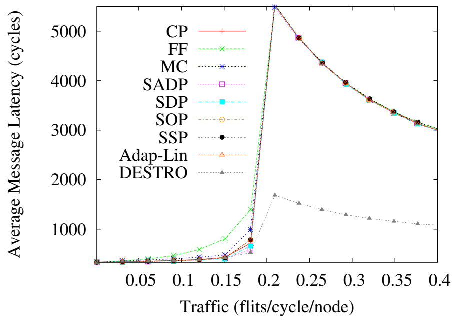
Finally, we have also analyzed a traffic pattern that negatively affect all the routing algorithms: the hot-spot traffic pattern. In Figure 4.16, we present the performance results for some networks where a hot-spot receives the 5% of all the packets of the network. As it can be seen, *DESTRO* is able to cope better with congestion since it provides a much smaller latency and slightly higher throughput. We can see that the network suffer two saturations. For instance, in Figure 4.16(a), at 0.48 flits per cycle per node, the hot-spot get saturated



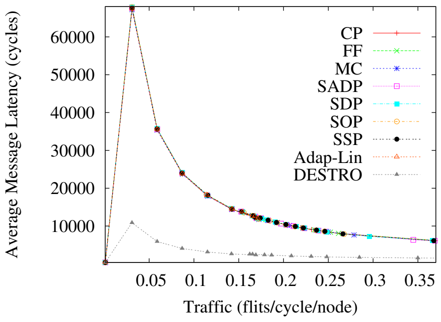
(a) 4-ary 2-tree.



(b) 4-ary 4-tree.



(c) 8-ary 2-tree.



(d) 8-ary 3-tree.

Figure 4.16: Average packet latency versus accepted traffic with a 5% hot-spot.

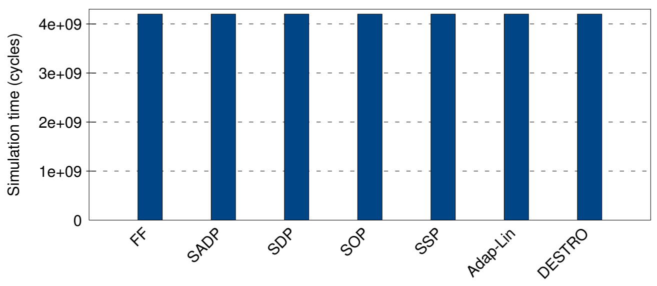
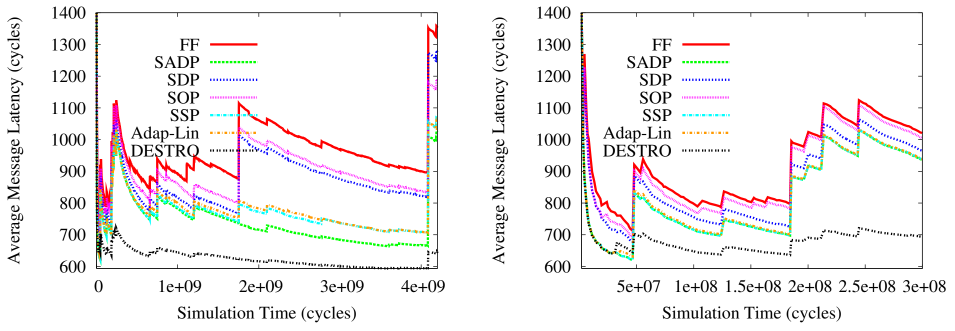


Figure 4.17: Simulation time in cycles for each routing algorithm using the same trace.

and it can not absorb more packets, but the network can still deliver packets to other nodes of the network till the point at 0.6 flits per cycle per node where the network gets completely saturated. Between these two saturations, *DESTRO* strongly diminishes the average packet latency by a factor of 2,5 on average, which remarks the capability of *DESTRO* to content the congestion without spreading it all over the network.

I/O Traces. In order to provide a more exhaustive comparison between the studied routing algorithms, we have also performed several simulations with traffic traces from the disk interface of the cello system, as commented above. For this, we first compared the simulation times for all the routing algorithms, as this time gives the time required to deliver all the injected packets. Figure 4.17 shows the simulation time that each routing algorithm required to complete a single trace. As it can be seen, most of the algorithms take almost the same number of cycles to completely deliver all the messages in the trace with just small differences of few clock cycles. Considering these results, all the routing algorithms seem to be equally efficient.

However, analyzing the trace file, we observed that the used trace has a bursty behavior till near the end. The last part of the trace is composed of several small packets very spaced on time, which could be hiding the inefficiencies of some of the routing algorithms. For this, we have also measured the average packet latency from generation for all the routing algorithms. These results are presented in Figure 4.18(a). As it can be seen, considering the av-



(a) Full trace. (b) Beginning of the trace.

Figure 4.18: Evolution of the average packet latency from generation for all the routing algorithms with a trace.

erage packet latency brings to light that there actually are differences among the routing algorithms. As it was expected from the results with synthetic traffic, FF is the worst selection function when using adaptive routing, and SADP is the best one, closely followed by SSP and Adap-Lin, that get an average latency from generation that is 47 clock cycles higher than the one from SADP. The important point of the figure is that *DESTRO* is the routing algorithm with the lowest average latency from generation, despite the fact that it is a deterministic routing algorithm that tends to content the packets in the nodes when congestion arises.

The reason of this low latency can be seen in Figure 4.18(b) that presents the average packet latency from generation zooming on the beginning of the trace. As commented, and as it can be seen in the figure, the trace is very bursty. The bursts correspond to the peaks on the latency in the figure. In the beginning, all the routing algorithms can deal with the burst, even *DESTRO* shows a higher latency than SADP, Adap-Lin and SSP. Nevertheless, when the next burst is injected into the network, the adaptive algorithms start to spread the packets of the burst all over the network, congesting it, highly increasing the latency of the packets. This effect is even more pronounced with each consecutive burst. But, *DESTRO* does not suffer from this effect. As it can be seen, *DESTRO* contents the congestion in the nodes without affecting the network. In this way, it can deal with the bursts while achieving a low packet latency. Of course, each burst increases the average packet latency, but *DESTRO* is able of lowering the latency before the next one is produced.

However, the reader may think that the results from the total simulation time and the average packet time may be inconsistent, since a higher packet latency would delay packets and this would likely make that the simulation time should be larger since a disk node cannot answer a request that it has not yet received through the network, for example. Nevertheless, the used trace files do not provide any information about the correspondence of the packets, so we cannot simulate such request/answer behavior, and therefore, the simulator generates the packets at the times indicated by the trace without considering the arrival of any other packets. Therefore, total simulation time is not an accurate metric in this case.

Results enforcing in-order delivery of packets

As commented during this chapter, deterministic routing algorithms always guarantee that packets are delivered to their destination in the same order that they were injected in the network, which is mandatory for several applications. However, adaptive routing algorithms cannot provide in-order delivery of packets by themselves. They must use additional mechanisms like the one commented in Section 4.3.1. In order to perform a fair comparison, we have also analyzed the performance of the adaptive routing algorithms when in-order delivery of packets is guaranteed by means of a complementary mechanism, considering several buffer sizes for the reordering buffer of the mechanism.

As we did in the previous section, first, we compare the routing algorithms using synthetic traffic patterns, and, next, we analyze them using traces. However, for the sake of legibility, we will only consider the adaptive routing algorithm with FF and SADP selection functions. We have chosen these two selection functions since they are in general the worst and the best selection functions. In this way, we want to establish the boundaries between which all the selection functions would behave in the general case. Also, we want to directly compare *DESTRO* with the deterministic routing algorithm proposed in [78], that is going to we referred as DET-Lin.

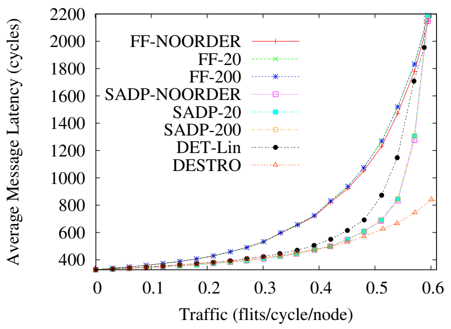
In the figures of this section, we refer as FF-NOORDER and SADP-

NOORDER as FF and SADP that do not guarantee in-order delivery of packets; FF-20 and SADP-20 stand for FF and SADP that guarantee in-order delivery of packets using a reorder buffer at each destination that can store up to 20 packets for each origin node; similarly, FF-200 and SADP-200 stand for FF and SADP with a reordering buffer at each destination that can store up to 200 messages for each origin node.

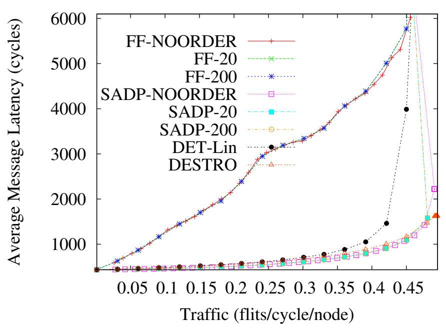
Synthetic Traffic Patterns. Figure 4.19 depicts the performance results for the aforementioned routing algorithms with uniform traffic, but guaranteeing the delivery order of packets. We could not perform the simulations of the largest networks since the reordering mechanism requires a high amount of memory, so we restricted our simulation to the small and medium-sized topologies. As it can be seen in the figure, guaranteeing the delivery order of packets by using the reordering-buffer mechanism does not seem to have a relevant impact on the performance, because uniform traffic pattern does not deliver packets out of order in low and medium traffic loads, it only tends to deliver packets out of order in high traffic loads. The adaptive algorithms that guarantee in-order delivery of packets have a slightly higher latency than the corresponding algorithm without in-order delivery of packets, but this latency increase is almost negligible in all the topologies.

Regarding DET-Lin, it can be observed that it outperforms FF in all the cases, but it can not achieve the performance of SADP. It has a lower throughput and a higher average latency no matter the load of the network. This results are very impressive from perspective, since DET-Lin is a deterministic routing algorithm whose performance is between the boundaries that define the best and worst case for adaptive routing algorithms. However, if we compare it with *DESTRO*, we can clearly see that *DESTRO* in the same scenarios can achieve the same performance than the best adaptive algorithm and in some networks, like the ones with two stages ($n = 2$), it can even outperform the adaptive routing algorithms while lowering the average packet latency.

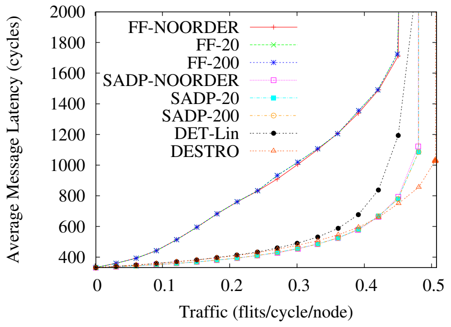
Figure 4.20 presents the performance results for the same networks but using complementary traffic. In this case, we start to see the impactof the reordering mechanism on the performance of the network, specially in Figure 4.20(d). In this figure, we can see that FF with in-order delivery of packets



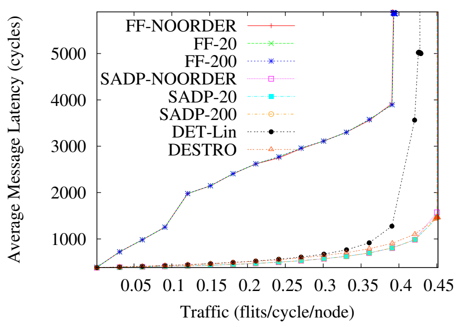
(a) 4-ary 2-tree.



(b) 4-ary 4-tree.

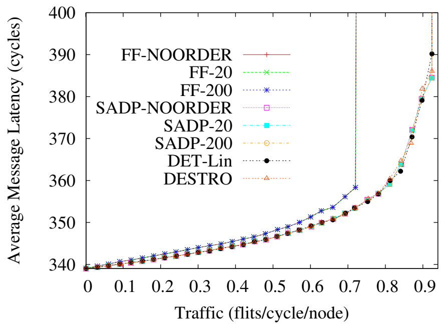


(c) 8-ary 2-tree.

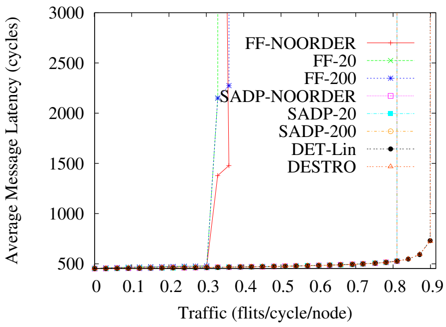


(d) 8-ary 3-tree.

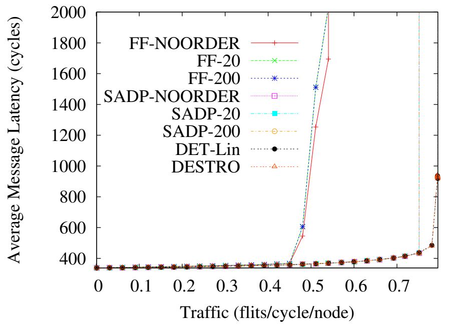
Figure 4.19: Average packet latency versus accepted traffic for uniform traffic guaranteeing in-order delivery of packets.



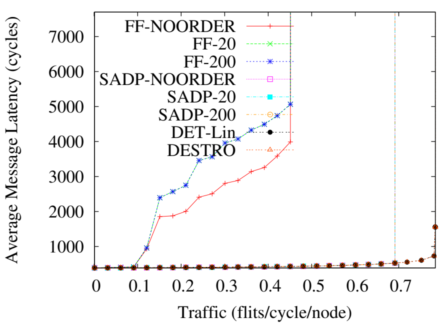
(a) 4-ary 2-tree.



(b) 4-ary 4-tree.



(c) 8-ary 2-tree.

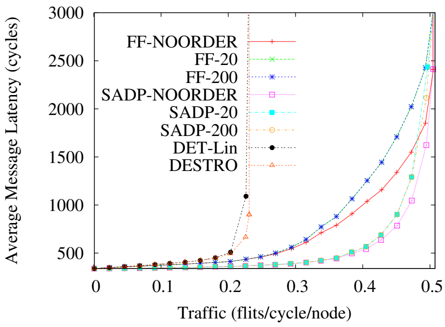


(d) 8-ary 3-tree.

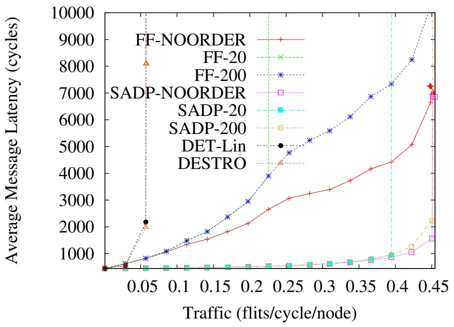
Figure 4.20: Average packet latency versus accepted traffic for complement traffic guaranteeing in-order delivery of packets.

obtains a higher average latency than the one that does not guarantee in-order delivery of packets. The increase in latency due to the arrival of out-of-order packets in some case is close to a 43.5%. However, as it can be see, SADP does not suffer any penalization for using a reordering mechanism. This different behavior is provoked by the fact that FF tends to overuse the same link no matter the destination of the packets. So, when this link gets congested, FF starts to scramble all the packets favoring that they reach their destination out of order. However, as SADP tends to classify packets depending on the destination, it requires to use a more stressing traffic pattern to start delivering most of the packets out of their delivery order.

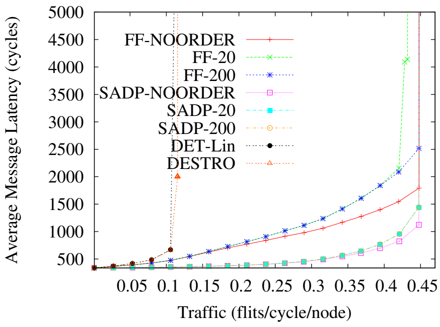
Regarding the deterministic routing algorithms, both obtain the same results since for this specific traffic pattern the routes provided by them are



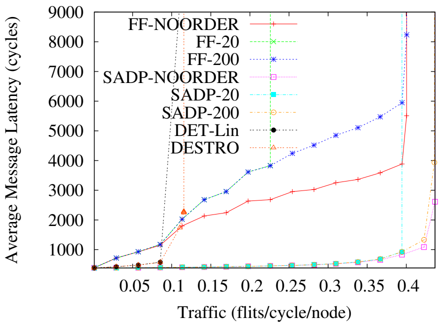
(a) 4-ary 2-tree.



(b) 4-ary 4-tree.



(c) 8-ary 2-tree.



(d) 8-ary 3-tree.

Figure 4.21: Average packet latency versus accepted traffic for bit reversal traffic guaranteeing in-order delivery of packets.

equivalent. In order words, as in the case in which in-order delivery of packets was not guaranteed, both deterministic algorithms outperform the adaptive ones. In the best case, the deterministic routing algorithms have an improvement of 14% in throughput over the adaptive ones.

Figure 4.21 shows the same results for bit reversal traffic pattern. As commented, *DESTRO* obtains a very low performance with this traffic pattern since it is its worst corner case traffic pattern. However, a positive point for *DESTRO* is that DET-Lin suffers from the same situation, obtaining similar results. In the worst case, the deterministic routing algorithms obtain a 89% lower throughput than the adaptive ones.

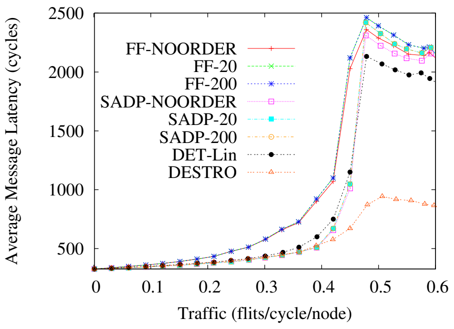
Another interesting point of these figures is that we can observe the importance of the reordering mechanism and the size of the used reordering-buffer.

For this traffic pattern, we can observe that the latency is increased when packet in-order delivery is ensured for all the topologies. Furthermore, this difference in latency is increased, as the network load rises. Moreover, we can see that the reorder-buffer size is important. In all the topologies, but the smallest one, we can observe that FF-20 obtains a lower throughput than the other configurations. This difference in throughput ranges from 4% lower throughput to 51% lower throughput.

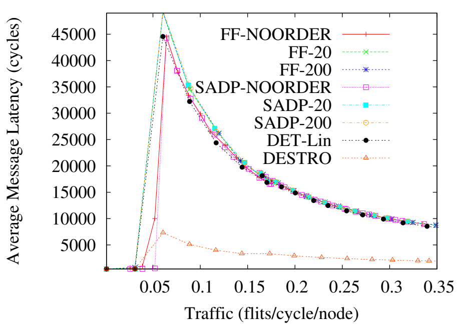
Concerning SADP, we can observe that it can preserve the delivery order of packets more efficiently than FF. However, we can observe that the SADP configurations that preserve the delivery order of packets have a slightly higher latency near saturation than the one that does not preserve the packet delivery order. Furthermore, in Figures 4.21(b) and 4.21(d), it can be observed that SADP-20, the one with the smallest reordering-buffer, can not achieve the same throughput than the SADP that does not preserve the ordering of the packets. Nevertheless, this lost of throughput is clearly smaller than the one suffered by FF. In the worst case, this lost of performance reaches the 12%.

Finally, Figure 4.22 presents the results for the same topologies, but having a hot-spot that concentrates the 5% of the total traffic of the network. In this case, the bottleneck is not the inefficiency of the routing algorithm at keeping the order of the packets, it is the ejection link that extracts the packets from the network to the hot-spot node. For this, there are not great differences between most of the routing algorithms, no matter if they preserve the ordering of packets, or they do not do it. We can observe that DET-Lin obtains almost the same performance than the adaptive algorithms, which confirms that DET-Lin spread the congestion of a single node to the whole network. In this way, most of the packets are blocked due to the effects of HOL blocking. As commented in the previous section, *DESTRO* is able to keep the congestion provoked by the hot-spot into the links that are assigned to that node, favoring that packets destined to other nodes are not heavily disturbed due to the hot-spot. For this, the latency after the congestion of the hot-spot is very small compared to the one observed in all the other routing algorithms.

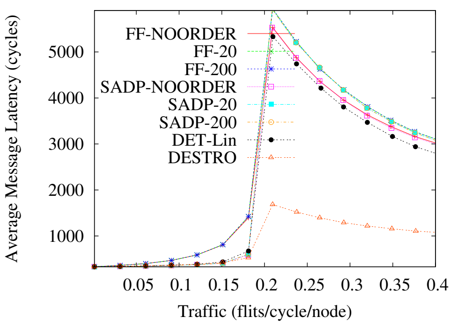
I/O Traces. Again, in order to provide more realistic results, we have performed the comparison of the analyzed routing algorithms by using traces.



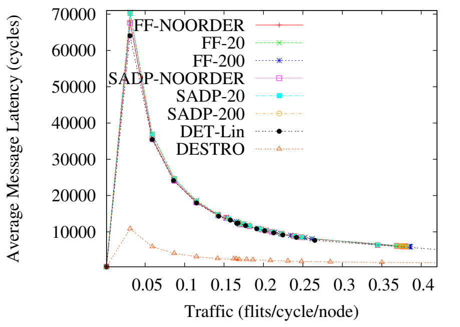
(a) 4-ary 2-tree.



(b) 4-ary 4-tree.



(c) 8-ary 2-tree.



(d) 8-ary 3-tree.

Figure 4.22: Average packet latency versus accepted traffic with a 5% hot-spot guaranteeing in-order delivery of packets.

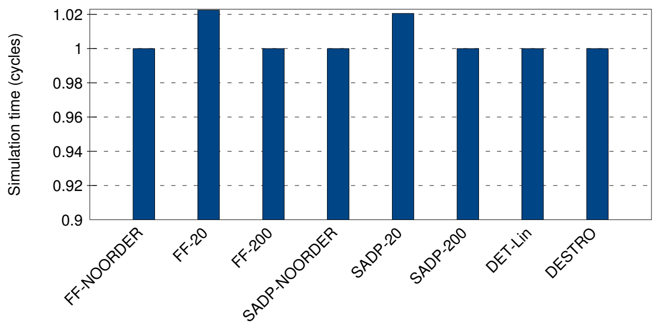


Figure 4.23: Simulation time in cycles for each routing algorithm using the same trace guaranteeing in-order delivery of packets.

Figure 4.23 shows the simulation time for the analyzed routing algorithms normalized to the simulation time of *DESTRO*. As it was expected, most of the configurations obtain the same simulation times due to the small disperse packets at the end of the trace, and the lack of relationship between packets in the trace. Nevertheless, we can observe that the configurations of both adaptive routing algorithms that preserve the delivery order of packets with the smallest reordering-buffer take more time than the other configurations to fully process the trace. Despite that these differences are very small (near 2% in both cases), it may indicate that we can expect a higher difference on average latency of packets.

Indeed, as it can be observed in Figure 4.24, the differences in average latency are very significative. Notice that the figure is using a logarithmic scale in the y-axis (average packet latency from generation). As the figure presents several significative points that should be commented, we want to start by analyzing the impact of preserving the order of the packets in the adaptive algorithms and then we will move to compare the different routing algorithms with *DESTRO*.

Concerning the adaptive algorithms, it can be seen in the figure that the bursty behavior of the traces has a great impact on their performance when in-order delivery of packets is required. For example, SADP-20 obtains worse results than FF without ordering restrictions. It even obtains a higher latency than FF-200 for the first half of the trace. Just to show the influence of the out-

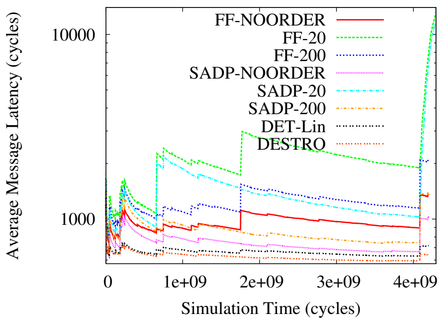


Figure 4.24: Evolution of the average packet latency from generation with a trace guaranteeing in-order delivery of packets.

of-order arrived packets, the FF configurations that preserve the ordering of the packets obtain a 54% higher average latency when using the large buffer, and a 947% higher latency when using the small buffer. In SADP, we can observe the same trend. When using the large reordering-buffer, SADP obtains a 37% higher latency than in the case when packet ordering is not preserved, and a 977% higher latency when using the small reordering-buffer.

Regarding *DESTRO*, it can outperform all the adaptive routing algorithm by using less resources since it does not require to use neither a selection function nor a reordering mechanism for out-of-order arrived packets. It obtains a 42% lower latency than the SADP that does not preserve packet ordering. Moreover, the latency obtained by the SADP configuration that preserves the order of packets and uses the small buffer is 16.46 times the one obtained by *DESTRO*; this difference is reduced to 1.16 times when using the large buffer. Finally, as it can be seen in the figure, *DESTRO* also obtains a significatively smaller latency than DET-Lin due to its capability of not congesting the network. In average, DET-Lin obtains a 11,2% higher latency than *DESTRO*.

4.3.5 *DESTRO* Memory Requirements

Finally, as we did in Section 3.4.5, we compare the amount of memory required by the *DESTRO* implementation based on FIR and by a routing algorithm based on forwarding tables.

In order to provide an easy reading, we remind that the memory required to implement an adaptive routing algorithm based on forwarding tables is $C_{FT_{adap}} = N \times (log(N) + k \times log(2 \times k))$ bits in each switch. However, as *DESTRO* is a deterministic routing algorithm, in order to perform a fair comparison, we just compare it with the cost of implementing a deterministic routing algorithm with forwarding tables, which is $C_{FT_{det}} = N \times (log(N) + log(2 \times k))$ bits per switch.

On the other hand, we have FIR that is based on IR. In Section 3.4.5, we showed that the cost per switch of implementing IR in a network composed by N nodes, built with switches with $2 \times k$ ports, is $2 \times k \times 2 \times log(N)$. Additionally, FIR uses two other registers per switch port, the Mask Register and the Routing Restrictions Register. MR size is equal to the size of LIB and UIB, that is, $log(N)$ bits which are the bits required to represent a destination identifier. Remember that MR has as many bits as the destination identifiers because it is used to mask destinations before comparing them with the routing interval. On the other hand, RRR only requires one bit per port in the switch, that is, it requires $2 \times k$ bits. Therefore, the cost per switch of implementing FIR is $C_{FIR} = (2 \times k) \times (3 \times log(N) + 2 \times k)$ bits.

Figure 4.25 shows the required total memory per switch as the number of nodes in the network is increased, for a deterministic routing scheme based on forwarding tables and one that is based on FIR. As we did on Section 3.4.5, we show the results up to 1M nodes to show how both schemes could handle the expected number of nodes in next-generation supercomputers. We again set $k = 32$ to reflex the trend of using high-radix switches. Again, the memory required to implement the forwarding tables schemes stays several orders of magnitude higher than the one required by the interval based one (FIR). Forwarding tables are only interesting on very small supercomputers with less than 100 nodes.

4.4 Conclusions

In this chapter, we have presented *DESTRO*, which is a deterministic routing algorithm for fat-trees that balances network traffic as much as possible and what is more important can hold the congestion of the network as near the

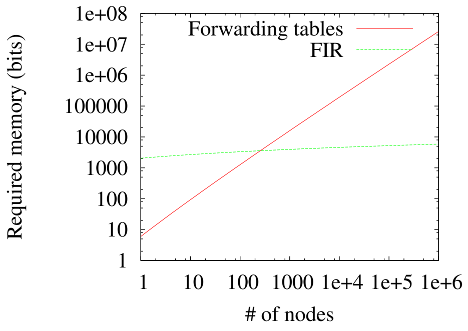


Figure 4.25: Memory requirements per switch for a deterministic routing scheme based on forwarding tables and one based on FIR in a network with $k = 32$.

origin nodes as possible. In this way, the rest of the network remains free of the congestion and it can be used by the packets that are not directed to the congestion. Indeed, *DESTRO* tends to classify the packets according to their destinations in such a way that the number of destination that share a link in the ascending routing phase is minimum, and it is reduced to only one destination per link in the downwards phase.

Thanks to this packet classification, *DESTRO* has shown to be able to obtain almost the same performance than the best adaptive routing algorithms, even outperforming them in some cases. However, when considering using disk traces, *DESTRO* has shown to be able to deal with all the traffic in the traces obtaining a markedly lower latency than the other routing algorithms.

Additionally, we have shown that the adaptive routing algorithms do not preserve the injection order of the packets, which is mandatory in several applications. They require to use additional mechanism in order to preserve the order of the packets. When considering this fact, the evaluation has shown that the adaptive routing algorithms are penalized when they must guarantee that packets are delivered in-order, specially when the simulation with traces are considered. In these simulations, the adaptive routing algorithms that guarantee the delivery order of the packets are clearly outperformed by *DESTRO*. This difference is more important that it could seem, since *DESTRO* does not require a selection function like adaptive routing algorithms, and it does not require the use of additional mechanism to preserve the ordering of packets, since deterministic routing algorithms preserve it by design. So, *DESTRO* can outperform or achieve the same performance than the best adaptive routing algorithms using less resources than them. Remember that adaptive routing requires the implementation not only of the routing function but also of a selection function, whereas deterministic routing only requires to implement the routing function. Moreover, adaptive routing requires a mechanism to guarantee in-order delivery of packets. In addition, we have shown that *DESTRO* can be easily implemented with FIR with $O(log(N))$ storage requirements.

Chapter 5

RUFT: Simplifying the Fat–tree Topology

"Do or do not. There is no try."

Yoda, The Empire Strikes Back

In current supercomputers, the hardware cost, power consumption and developing cost are becoming main concerns when developing a new system [19, 21, 29, 92, 98]. Furthermore, there are other fields where metrics such as energy consumption and development cost are as important as performance [88]. This is the case, for example, of Networks-on-Chip (NoCs), which are interconnection networks specifically developed to be fitted inside a chip. For this, they have very severe restrictions on space and power consumption.

A lot of works have been made in order to reduce the costs of the interconnection network for off-chip interconnection networks [17, 62, 68, 119] and for NoCs [22, 51, 53, 87, 107, 132]. Some of these works rely on developing mechanism that dynamically can switch on and off the links of the interconnection network [17], or adjusting the working frequency or voltage of the system [53, 119], or reducing the amount of memory at the switches [62], or designing new flow-control mechanisms [107], and many other changes on the structure of the elements of the network.

However, there are few works that are focused on exploring the benefits of simplifying the topology in terms of hardware cost, power consumption, or developing costs. In this chapter, we take on this challenge by deriving the fat-tree topology to a new topology, that can achieve almost the same level of performance than the fat-tree, but with a markedly smaller cost than the fat-tree.

5.1 Introduction

In the previous chapters of this dissertation, we have shown that interconnection networks play a key role to achieve performance and fault-tolerance on parallel computers. In particular, in Chapter 4, we showed how to maintain, even improve, the fat-tree performance while keeping the simplicity of the network, specially for systems that require in-order delivery of the packets. Concretely, we showed how to simplify the network with a new deterministic routing algorithm (*DESTRO*) for fat-trees that can outperform the more complex adaptive routing algorithms while ensuring in-order delivery of packets.

The aim of this chapter is to simplify even further the switch architecture and to decrease the required network resources, while keeping, or even improving, network performance. The basic idea of this work is to simplify the fat-tree topology by reducing the complexity of the downwards phase, taking advantage of some particular properties of the *DESTRO* routing algorithm. In this way, we can design faster, lighter and more compact switches, while providing some interesting power-saving capabilities. The benefits of this proposal seem clear and applicable to a wide field of interconnection networks, ranging from cluster-based machines to NoCs.

There are some other works that progress in the same direction. For instance, in [67,68] the authors introduce the flattened butterfly topology. In this new topology, the fat-tree is flattened into a single stage. In other words, all the switches from different stages that belong to the same row are compacted into a single high-radix switch. In order to preserve network connectivity, a high number of longer links are necessary among the flattened switches. In this way, the network is simplified, since the number of switches and wires are reduced. Nevertheless, switch complexity is considerably increased.

Another example of compacting topologies is presented in [21]. In this work, the authors present a strategy to compact the mesh topology, by attaching several processing nodes to a single switch. However, in order to preserve the bisection bandwidth of the topology, express channels [34] are added along the perimeter of the mesh. Again, the number of switches of the network is reduced at the cost of increasing switch radix and complexity.

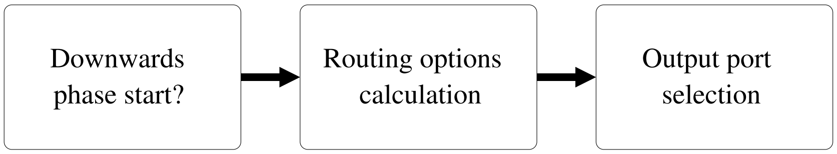
On the contrary to these two proposals, in our approach, both network and switch architectures are considerably simpler than the ones used in the fat-tree topology.

The rest of the chapter is organized as follows. First, we introduce our new topology in Section 5.2. Next, in Section 5.3, we enumerate the advantages and disadvantages of our topology. In Section 5.4, we evaluate our proposed topology according to several criteria. In particular, we perform a cost comparison between our proposal and the fat-tree topology in Section 5.4.1, a performance evaluation in Section 5.4.3, and a comparison of the cost/performance ratio of the studied topologies. Finally, some conclusions are drawn in Section 5.5.

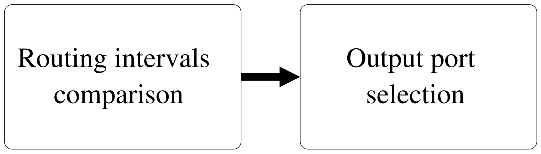
5.2 Description of the *RUFT* Topology

As commented in Chapter 4, by using a deterministic routing algorithm, switch architecture complexity is reduced, compared to an adaptive one, since neither selection function, nor additional hardware resources to ensure in-order delivery of packets are required. This can be easily seen in Figure 5.1 that shows the steps that usually are performed during the routing of a packet in a fat-tree. Figure 5.1(a) shows that a switch that uses an adaptive routing algorithm implemented by fixed logic identifies in first place if the packet has to start its downwards routing phase; then, it calculates the routing options of the packet and performs the selection function. Figure 5.1(b) depicts the routing steps for an adaptive routing algorithm implemented with IR. As it can be seen, the implementation with IR does not require to check if the packet has to start its downwards phase, since this has been pre-calculated before setting the routing intervals to the correct values to provide minimal routing.

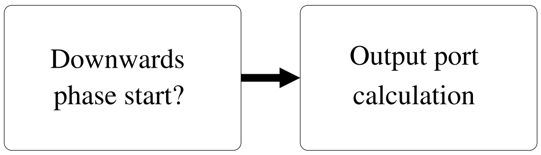
Figure 5.1(c) represents the routing steps in a deterministic routing algorithm like *DESTRO* implemented by dedicated logic. As it can be observed, first, the switch identifies if the packet has to start its downwards phase, calculating the output port after that. As it can be seen, deterministic routing



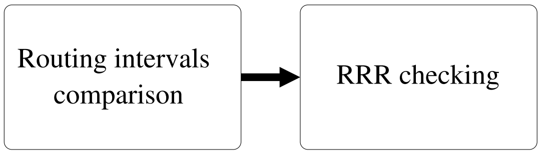
(a) Adaptive routing implemented with logic.



(b) Adaptive routing implemented with IR.



(c) *DESTRO* implemented with logic.

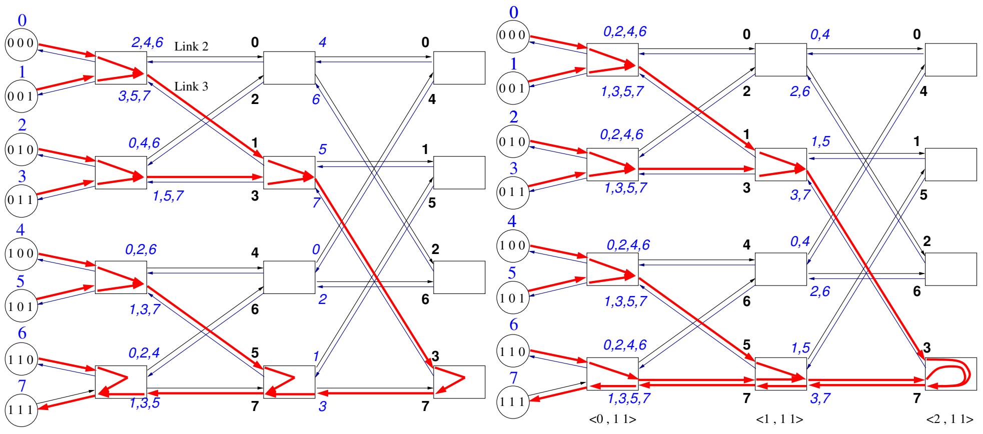


(d) *DESTRO* implemented with FIR.

Figure 5.1: Steps to route a packet in a fat-tree.

algorithms implemented in logic are simpler, take less steps, than adaptive ones also implemented in logic. In addition, in *DESTRO*, the output port for each packet can be calculated by just looking up a single component of the destination identifier, both in the upwards and downwards phases, which helps in simplifying and speeding up the routing logic. Finally, in Figure 5.1(d), we show the routing steps performed by FIR to route the packets using *DESTRO*. In this case, the routing logic calculates all the possible routing options by using the routing intervals of the output ports of the switches, and then the RRR registers are taken into account to provide only minimal routing. Despite that the adaptive routing algorithm implemented with IR has the same number of steps than the deterministic algorithms, the implementation of the selection function is much more complex than the second step of the deterministic routing algorithms. This can be easily observed by comparing Figure 4.12, which presents the steps to implement a selection function, and Figure 4.9, which presents the logic required to check the RRR of a port in a switch with FIR.

However, despite that *DESTRO* routing is very simple, there are some physical design issues concerning the routing logic at the switches that can still be addressed. In *DESTRO*, choosing the output port can be very simple, but specific logic is still required to decide when to start the downwards phase. As commented, the output port to use for a given packet is provided by the component of the destination address corresponding to the stage of the switch (i). During the downwards phase, the output port to use is always i, but during the upwards phase the output port can be either $k + i$ or i depending on the output direction of the packet. If the packet continues traveling upwards it would be forwarded through link $k + i$, whereas if packet has to start the downwards routing phase it would use the port i. In the implementation of *DESTRO* shown in Figure 5.1(c), the decision of starting the downwards phase involves comparing up all the bits of the destination address with the switch identifier. This large comparison may delay the routing decision. Moreover, in the implementation with FIR, RRR are only used to indicate which packets must start their downwards routing phase at each switch. So, to check if the downwards phase must be started does not only takes time, but also it requires an additional register per output port. That is, it also increases the cost of the implementation. Next, we pretend to remove such decision from the routing



(a) Minimal routing with *DESTRO*. (b) Non-minimal routing based on *DESTRO*.

Figure 5.2: A 2-ary 3-tree with deterministic routing. All the routes to node 7 have been highlighted.

steps and study its consequences.

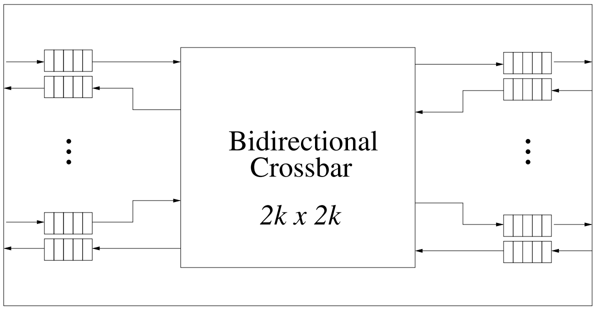
As a first attempt to remove this comparison from the routing logic, in order to perform a quicker and simpler routing decision, we propose to use non-minimal routing. To help understanding the consequences of this change, in Figure 5.2, we show all the reachable destinations at each output port for *DESTRO* with minimal routing and with non-minimal routing, highlighting in both cases the routes for all the packets whose destination is node 7. As it can be seen, with non-minimal routing, all the packets are enforced to reach the last stage of the network before starting the downwards phase.

In this way, contention in the downwards phase is fully removed, since the only point where packets can begin their downwards routing phase is at the last stage. On the contrary, in the original algorithm, packets may begin their downwards routing phase at any stage of the network, creating a potential contention point at every stage in the downwards phase (see Figure 5.2(a)). In addition, in the non-minimal scheme, the number of hops of all the paths along the network is the same, as all the packets are forced to reach the last stage of the network before starting the downwards phase. Concretely, the length of the routes for all the packets always is $n \times 2$, n being the number of network stages. Thus, non-minimal routing helps to obtain the same latency for every source–destination pair. Nevertheless, there is a very important drawback in this approach: the average packet latency is increased, as all the packets are enforced to reach the last stage, using routes that are longer than the ones that they would use in minimal routing.

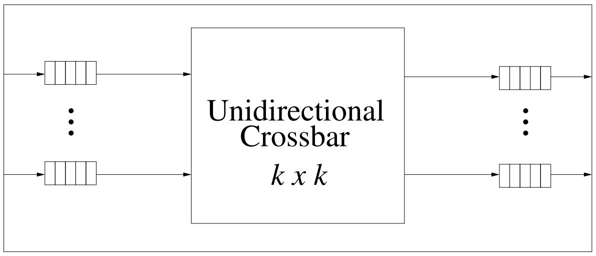
In the previous chapter, in Figure 4.6, we showed that all the descending links are only used by the packets directed to one destination. With this property in mind and using the presented naive solution to remove the decision of starting the downwards phase from the routing logic, we can observe a very interesting property in the network with non-minimal routing: all the conflicts between packets can only appear in the upwards phase. Indeed, in Figure 5.2(b), each descending output port only receives packets from the same input port. As it can be seen in Figure 5.2(b), at switch $< 1, 11 >$ packets received through link 2 are always routed to link 0, while the ones received through link 3 always go through link 1. Moreover, output port 0 can only receive packets from port 2, and output port 1 only receives packets from port 3.

In fact, all the switching activity in the downwards routing phase is completely unnecessary. So, our proposal is to remove all the hardware associated to the downwards phase from the switches, reducing the required resources, as it can be seen in Figure 5.3. A block diagram of a fat-tree switch is depicted in Figure 5.3(a). The figure depicts a simplified structural view of a switch with the crossbar in the center which is connected to the buffers of the ports of the switch. Notice that the switch is bidirectional, so most elements are replicated to provide service to ascending and descending packets at the same time. By removing the hardware associated to the downwards routing phase, the switch architecture can be considerably reduced, as shown in Figure 5.3(b). As it can be observed comparing both figures, buffer requirements and the number of ports are reduced by a half, while crossbar complexity is reduced by a factor of four, as the switch has became unidirectional.

Furthermore, as switches no longer need to check if packets have to begin their downwards routing phase during the upwards phase, routing function is simplified. So, the first step from Figure 5.1(c) or the RRR checking in Figure 5.1(d) are no longer necessary. In addition, as the number of ports have been reduced to the half, arbitration is also simplified. This two facts will likely speed up the switch. As shown in Figure 5.3(a), in conventional



(a) Fat-tree switch



(b) Our switch

Figure 5.3: Switch comparison.

switches, arbitration has to perform a $2k \times 2k$ matching, being k the switch arity; whereas in the simplified version (Figure 5.3(b)), it has to perform a $k \times k$ matching. Additionally, as links are unidirectional, we have also reduced switch port complexity, and the number of total links in the network.

In this way, since packets destined to a given node always follow the same descending path and the switching activity is not necessary in the descending phase, we propose to introduce a single long link connecting the output ports of the switches of the last stage with the input port of the corresponding destination, eliminating the downwards phase hardware from all the network. Figure 5.4 shows the resulting topology which will be referred to as *RUFT*: Reduced Unidirectional Fat-Tree. Paths going to node 7 are highlighted. This points out possible issues related to the delay of the long links, although it can be compensated by the reduced delay derived from the simplified switch architecture, as we will show on the evaluation section. The resulting topology resembles an unidirectional butterfly, with a permutation on the reachable

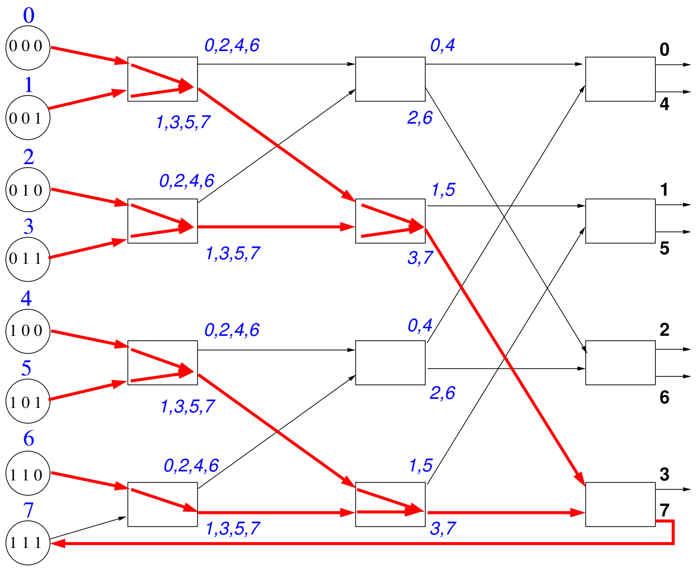


Figure 5.4: The $RUFT$ topology derived from the 2-ary 4-tree. All the routes to node 7 have been highlighted and each switch port shows its reachable destinations.

destinations from the last stage. A similar topology is presented in [133], whose topology has a different permutation in the last stage. Notice that due to this permutation, the routing algorithm of $RUFT$ differs completely from the proposed ones in the aforementioned topologies.

Finally, $RUFT$ has the same number of switches than the fat-tree, since it is designed in the same parametrical way and it has the same number of stages (n) and switch arity (k). $RUFT$ is able to connect $N = k^n$ processing nodes using nk^{n-1} unidirectional switches and $(n+1)k^n$ unidirectional links. Notice that the fat-tree topology requires to use bidirectional switches and links. The cost comparison between both topologies is performed in Section 5.4.1.

5.3 Advantages and Disadvantages

$RUFT$ has several advantages that can be exploited for a very wide range of configurations. First, the routing delay is reduced since just a destination component must be looked up. No routing decision or calculation must be made, the switch just uses the corresponding component as output port. In this way, the routing of *RUFT* obtains the benefits of both distributed and source routing. On one hand, it does not have to create a large header to store the route of the packet through the network, since the route is directly indicated by the destination identifier. So, the size of the packet header is as small as in distributed routing. On the other hand, the switch can work at the same speed as with source routing, since it does not require to take any routing decision. The switch just tries to forward the packet through the output link corresponding to the component of the stage at which the switch is located.

In systems like clusters that are typically wire-constrained, *RUFT* may present a lower wiring density depending in the final implementation. This can be observed by comparing Figure 5.2(a) and Figure 5.4. This lower wiring density involves that *RUFT* can be more easily encapsulated in a cabinet, in off-chip systems; whereas in on-chip networks, this involves that *RUFT* would likely have less crosspoint in its layout, reducing in this way the number of metal layers used by the network which reduces the cost of the chip [88, 99]. Additionally, the proposed topology helps in addressing some emerging constraints in system design due to the increasing density of nodes, such as power consumption, heat dissipation, packaging weight and size limitations. *RUFT* can reduce or lessen all these constraints due to the fact that it completely removes the downwards phase, reducing the resources of the network to almost the half, as it can be seen in Section 5.4.1.

On the other hand, since NoCs are mainly constrained by power and area requirements (mostly driven by buffers and crossbar [22, 131]), the reduction in buffer requirements and crossbar complexity directly impacts on these issues, as *RUFT* relies in unidirectional switches instead of using bidirectional switches. Additionally, the reduction in switch complexity provides smaller routing and arbitration times, thus allowing to decrease switch input-to-output cross-time.

However, *RUFT* has its own weak points. For example, spatial locality can no longer be exploited. Packets have to cross the whole network no matter their origin and destination. This could be considered a very important weak point since there are several applications that are optimized to take advantage of spatial locality in parallel computers. Anyway, the lack of locality has some advantages, for example, techniques to efficiently map tasks into the system are no longer needed, since all nodes are equally distant. Moreover, as all the packets must perform the same number of hops, latency is more uniform and less sensitive to changes, reducing the *jitter* experienced by the applications. Finally, notice that the maximum packet latency is reduced since the downwards phase is less time-consuming as no hops are necessary, only a link must be traversed. In the worst case, in a fat-tree, a packet would have to cross all the stages in its upwards phase and again in its downwards phase. In *RUFT*, all the packets have to cross all the stages only once no matter their origin or destination. As it can be seen, the highest number of hops that a packet can make in a fat-tree doubles the maximum number of hops for a packet in *RUFT*.

Finally, the reduction of the switch hardware opens the doors to explore several ways of reusing the removed hardware from the downwards phase. For instance, larger buffers can be introduced, doubling switch storage capability, thus keeping the same buffer resources of the fat-tree. Another possibilities are to replicate the network to deal with higher traffic loads like in [21], or to provide fault-tolerance like in [116].

5.4 Evaluation

In this section, we perform the evaluation of the topology taking into account both: cost and performance. In the fist part of this section, we compare the cost of implementing *RUFT* against the cost of a fat-tree. Next, we study the performance of the new topology against the fat-tree using adaptive routing with the two most significative selection functions shown in the previous chapter of this dissertation (FF and SADP), *DESTRO*, and the non-minimal version *DESTRO* presented in Figure 5.2(b). Also, we analyze the convenience of using the same buffer space in *RUFT* as in a fat-tree. Finally, we use the results obtained in the two previous sections to compare the topologies considering the cost/performance ratio.

5.4.1 Cost Comparison

Despite that we can not offer a real economical cost analysis, we can provide the relative cost of *RUFT* over the fat-tree topology using both, adaptive and deterministic routing. For this purpose, we assume that network cost mainly depends on two factors: switch and link cost.

Switch cost (c_S) can be approximated as the number of ports (n_p) of the switch multiplied by the cost of a port (c_p) plus the cost of the switch control logic hardware (c_{cl}), which is also indirectly influenced by the number of ports. That is $c_S = n_p \times c_p + c_{cl}$. However, these components of the switch cost can be broken down to have a more detailed look of the switch cost. On one hand, c_p can be broken down into the cost of the buffers located at the switch port (c_b), the contribution of each port to the cost of the crossbar (c_{cr}), and other additional costs related to the switch ports (c_{po}), like connector interfaces. On the other hand, c_{cl} includes the routing logic cost (c_r), selection function cost (c_{sf}), and other costs related to control logic (c_{clo}). Substituting in the previous expression, we obtain:

$$c_S = n_p \times (c_b + c_{cr} + c_{po}) + c_r + c_{sf} + c_{clo} \tag{5.1}$$

Total link cost (c_L) can be more easily approximated. c_L can be established as the total number of links in the network (n_l) multiplied by the cost of the connectors (c_{cn}); plus the total network cable length (l_l) multiplied by the cost of each meter of link (c_{ml}). That is:

$$c_L = n_l \times c_{cn} + l_l \times c_{ml} \tag{5.2}$$

For this cost comparison, we are considering for the sake of simplicity that the fat-tree topologies are composed by unidirectional links like *RUFT*. In this way, the links from both topologies are composed by the same number of wires, and the logic at the switches per port for each topology is also the same. Notice that this approach is completely equivalent to consider that fat-trees are composed by bidirectional links, and *RUFT* by unidirectional links that have half the number of wires than the ones used in the fat-tree.

Table 5.1 shows the values of the different parameters for the approximated network cost of several topologies normalized to the adaptive fat-tree.

Table 5.1: Parameters of the relative network cost for the analyzed topologies normalized to the adaptive fat-tree.

	Adaptive Fat–tree	Deterministic Fat–tree	*RUFT*
Switches			
n_p	1	1	0.5
c_b	1	1	1
c_{cr}	1	1	1
c_{po}	1	1	1
c_r	1	1	0.6
c_{sf}	1	0	0
c_{clo}	1	1	1
Links			
n_l	1	1	0.5+d
c_{cn}	1	1	1
l_l	1	1	[0.5 – 1.25]
c_{ml}	1	1	1

Restricting the scope to the fat-trees, the only significant difference between both adaptive and deterministic-routed networks is that the deterministic fat-tree does not require using a selection function, so c_{sf} is 0 for this network. All the other parameters are equal since both of them use the same links and the same switches, and we have separated the costs from the selection function and the routing function.

From the point of view of the switch cost, our proposal halves the total number of ports (n_p), as we assume that links are unidirectional in all the topologies. In this way, as it can be deduced from Equation 5.1, *RUFT* reduces by half the total cost per switch related to the buffers, crossbar[1], and other additional costs related to the switch ports, despite that individual cost per port is the same ($c_b + c_{cr} + c_{po}$) than in the fat-tree[2]. Additionally, routing

[1]Indeed, crossbar size and cost increase quadratically with the number of ports, so the reduction in the crossbar cost would be higher than in other switch components.

[2]Notice that this approach is equivalent to consider that the fat-tree and *RUFT* have the same number of ports per switch, but the cost per port in *RUFT* would the half, since

control logic cost (c_r) is also decreased, due to the reduced crossbar size[3]. On the other hand, $RUFT$, as the fat-tree, that uses deterministic routing does not require a selection function, so its cost is non-existent (c_{sf}).

By applying the normalized costs from Table 5.1 to Equation 5.1, we obtain that the cost for the adaptive switch is $3n_p + 3$; for the deterministic switch is $3n_p + 2$; and for the reduced one is $1.5n_p + 1.6$; being n_p equal to $2k$. Hence, $RUFT$ reduces the cost of a switch almost by a half. As can be seen in Section 2.1.3 and Section 5.2, all three topologies have the same number of switches, so $RUFT$ halves the total cost of the switches in the network since it halves the cost of each switch.

From the point of view of the links, $RUFT$ uses a single unidirectional link per physical link whereas the fat-trees use two unidirectional links per physical link. Therefore, $RUFT$ almost halves the total number of links of the fat-tree. The number of links in $RUFT$ is not exactly half the number of links in a fat-tree because we have to consider the links that connect the switches from the last stage to the nodes. In $RUFT$, each destination only receives an input link. So, the total number of links in $RUFT$ is equal to half the number of links of the fat-tree plus one additional link for each destination. This over-cost is represented in Table 5.1 by the variable d. The value of d can be obtained from the total number of links of both topologies presented in Sections 2.1.3 and 5.2. In particular, d is approximately equal to $\frac{1}{2\times n}$. That is, it is inversely proportional to the number of stages in the network. Therefore, d is always smaller than 1 for all the possible network sizes. In the worst case, for a network with just one stage[4], d would be equal to 0.5, and $RUFT$ would have the same number of links than the fat-tree topology.

In the fat-tree, total link length is composed by the accumulated length of the ascending links and the descending links. Since a fat-tree has the same number of ascending and descending links, and they are of the same length, we can state that half of the total link cost of the fat-tree is derived from its ascending links, and the other half from its descending links.

For the links of the upwards phase in $RUFT$, we consider the same length

it is using half the number of wires per link than the fat-tree.

[3]We have assumed a 40% reduction in the routing control logic cost.

[4]Notice that this network would be composed by a single switch, and it is very unlikely that a supercomputer would be built up with a single switch.

than in the fat-tree topology, since the length of the links in the upwards phase of *RUFT* and the fat-tree is exactly the same independently of the packaging constraints. Nevertheless, the descending phase in *RUFT* is very different from the one in a fat-tree. The descending phase in *RUFT* is only composed by the links of the last stage that connect them to the destination nodes. Link length in a cluster is heavily dependant on the final packaging of the topology. In order to provide a wide comparison between the fat-tree topology and *RUFT*, we have considered both extremes: the ideal and the worst case. In the ideal case, we assume that switches belonging to the last stage are placed very near to the destinations, so the length of the links that connect the switches of the last stage to the destinations is negligible. On the worst considered case, we assume that those switches are placed far away of the destinations and the wiring of the long links is quite complex, so total link length of downwards phase is increased by 50%. So, the total link length of *RUFT* is 0.5 in the ideal case, and 1.25 in the worst case.

By applying the normalized costs from Table 5.1 to Equation 5.2, we obtain that the link cost for the adaptive and deterministic network is 2; for *RUFT* in the ideal case is $1+d$; in the worst considered case, *RUFT* link cost is $1.75+d$.

To summarize, *RUFT* almost halves switch cost, and link cost in the ideal case, in relation with classical fat-trees. At worst, link cost is a bit lower than in a fat-tree.

5.4.2 Simulation Environment

To evaluate *RUFT*, we have modified the event-driven simulator described in Section 4.3.3. The simulator can model a k-ary n-tree with *DESTRO* and adaptive routing, or a *RUFT*, always using virtual cut-through switching. As commented, each router has a full crossbar with queues both at the input and output ports. We assumed that it takes 20 clock cycles to apply the routing algorithm in the fat-tree and 12 clock cycles in *RUFT*[5]; switch and link bandwidth have been assumed to be one flit per clock cycle; and fly time has been assumed to be 8 clock cycles. Long link delay is also parameterized

[5]As aforementioned, we have assumed a 40% improvement in the arbiter due to the simplified switch architecture. Taking into account that this time depends on the number of switch ports, it is a consecutive approach.

(see below).

In order to analyze the relative performance of our proposal, we have defined six different network configurations: adaptive fat-tree with FF and SADP, the fat-tree using *DESTRO*, the fat-tree using non-minimal *DESTRO* that forces packets to reach the last stage of the network before starting the downwards phase, *RUFT* and the aforementioned double-buffered *RUFT*. Each port link has a two-packet buffer, but for the double-buffered configuration, whose ports have a four-packet buffer. For both *RUFT*s, length of long links varies from ideal length (same delay as regular links) up to the addition of the delays of the equivalent links crossed in the downwards phase in the fat-tree topology.

When adaptive routing is used, we use the most significative selection functions (both the worst and the best) from the previous chapter: First Free (FF) and Stage And Destination Priority (SADP). For more details about these selection functions see Section 4.3.1.

As in the previous chapter, we have evaluated several synthetic traffic patterns: uniform, hot-spot, bit-reversal and complement. In uniform traffic, packet destinations are randomly chosen with the same probability for all the nodes. Hot-spot concentrates a fixed percentage of the traffic in a particular destination, and the rest of the traffic behaves like in uniform traffic. In complement traffic pattern, each processor sends all its packets to its opposite node. Finally, bit-reversal forces that each processor send all its packets to the node resulting from inverting the origin label (i.e. node $< 100 >$ only sends packets to $< 001 >$). For more details, see Section 4.3.2. Also, we have performed the simulations using the I/O traces from the disk interface of the *cello* system described in Section 4.3.2.

As we have done in the previous chapter, in order to perform a fair comparison, we have implemented a basic approach to ensure in-order packet delivery with the adaptive routing algorithm. As commented in Section 4.3.1, this mechanism is based on a reorder-buffer and our implementation of the mechanism is very unrealistic, so the differences in performance that can be observed are not induced by the in-order delivery mechanism, but they are introduced by the packets that arrive out of their order.

5.4.3 Performance Results

In this section, we evaluate the performance of *RUFT* and compare it with the one obtained with the fat-tree using adaptive routing and *DESTRO*. In first place, we evaluate the influence of the delay of the link that connects the switches of the last stage to the destinations nodes on the performance of *RUFT*. Next, we compare *RUFT* against the fat-tree using the same simulation methodology used in the previous chapter.

Impact of Long Links

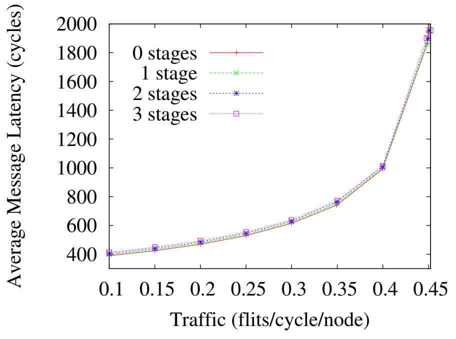
As we have already pointed out, the influence on the performance of the delay of the links that connect the switches of the last stage to the destination nodes must be studied, since we do not know whether the delay of these links may be a bottleneck for *RUFT* or not. In this section, we refer to the links that connect the switches of the last stage to the destination nodes as long links, since they could be markedly larger than the other links of the network.

Figure 5.5 presents the performance results for several 4-ary 4-trees with different long link delays for uniform and complement traffic. In both figures, those delays are represented as the required clock cycles to cross zero or more stages of the network[6]. The delay of these long links varies from the ideal latency (0 stages) to the worst considered case (3 stages). As it can be seen, network throughput is unaffected by the delays of the long links, since all the configurations for both traffic patterns achieve exactly the same throughput. On the other hand, concerning average packet latency, we can observe that this delay is directly added up to the average packet latency. This can be more clearly observed in Figure 5.5(b).

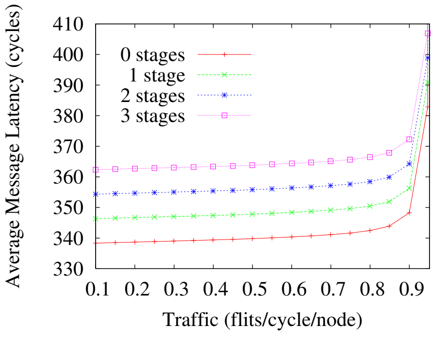
This behavior has been checked with several traffic patterns and networks configurations, going from 2-ary 3-tree to 2-ary 8-tree, from 4-ary 2-tree to 4-ary 6-tree, and from 8-ary 3-tree to 8-ary 4-tree and the results are qualitatively the same. For this reason, in the following sections of this chapter, we only consider the worst long link delay for the sake of clarity.

Now, that we clearly know how the delay of the long links affects to the

[6]This is approximated as the addition of the fly time through a link, the required time to perform the routing, and to cross the switch.



(a) Uniform traffic.



(b) Complement traffic.

Figure 5.5: 4-ary 4-*RUFT*s varying the delay of the links that connect the switches of the last stage to the destination nodes.

performance of the network, we pretend to analyze if this delay and the lost of the minimality of paths is compensated by the reduction in latency expected by removing all the descending phase in *RUFT*.

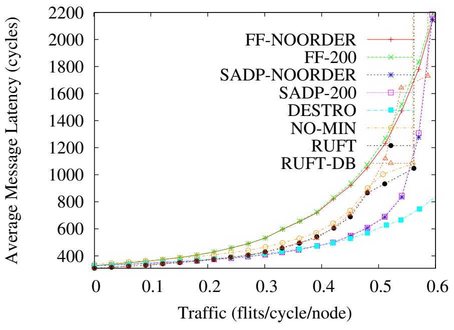
Results for Synthetic Traffic Patterns

As commented above, in this section, we compare the two representative selection functions for adaptive routing (FF and SADP), minimal and non-minimal *DESTRO*, *RUFT* and a version of *RUFT* that doubles the size of the buffers of the switches. In the previous chapter, we showed that enforcing in-order delivery of packets impacts on the performance of adaptive routing. For this reason, we present the figures including the curves for adaptive routing algorithms that preserve the order of the packets by using a reorder-buffer of 200 packets per node in each destination NIC in addition to the ones which do not preserve the packet order. Results for lower reorder-buffers are not shown for the shake of clarity. We include results for both FF and SADP to show the performance range in which we expect that all the adaptive routing algorithms are included. We also compare *RUFT* with *DESTRO*, since we showed in the last chapter that *DESTRO* was able to outperform the adaptive routing algorithms in several scenarios. Finally, we include the comparison against non-minimal *DESTRO* since it was the starting point for *RUFT*, and the only difference between them is that all the hardware associated to the downwards phase has been completely substituted by some links from the switches of the last stage to the destination nodes in *RUFT*. Thus, we expect that *RUFT* obtains a similar performance to this version of *DESTRO*, but with a lower latency, and eventually a lower throughput.

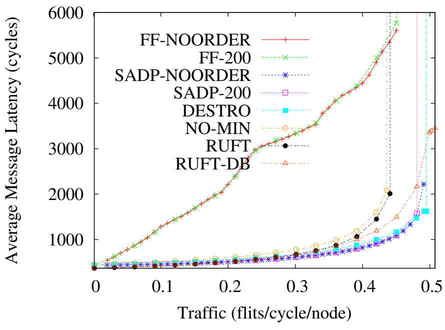
As in the previous chapter, we start by comparing the different networks with uniform traffic. In Section 4.3.4, we showed that FF was clearly the worst selection function because it obtained a lower throughput and a markedly higher latency than the other selection functions. On the contrary, SADP was the best selection function because it was the one that obtained the highest throughput and the lowest average latency. On the other hand, we showed that *DESTRO* obtained a higher throughput than all the adaptive routing algorithms in the networks with few stages, and this difference was reduced as the number of stages was increased.

Figure 5.6 shows the results for uniform traffic for different k-ary n-trees. As it can be seen, in all the cases, *RUFT* is not able to achieve the same throughput than *DESTRO*. Indeed, it obtains 8.3% lower throughput in the 4-ary 2-tree case and 15% lower throughput in the 4-ary 4-tree. Concerning adaptive routing, *RUFT* clearly obtains a lower latency than FF in all the cases and a slightly higher latency than SADP. In terms of throughput similar conclusions can be drawn. *RUFT* can not outperform SADP, but it outperforms FF. These results where expected, since in uniform traffic any node can send packets to any node of the network with the same probability for each destination. Therefore, in this traffic pattern, the minimality of the paths is exploited by the network configurations that provide minimal paths. Indeed, as it can be seen, the non-minimal version of *DESTRO*, referred to as *NO-MIN*, obtains similar results as *RUFT*, but with a higher average latency of packets.

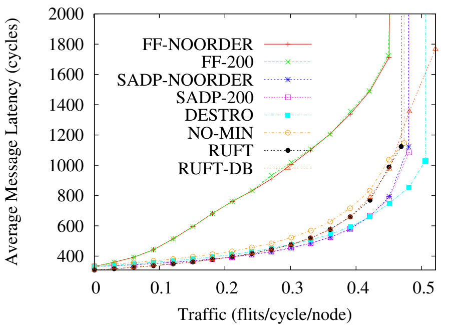
Nevertheless, the average packet latency of *RUFT* for low injection rates is lower than the one of *DESTRO* and SADP. Indeed, it obtains a latency at zero load 8.7% and 5.7% lower than *DESTRO* and SADP, respectively. This is due to the fact that packets traverse a lower number of stages. However, as the traffic in the network increases, the minimal algorithms could deliver packets to near nodes in less hops, avoiding that these packets are highly affected by the congestion, whereas in *RUFT* packets have to cross the whole network.



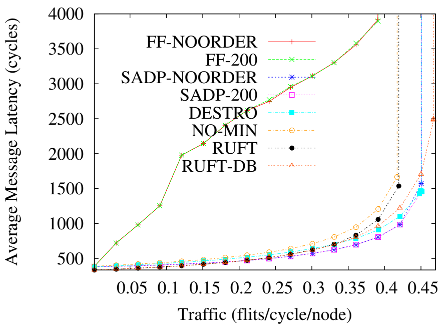
(a) 4-ary 2-tree.



(b) 4-ary 4-tree.



(c) 8-ary 2-tree.



(d) 8-ary 3-tree.

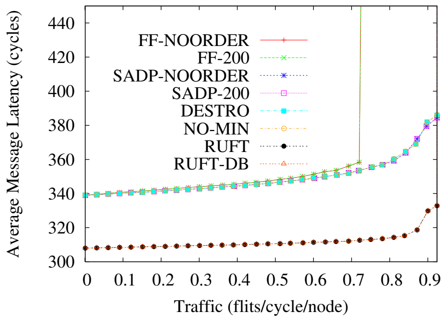
Figure 5.6: Average packet latency versus accepted traffic for uniform traffic guaranteeing in-order delivery of packets.

Therefore, for high injection rates, *RUFT* can not cope with the same amount of packets than networks that use minimal routing algorithms.

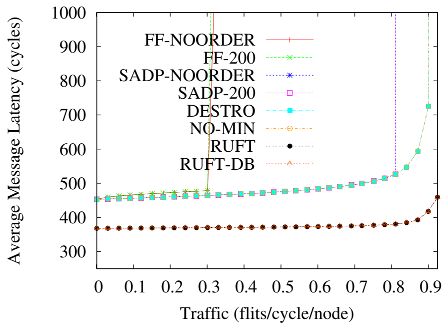
On the other hand, the double-buffered version of *RUFT*, which is indicated in the figure as *RUFT-DB*, behaves similarly to *RUFT* for low and medium injection rates, but for high injection rates, it outperforms the single-buffered version of *RUFT*, obtaining even a higher throughput than SADP for all the cases though the difference is very small. Moreover, *RUFT-DB* reaches a higher throughput than *DESTRO* for all the networks but the smallest one, being the difference around 1-2%. Despite that the difference in throughput is almost negligible, we have to remark that *RUFT-DB* has almost half the resources in the network than the fat-tree topologies.

Next, we show in Figure 5.7 results for complementary traffic. In the previous chapter we have presented this traffic pattern, in which all the packets are always sent from a node to its complementary node in the tree, forcing that all the packets reach the last stage of the network. This provoked that the adaptive algorithms were outperformed by *DESTRO* since it can contain the packets for each destination into a single path, while the adaptive algorithms tend to spread them all over the network, increasing the HOL blocking effect and as consequence increasing the packet latency.

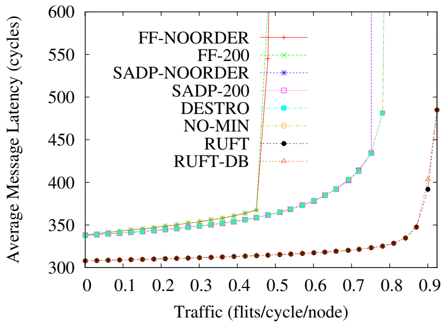
As expected, with this traffic pattern, non-minimal *DESTRO* obtains exactly the same results than *DESTRO*. Since all the packets are forced to reach the last stage of the network by the traffic pattern, the minimal version of the routing algorithm provides the same paths to the packets than the non-minimal one. However, the most important point of the figure is that *RUFT* obtains a higher performance than the other networks. As commented above, in a network without collisions, *RUFT* obtains a lower latency than the other networks. Moreover, with this traffic pattern the average latency of *RUFT* is always lower than the one from adaptive routing algorithms and *DESTRO*. With complement traffic, *RUFT* provides the same routes than *DESTRO* in the upwards phase to all the packets. However, in the downwards phase, despite that *DESTRO* does not suffer any delay related to HOL blocking, it requires to route the packets in each stage increasing the average packet latency. On the other hand, in *RUFT*, packets only need to traverse a link to reach their destination once the ascending phase has finished. For this reason,



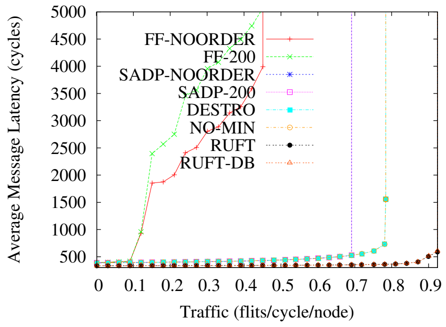
(a) 4-ary 2-tree.



(b) 4-ary 4-tree.

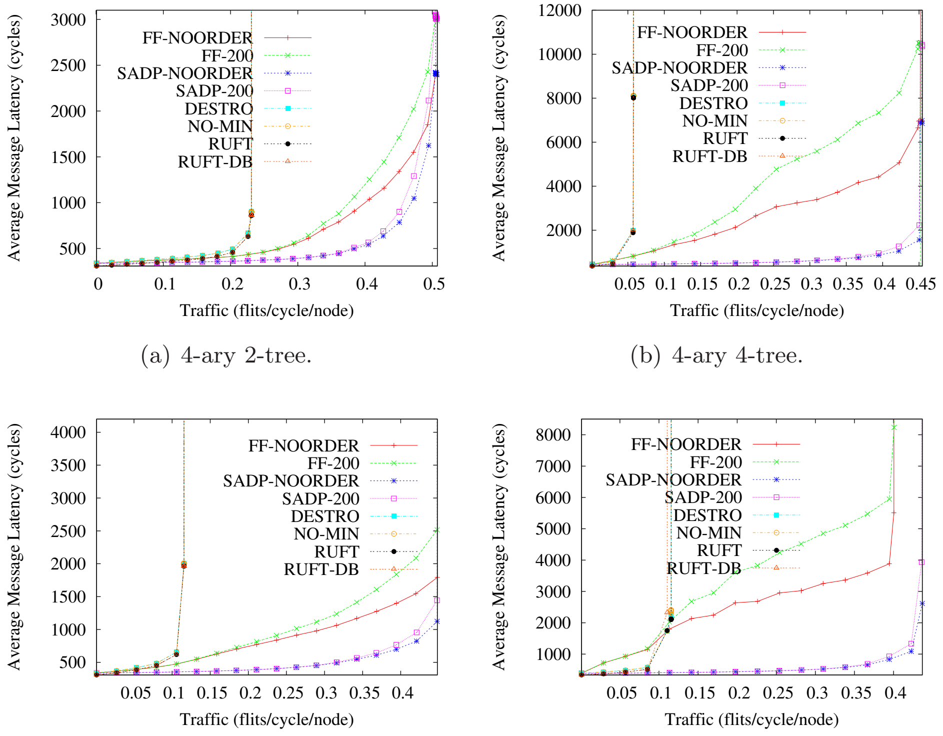


(c) 8-ary 2-tree.



(d) 8-ary 3-tree.

Figure 5.7: Average packet latency versus accepted traffic for complement traffic guaranteeing in-order delivery of packets.



(a) 4-ary 2-tree.

(b) 4-ary 4-tree.

(c) 8-ary 2-tree.

(d) 8-ary 3-tree.

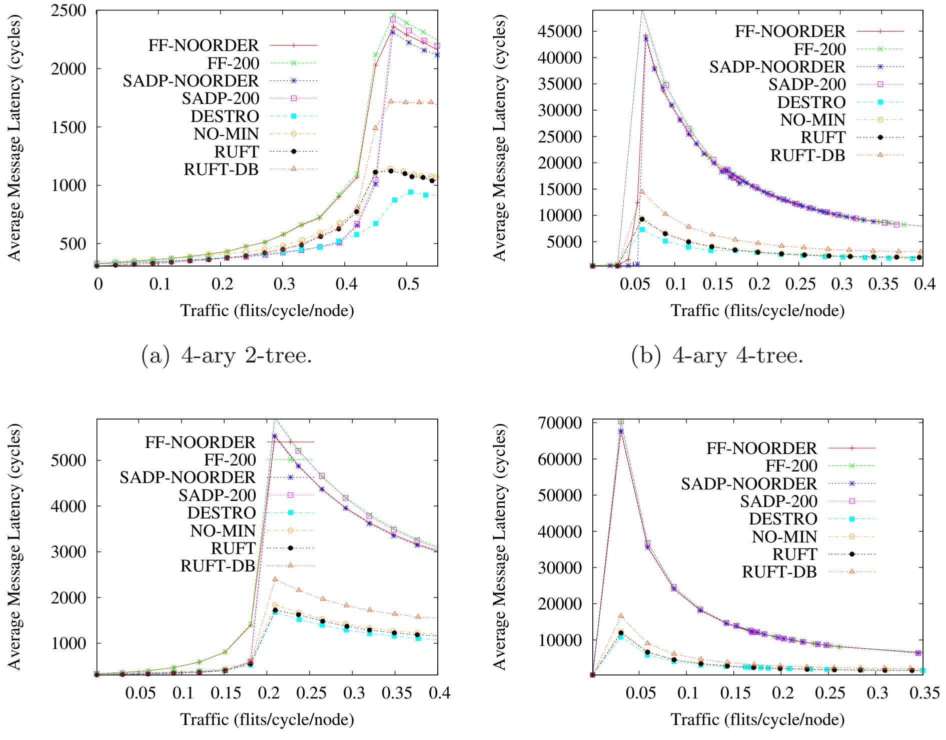
Figure 5.8: Average packet latency versus accepted traffic for bit reversal traffic guaranteeing in-order delivery of packets.

the difference in latency increases with the number of stages.

Concerning throughput, we can see that *RUFT* obtains a higher throughput than *DESTRO* as the size of the network is increased. It obtains the same throughput than *DESTRO* in the 4-ary 2-tree, and it obtains 2,7%, 16,2%, and 17,9% higher performance than *DESTRO* in the 4-ary 4-tree, 8-ary 2-tree, and 8-ary 3-tree, respectively. Furthermore, *RUFT* reaches exactly the same throughput for all the network sizes with this traffic pattern, indicating that the network is not a bottleneck in RUFT when this traffic pattern is used.

Finally, *RUFT-DB* obtains the same results than *RUFT* since in this traffic pattern *RUFT* can deal with all the traffic. So, adding new buffers does not provide any additional advantages.

Following, Figure 5.8 presents the results for bit reversal traffic pattern.



(a) 4-ary 2-tree. (b) 4-ary 4-tree.

(c) 8-ary 2-tree. (d) 8-ary 3-tree.

Figure 5.9: Average packet latency versus accepted traffic with a 5% hot-spot guaranteeing in-order delivery of packets.

As commented in the previous chapter, this traffic pattern is the worst case for *DESTRO* because it reduces the effective bandwidth of the network by a factor of k, since switches only use a single up link in the ascending phase for all the packets. Since the reduction of the bandwidth appears in the upwards phase, it is expected that bit reversal traffic pattern represents also the worst corner case for *RUFT*. As it can be seen in the figure *RUFT*, *RUFT-DB*, and non-minimal *DESTRO* obtain almost the same performance as *DESTRO*. Therefore, for this traffic pattern *RUFT*, at best, obtains 53% lower throughput than SADP, and 87,7% lower throughput, at worst.

Finally, Figure 5.9 shows the performance results for a traffic pattern with a hot-spot node that receives the 5% of the total traffic of the network. As we commented in the previous chapter of this dissertation, hot-spot traffic pattern negatively affect all the routing algorithms. But, *DESTRO* and *RUFT* obtain a lower latency than the adaptive routing algorithms. However, *RUFT* always has a bit higher latency than *DESTRO*, and a slightly lower latency than non-minimal *DESTRO*. For this traffic pattern, we can see that *RUFT-DB* always obtains a higher latency than the single-buffered one. This effect is fully explained in the following section.

To sum up, *RUFT* is able to obtain similar performance than *DESTRO* for the worst traffic patterns, that is, bit reversal and hot-spot traffic patterns. It also is able to outperform it and the adaptive routing algorithms for complementary traffic, and obtains lower performance for uniform traffic. Nevertheless, we want to remark that despite having lower performance for that traffic pattern, it obtains such results with half the resources than the other networks.

Results for I/O Traces

In order to thoroughly evaluate *RUFT*, we extend the evaluation to the I/O traces obtained from *HP* (see Section 4.3.2). However, in order to compact the presentation of these results, we summarize all of them in Table 5.2, which presents for each trace the time required to fully process it, the average network latency, and the average latency from generation. All of them are normalized to the ones obtained by *DESTRO*. Also, we do not present results for FF because the behavior of this selection function is clearly the worst by a large difference with SADP, and we want to focus on the networks and routing algorithms that obtain similar results. As in the previous chapter, the *cello* system has been mapped to a 2-ary 7-tree.

As expected, differences on the time required to deliver all the messages of each trace are not very representative, since we also observed in the previous chapter that all the routing algorithms required almost the same time to fully process the trace. As we have already explained, this is due to the fact that the traces do not include any information about which packet is a response to which other packet, and inter-burst idle periods hide the effectiveness of the network. Only in Trace #5 and Trace #6 (the most demanding ones in terms of bandwidth) it is possible to observe a difference of 8.6% and 10.7% between SADP and Deterministic, respectively. In these two traces, inter-burst idle periods are so short that bursts of traffic are too close to each other, and the most inefficient configuration (SADP) is not able to cope with all the traffic from the bursts on time. As we commented in the previous chapter, total required time on its own is not a good metric to compare the different configurations, as it is heavily influenced by the idle periods included in the traces. For this, we also consider average network latency and average latency from packet generation. Network latency is the time from packet injection into the network until its reception at the destination. On the other hand, latency from generation is the time from packet generation until its reception at the destination. Notice that this includes the waiting time at the source NIC and the network latency. Commonly, a high network latency implies that the network is heavily congested. In addition, a high difference between both latencies indicates a high level of packet contention at the sources.

In Table 5.2, SADP obtains a markedly higher network latency than *DESTRO*, ranging from 2.16 to even 44.98 times the network latency of *DESTRO*. On the other hand, latency from generation goes from 1.04 to 5.86 times the one of *DESTRO*. Even when SADP achieves a much higher latency from generation than *DESTRO*, this difference is noticeably smaller than the difference in network latency. These results confirm the trend shown in the previous chapter. With adaptive routing, congestion is spread along the network, increasing network latency. On the contrary, with *DESTRO*, network congestion is contented near the sources, providing a lower network latency, but a larger waiting time prior to injection into the network. Anyway, in *DESTRO* the reduction of the congestion alleviates this effect and even latency from generation is lower than in SADP. Notice that the lowest differences in both latencies are obtained in Trace #4, which indicates that trace is not very bandwidth demanding.

Table 5.2 also shows the results for the Non-minimal *DESTRO*. As it can be seen, this routing algorithm obtains almost the same latency than *DESTRO* in most of the traces. These results were very unexpected, since using non-minimal routing should involve increasing packet latency, as packet route length is highly increased. For this, we analyzed the I/O traces, and we observed that the average packet route length is 12 hops in the case of minimal routing, and it is only increased to 14 hops for non-minimal routing. Such a

Table 5.2: Results for different traces normalized to *DESTRO*.

	Trace #1			**Trace #2**			**Trace #3**		
	T. Time	Net. Lat.	Gen. Lat.	T. Time	Net. Lat.	Gen. Lat.	T. Time	Net. Lat.	Gen. Lat.
Non-minimal	1	1.029	1.013	1	1	1.004	1	0.969	0.999
RUFT	1	0.946	1.008	1	0.939	1.003	1	0.939	0.996
Double Buff.	1	1.938	1.003	1	1.85	0.999	1	1.871	0.988
SADP	1	44.98	2.229	1	20.83	1.035	1	35.577	1.239
	Trace #4			**Trace #5**			**Trace #6**		
	T. Time	Net. Lat.	Gen. Lat.	T. Time	Net. Lat.	Gen. Lat.	T. Time	Net. Lat.	Gen. Lat.
Non-minimal	1	1.068	1.004	1	1.074	0.906	0.998	1.025	0.995
RUFT	1	0.876	0.997	1	1.02	0.805	0.998	0.997	0.959
Double Buff.	1	1.068	0.997	1	1.986	0.803	0.998	1.954	0.95
SADP	1	2.164	1.037	1.086	15.711	2.651	1.107	19.224	5.864

small difference in route length enforces minimal *DESTRO* to perform very similar to non-minimal one. The most confusing results can be observed for Trace #5. In this trace, non-minimal *DESTRO* increases the network latency by 7.4%, whereas it decreases latency from generation by a 9.4%. As commented, Trace #5 is one of the most congested traces, it is very bandwidth demanding. As commented, *DESTRO* deals with congestion by contenting it on the source nodes, increasing the waiting of the packets before injecting them in the network (that is, latency from generation) while decreasing network congestion and therefore decreasing network latency. In non-minimal *DESTRO*, as packet routes have been lengthened, network latency is increased. Nevertheless, longer routes can use a higher number of buffers, therefore source nodes are able to inject more packets before stopping injection due to congestion. This allows packets that do not follow congested routes to have a reduced waiting time at the source NICs, so latency from generation is decreased. These results can be also observed in the other highly congested trace (Trace #6), but at a lower extent.

The most interesting issue of Table 5.2 is *RUFT* behavior. As expected, *RUFT* follows the same trend than non-minimal *DESTRO*, but with lower network and generation latencies. *RUFT* obtains similar results to non-minimal *DESTRO*, because both follow identical routes in the upwards phase. However, it reduces the packet latency because it totally removes the downwards routing phase. Despite the length of the long links, packets do not suffer any delay related to switches and congestion in the downwards phase of *RUFT*.

In general, *RUFT* obtains slightly lower latencies than *DESTRO*. Particular differences between *DESTRO* and *RUFT* depend on the specific communication pattern of each trace. For instance, *RUFT* decreases network latency over *DESTRO* by 12.4% in Trace #4, while in Trace #5, it increases network latency by 2%. Regarding latency from generation, *RUFT* improves the one from *DESTRO* by 3% and 19.5% in Trace #4 and in Trace #5, respectively. With these results, we can state that the elimination of the downwards routing phase in *RUFT* highly compensates the lost of minimal paths and the additional delay introduced by the links that connect the switches from the last stage of the network to the destination nodes. Furthermore, notice that *RUFT* halves the resources of the interconnection network while obtaining the same performance than *DESTRO* and SADP, even outperforming them in latency in some cases.

Finally, we have also analyzed double-buffered *RUFT*. As commented, in this version of *RUFT*, we have reused the buffers that have been taken from the downwards phase to double the buffers in the input and output ports of the upwards phase. This version of *RUFT* always obtains a higher network latency than single-buffered *RUFT* (i. e. network latency is increased by a factor between 85% and 98% for all the traces, but the most relaxed one). On the other hand, it slightly reduces latency from generation. Latency from generation is a 0.9% lower in the double–buffered *RUFT* than in regular *RUFT*. This behavior is due to the higher number of buffers available in double-buffered *RUFT*. As switches can store more packets, sources can inject them quicker, therefore packets spend less time waiting at the source nodes, but more time in the network since the queues of the buffers of the switches are longer, finally having a similar latency from generation.

5.4.4 Cost and Performance Comparison

In order to provide a cost/performance comparison of the different networks analyzed in this chapter, we present in Table 5.3 the cost of each studied network normalized to the fat-tree with deterministic routing. The values in the table have been obtained by applying the equations and parameters presented in Section 5.4.1. From the table, we can observe that the fat-trees

Table 5.3: Cost of each network for the analyzed network sizes without in-order delivery mechanism normalized to *DESTRO*.

k	n	Adaptive	*DESTRO*	*RUFT* Ideal	*RUFT* Worst
4	2	1.04	1	0.53	0.56
4	4	1.04	1	0.53	0.55
8	2	1.02	1	0.52	0.53
8	3	1.02	1	0.51	0.53
2	7	1.06	1	0.54	0.59

with adaptive routing has a slightly higher cost than *DESTRO*, since they use the same topology and almost the same switches, the only difference is that switches for deterministic routing do not require the use of a selection function. Concerning *RUFT*, we can observe that *RUFT* halves the cost of the network in the ideal case for all the switch and network sizes that have been analyzed. Furthermore, we can observe that the difference between the best and the worst case for *RUFT*, in which the cables from last stage to destination nodes are extremely long, is not as big as could be expected. At worst, *RUFT* cost is 59% the cost of the deterministic fat-tree. This reduction is produced for the 2-ary 7-tree, which is the network with the highest number of stages, since in this network the number of links that connect the last stage to the destination nodes is also the highest one from all the studied networks, and therefore the penalization for having long links is higher. Nevertheless, notice that even in this case, *RUFT* requires to use 41% less resources than a fat-tree with deterministic routing.

Once the relative cost of each topology is clear, we want to analyze the cost of each topology in conjunction with its performance. As we have commented several times along this dissertation, performance has been the main and unique metric in interconnection networks till few years ago. Nowadays, cost is a metric of growing importance in interconnection networks. Even, there are fields of the interconnection networks where cost is considered at the same level of importance than performance. For this, in order to evaluate a new topology, we need to define a new metric that combines both. We will use the cost of delivering the maximum possible number of flits per node and cycle in each topology as this new metric. This metric is calculated as the resources that each topology requires divided by its throughput. In other words, this metric indicates the cost to deliver a flit per node per cycle in the network. So the lower it is, the better. In this way, if a topology is very expensive and has the same throughput than another more cheaper one, this metric would be lower in the second one, indicating that the second topology is more efficient. On the other hand, if two topologies have the same cost and one of them has a higher throughput, the metric would favor this one, indicating again that this topology is the most efficient one.

In order to compare the different topologies with this new metric, we use the normalized cost presented in Table 5.3 and the throughput results shown in Section 5.4.3. First, we show the results for uniform traffic in Figure 5.10. The figure presents this new metric for FF, SADP, *DESTRO* and *RUFT* for all the studied network sizes without additional mechanisms to provide in-order delivery of packets in the adaptive algorithms. In Section 5.4.3, we showed that the ordering of the topologies in descending order of performance was *DESTRO*, SADP, *RUFT*, and FF. With the new metric, FF still is the worst network configuration. In addition, we can observe that the cost per flit of FF is increased as the network size is increased, since the resource cost of the topology is the highest and its throughput gets even worse as the size of the network is increased. In particular, FF increases the SADP cost per flit in 3.8% for the 4-ary 2-tree, 6.1% for the 8-ary 2-tree, 7.6% for the 4-ary 4-tree, and 15% for the 8-ary 3-tree.

On the other hand, SADP cost per flit is the second worst one. SADP reaches a very good throughput for this traffic pattern, but requires too much resources. *DESTRO* has a slightly lower cost per flit than SADP, but the difference between them is almost constant because they require almost the same resources and obtain similar throughput. Nevertheless, on the contrary to the results that only consider performance, the best network configuration is *RUFT*. Despite that *RUFT* can not reach the same throughput than SADP and *DESTRO*, it strongly reduces the resources required to build the network, resulting in the lowest cost per flit of all the analyzed topologies. Compared to *RUFT*, *DESTRO* has a 60.6% higher cost per flit for the 4-ary 2-tree. This difference even increases to 76.8% in the 8-ary 3-tree.

Figure 5.11 presents similar results but using complement traffic. In this

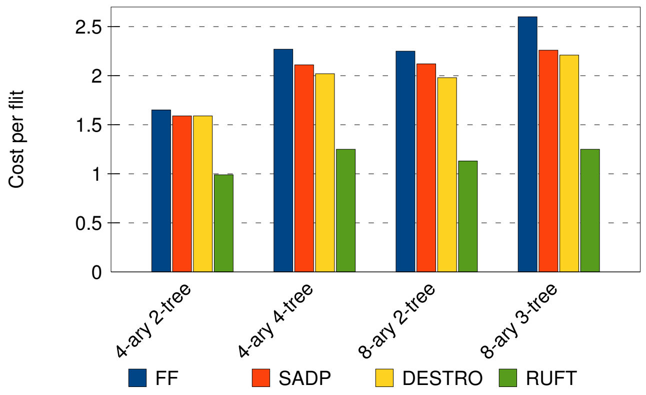


Figure 5.10: Cost required in each network to deliver a flit per node per cycle for uniform traffic without in-order delivery of packets.

traffic pattern, *RUFT* was the network configuration that reached the highest throughput, followed by *DESTRO*, SADP, and FF. In terms of cost per flit, FF is clearly the worst network configuration again. It even has a cost per flit that is more than twice the cost for SADP in the 4-ary 4-tree. The differences between SADP and *DESTRO* are equivalent to the ones observed in the performance evaluation due to their similar cost. But, again, *RUFT* can be considered the best network configuration, since it does not only achieve the best performance, it also achieves it with the lowest cost per flit. For example, the cost per flit of *DESTRO* is 80% higher than the one of *RUFT* one for the 4-ary 2-tree, this difference grows to 85%, 121%, and 125% for the 4-ary 4-tree, the 8-ary 2-tree, and the 8-ary 3-tree, respectively.

However, as we have commented previously, complement traffic supposes one of the best cases for *RUFT* and *DESTRO*. In order to make a more complete and fair comparison, we also analyze the worst traffic pattern for *RUFT*, that is, bit reversal. The results for the bit reversal traffic pattern are presented in Figure 5.12. As expected, the worst network configurations are *DESTRO* and *RUFT*. This is due to their very poor performance with this particular traffic pattern. As can be seen, the cost per flit of *DESTRO* is 112.3% higher than the cost per flit of SADP in the best case. Furthermore, in the worst case, the cost per flit of *DESTRO* is 659.2% higher than the cost per flit of SADP.

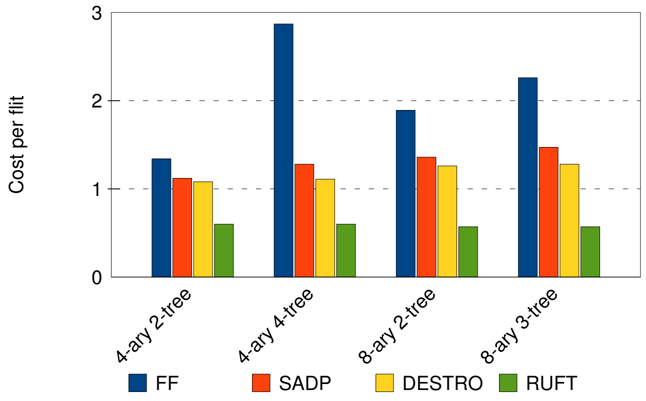


Figure 5.11: Cost required in each network to deliver a flit per node per cycle for complement traffic without in-order delivery of packets.

RUFT also shows a very bad cost/performance ratio for this traffic pattern. However, its low cost reduces significatively the cost per flit, specially when we compare it against *DESTRO*. Nevertheless, instead of comparing *RUFT* against *DESTRO*, we compare it against the best network configuration, that is, SADP. Concretely, the cost per flit of *RUFT* is 18.1% in the best case, which corresponds to the 4-ary 2-tree. At worst, the cost per flit of *RUFT* is 327.6% higher than the one from SADP, corresponding to the 4-ary 4-tree. However, as commented, these differences are for this particular traffic pattern. So, in order to provide a more realistic comparison, we also show the cost/performance results for the traces used in Section 5.4.3.

Nevertheless, as we have commented previously, using the time required to deliver all the messages that contain the trace is not a good metric because the traces do not indicate the correspondence between the packets, so our simulator injects the packets at the timestamps indicated in the trace file without knowing if a packet is a response to another packet which has not arrived yet. For this same reason, we can not rely on throughput, which is calculated as the total number of delivered data units divided by the total time. However, the average latency of the packets can be used to compare the different network configurations. For this reason, we use a different metric in the cost/performance evaluation for the traces. Instead, of using the cost per flit per node per cycle like above, we use the product of the normalized cost

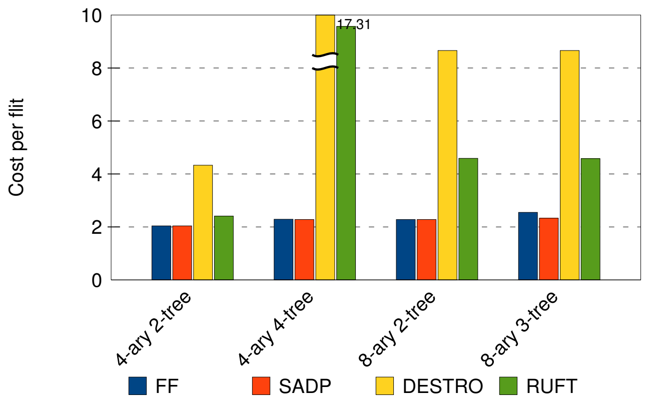


Figure 5.12: Cost required in each network to deliver a flit per node per cycle for bit reversal traffic without in-order delivery of packets.

Table 5.4: Normalized cost multiplied by total simulation and latency for different traces.

	Trace #1			**Trace #2**			**Trace #3**		
	T. Time	Net. Lat.	Gen. Lat.	T. Time	Net. Lat.	Gen. Lat.	T. Time	Net. Lat.	Gen. Lat.
DESTRO	1	1	1	1	1	1	1	1	1
RUFT	0.59	0.55814	0.59472	0.59	0.55401	0.59177	0.59	0.55401	0.58764
SADP	1.06	47.6788	2.36274	1.06	22.0798	1.0971	1.06	37.71162	1.31334
	Trace #4			**Trace #5**			**Trace #6**		
	T. Time	Net. Lat.	Gen. Lat.	T. Time	Net. Lat.	Gen. Lat.	T. Time	Net. Lat.	Gen. Lat.
DESTRO	1	1	1	1	1	1	1	1	1
RUFT	0.59	0.51684	0.58823	0.59	0.6018	0.47495	0.58882	0.58823	0.56581
SADP	1.06	2.29384	1.09922	1.15116	16.65366	2.81006	1.17342	20.37744	6.21584

of the network and the average packet latency. In this way, if two networks have the same cost, but one has a lower latency, the product will indicate that the network with the lower latency is the most efficient one. In the same way, if the average packet latency of two networks is the same, the product will indicate that the most efficient one is the network whose cost is the lowest one. Notice that this metric is the product of the cost of the network and its average latency, so the lowest it is, the better because the ideal network is the one with the lowest cost and the lowest latency. Finally, we have followed the same procedure with the total time.

This metric can be explained as the latency or time to deliver all the messages that each network configuration would have if they are built up with exactly the same amount of resources. On the other hand, it can be also approximated as the cost that each network configuration would require to achieve the same latency or simulation time. Using this second approach, we can clearly see in Table 5.4 that SADP would always require to use more resources to achieve the same simulation time than *DESTRO*, as it achieves the same time than *DESTRO* for traces from 1 to 4, but using more resources (the ones corresponding to the selection function). In the other two traces, SADP could not achieve the same simulation time than *DESTRO*, so it requires to use even more resources than in the other traces to be able to complete the traces in the same time than *DESTRO*. Concretely, at least 17% more resources for trace 6. Furthermore, we can see that SADP requires to use 130% more resources than *DESTRO* to achieve the same network latency in the most relaxed trace, and would require to use 46.7 times more resources than *DESTRO* to achieve the same network latency in the worst case. Regarding latency from generation, the resources required by SADP to obtain the same results than *DESTRO* at worst are only 5.2 higher. This huge difference between the resources required by SADP to obtain the same results than *DESTRO* in terms of network latency and latency from generation was expected since they follow the same trend than the results shown in Section 5.4.3.

Finally, we can observe in Table 5.4 that *RUFT* requires approximately 40% less resources than *DESTRO* to obtain the same trace processing time. Moreover, in order to achieve the same network latency than *DESTRO*, *RUFT* only requires between 40% and 49% less resources. And, approximately the same results can be observed for latency from generation. The cost$\times$latency ratio of *RUFT* almost doubles the ones from the different fat-trees. Hence, *RUFT* is the most efficient topology when dealing with the traces. As commented in Section 5.4.3, *RUFT* does not only reduce the average latency of the packets in most of the traces, it obtains such results with almost half the resources than a fat-tree, as has been seen in this section.

5.5 Conclusions

In this chapter, we have presented a new topology derived from the fat-tree topology that reduces to the half its resources providing slightly smaller performance results. This topology has been created taking in mind simplifying the routing logic of the switches that implement *DESTRO*. This new topology is referred to as *RUFT*, Reduced Unidirectional Fat-Tree, since it completely removes the downwards phase of the fat-tree, becoming an unidirectional MIN. *RUFT* replaces the downwards phase of the fat-tree by a set of links that directly connect the switches of the last stage of the network to the corresponding destination nodes. Moreover, routing in this topology is very simple, since only a component of the packet destination gives the switch output port. In this way, packet headers are as small as in distributed routing, and routing logic at the switches is as fast and simple as in source routing.

The evaluation has shown that *RUFT* can obtain similar performance to the fat-tree with *DESTRO* when using synthetic traffic patterns. It reaches 15% lower throughput than *DESTRO* at worst in uniform traffic, but reduces the latency at zero load by a 8.7%. For complementary traffic, *RUFT* outperforms all the other network configurations, and the difference grows with the network size. For this traffic pattern, *RUFT* is always able to deal with the injected traffic. Finally, bit reversal is the worst case for *RUFT*, as happened with *DESTRO*. In this traffic pattern, the effective bandwidth of the network is reduced by a factor of k, being k the number of up links of the switches. Regarding the evaluation with traces, *RUFT* achieves the same results than *DESTRO*, and clearly outperforms the adaptive algorithms.

However, we have to take into account that *RUFT* obtains these results by using half the resources of the fat-tree configuration. For this, the results that take into account the cost of the network increase the differences in benefit of *RUFT*. In particular, the results for the cost/performance ratio in synthetic traffic has shown that *DESTRO* (the best network configuration in the performance evaluation) has even a 76.8% higher cost/performance ratio than *RUFT* in uniform traffic, and 125% in complement traffic. Finally, the traces has shown that *RUFT* halves the cost/performance ratio of the other topologies in the worst case. So, *RUFT* is the most efficient topology in terms of cost/performance ratio.

To conclude, we must remark that these results are not only applicable to *RUFT*. We have compared *RUFT* against the fat-tree topology because this unidirectional MIN has came up of our previous research. From non-minimal *DESTRO*, it was clear that the downwards phase was not necessary. However, these results may be also applicable to other similar unidirectional MINs like the unidirectional butterfly. That is, this chapter can help in comparing the effectiveness of bidirectional MINs against unidirectional ones.

Chapter 6

Conclusions

"The truth may be out there, but the lies are inside your head."

Terry Pratchett, Hogfather.

In this final chapter, we present the conclusions of this dissertation and a brief list of points that will be addressed in the future. Finally, we also expose the results in terms of scientific publications and other contributions derived from the work presented in this dissertation.

6.1 Conclusions

In this dissertation, we have focused on proposing different techniques to improve fault-tolerance, performance and cost in the fat-tree topology, one of the most-commonly used topologies for cluster-based machines. The proposed techniques rely on very simple solutions that require none or low hardware overhead.

First, we have developed a dynamic fault-tolerant routing strategy (FT^{2}EI) that can tolerate the maximum number of faults that preserves the connectivity in a fat-tree topology. It is based on extending the Interval Routing scheme by introducing exclusion intervals, therefore it requires minimum hardware cost, only two registers per ascending output port, contrary to previous works based on switch and link replication. Moreover, this strategy introduces the minimum performance degradation since it is based on exploiting the rich connectivity available in the fat-tree topology and only the paths that traverse the faults are eliminated, allowing minimal fully adaptive routing through all non-faulty paths. Additionally, a dynamic mechanism has been developed in order to update the exclusion intervals dynamically without losing packets and without stopping the machine activity. Moreover, its reconfiguration time is several orders of magnitude smaller than the mean time between faults of current systems.

Next, we have developed a deterministic routing algorithm for fat-trees (*DESTRO*) that can achieve similar performance results to the ones obtained with the adaptive routing algorithm commonly-used in fat-trees, but with the advantages of being a deterministic algorithm, that is, a simple hardware implementation, reducing routing time and preserving in-order delivery of packets. Moreover, it is able to obtain even better performance results than the best adaptive routing algorithm if the packet in-order delivery is mandatory. The proposed deterministic routing algorithm is based on using at each switch the ascending output port given by the destination component of the packet that is being routed corresponding to the switch stage. This routing algorithm is able to evenly balance network traffic and, more importantly, it reduces to the minimum the number of paths that share each link, and as a consequence, it reduces network contention. Moreover, we have presented a compact implementation of the routing algorithm using Flexible Interval Routing (FIR).

Finally, we have developed a simplified topology derived from the fat-tree (*RUFT*) by taking advantage of the particular characteristics of the deterministic routing algorithm. This topology almost halves the resources required by the fat-tree topology. In terms of performance, the new topology obtains similar results or slightly lower ones than the fat-tree. But if we consider the cost/performance ratio, the new topology almost halves the one of the fat-tree. Moreover, routing in this topology is very simple, only a component of the packet destination gives the switch output port. In this way, the routing mixes the good properties from distributed routing (small packet headers) and the ones from source routing (fast routing times and simple routing logic at the switches).

Summing up, this dissertation has made three proposals focused on the fat-tree topology directed to provide fault-tolerance and high performance, but with low hardware and developing costs. These proposals are especially convenient for the large-scale machines that are being developed nowadays whose sizes reinforce the need of fault-tolerance while keeping high performance and a scalable cost.

6.2 Future Work

Despite our yearning for completeness, as the given time to develop the work presented in this dissertation is finite, there are a few points for each of our proposals that could not be addressed and will be addressed in the future.

Regarding FT^2EI, the points that can be addressed in the future are:

- When the reconfiguration of the exclusion intervals has been completed, the utilization of the up links of the switches can be unevenly distributed due to the destinations whose packets cannot be longer routed through some ascending links, so we pretend to develop a mechanism to balance the traffic in the network after the reconfiguration of the exclusion intervals. For example, the uneven utilization of links can be observed in a switch with two up links, one of them does not have any active exclusion interval, while the another one does. In this switch, packets whose destination are included in the exclusion interval can only use the link without exclusion intervals, while packets whose destinations are not included in the exclusion interval can use any of them. In this way, the link without exclusion interval has a higher utilization than the link with exclusion interval.
- The strategy has been developed to work with distributed routing, but it can be adapted to other routing paradigms like centralized and source routing.

Regarding *DESTRO*, the pending points that can be addressed in the future are:

- We pretend to provide a deterministic fault-tolerant routing algorithm that can achieve the same degree of fault-tolerance than FT^2EI.

- *DESTRO* has been designed specifically for the fat-tree topology. However, we think that it can be extended to work with any other MIN providing the same performance results.

Finally, regarding *RUFT*:

- As commented in Section 5.5, we think that the conclusions of Chapter 5 can be applicable to other unidirectional MINs. For this, we pretend to extend the evaluation to comprise other unidirectional MINs like the unidirectional butterfly.

- We would like to create a more accurate model of the influence of the final cluster packaging on the length of the links that connect the switches of the last stage to the destination nodes.

6.3 Contributions

While conducting the research necessary for the work presented in this dissertation, we have the opportunity to publish several papers in scientific conferences and journals, having the chance of receiving very useful feedback from many reviewers. Following, we present the list of published papers for each of the proposals of this dissertation.

The publications in conferences corresponding to FT^2EI are [58–60]:

- C. Gómez, M.E. Gómez, P. López, and J. Duato. *"An Efficient Fault-Tolerant Routing Methodology for Fat-tree Interconnection Networks"*. Fifth International Symposium on Parallel and Distributed Processing and Applications (ISPA07). August 2007. ISBN 978-3-540-74741-3. Pages 509–522. Awarded with the best student paper.

- C. Gómez, M.E. Gómez, P. López, and J. Duato. *"A Dynamic and Compact Fault-Tolerant Strategy for Fat-tree"*. IFIP International Conference on Network and Parallel Computing (NPC06). October 2006.

- C. Gómez, M.E. Gómez, P. López, and J. Duato. *"FT2EI: A Compact Fault-Tolerant Routing Strategy for Fat-trees with Exclusion Intervals"*. XVII Jornadas de Paralelismo. September 2006. ISBN 84-690-0551-0.

In addition, there is a published paper in a journal corresponding to FT^2EI [109]:

- C. Gómez, M.E. Gómez, P. López, and J. Duato. *"FT2EI: A Dynamic Fault-Tolerant Routing Methodology for Fat Trees with Exclusion Intervals"*. IEEE Transactions on Parallel and Distributed Systems. Vol. 20. 2008. ISSN 1045-9219. Pages 802-817. IEEE Computer Society.

The publications in conferences corresponding to *DESTRO* are [48, 54]:

- C. Gómez, F. Gilabert, M.E. Gómez, P. López, and J. Duato. *"Deterministic versus Adaptive Routing in Fat-Trees"*. Workshop on Communication Architecture on Clusters (CAC'07), as a part of IPDPS'07. March 2007. ISBN 978-1-4244-0909-9.

- F. Gilabert, C. Gómez, M.E. Gómez, P. López, and J. Duato. *"A New Deterministic Routing Algorithm for Fat-Trees"*. II Congreso Español de Informatica - XVIII Jornadas de Paralelismo. October 2007. ISBN 978-84-9732-593-6.

In addition, there is a submitted journal:

- C. Gómez, F. Gilabert, M.E. Gómez, P. López, and J. Duato. *"A New Mechanism for Selecting the Output Port in Fat-trees Topologies"*. pending of acceptation on IEEE Transactions on Parallel and Distributed Systems. IEEE Computer Society.

There are also two other international contributions regarding *DESTRO*:

- C. Gómez, F. Gilabert, M.E. Gómez, P. López, and J. Duato. *"Fat-Trees for HyperTransport^{TM}Interconnects"*. White paper for the HyperTransport Consortium.

- USA patent under review with number US 11/845,813.

Finally, we have published the following articles in conferences regarding *RUFT* [49, 56, 57, 80, 81]:

- C. Gómez, F. Gilabert, M.E. Gómez, P. López, and J. Duato. *"RUFT: Simplifying the Fat-Tree Topology"*. 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS'08). December 2008. ISSN 1521-9097.

- D. Ludovici, F. Gilabert, S. Medardoni, C. Gómez, M.E. Gómez, P. López, G.N. Gaydadjiev, and D. Bertozzi. *"On the feasibility of fat-tree topologies for Networks-on-chip"*. Design, Automation & Test in Europe Conference & Exhibition (DATE '09). April 2009. ISSN 1530-1591.

- C. Gómez, F. Gilabert, M.E. Gómez, P. López, and J. Duato. *"RUFT: Reduced Unidirectional Fat–Tree"*. XX Jornadas de Paralelismo. September 2009. ISBN 84-9749-346-8.

- F. Gilabert, C. Gómez, M.E. Gómez, P. López, and J. Duato. *"On the feasibility of fat-tree topologies for Netowrks-on-chip"*. XX Jornadas de Paralelismo. September 2009. ISBN 84-9749-346-8.

- D. Ludovici, F. Gilabert, C. Gómez, M.E. Gómez, P. López, and G.N. Gaydadjiev. *"Buttery vs. Unidirectional Fat-Trees for Networks-on-Chip: not a Mere Permutation of Outputs"*. 3rd Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip, The 4th International Conference on High Performance and Embedded Architectures and Compilers. January 2009.

We have also published a journal article regarding *RUFT* [55]:

- C. Gómez and F. Gilabert and M.E. Gómez and P. López and J. Duato. *"Beyond Fat-tree: Unidirectional Load–Balanced Multistage Interconnection Network"*. IEEE Computer Architecture Letters. July-Dec. 2008., vol. 7, number 2. ISSN 1556-6056.

Bibliography

[1] Advanced switching core architecture specification. Available at http://www.asi-sig.org/specifications for ASI SIG Members only.

[2] Asci red web site. http://www.sandia.gov/ASCI/Red/.

[3] Bluegene general information. Available online http://www.top500.org/wiki/index.php/Blue_Gene.

[4] Earth simulator center. http://www.es.jamstec.go.jp/esc/eng/index.html.

[5] Google platform at wikipedia, the free encyclopedia. Available online: http://en.wikipedia.org/wiki/Google_platform.

[6] Grupo de arquitecturas paralelas de la universidad politcnica de valecia (parallel architectures group of the technical university of valencia. http://www.gap.upv.es/.

[7] Infiniband® trade association. http://www.infinibandta.org/.

[8] Myricom®. http://www.myri.com/.

[9] Quadrics. http://www.quadrics.com/.

[10] Roadrunner web site. http://www.lanl.gov/roadrunner/.

[11] System availability statictics. Available online http://www.nersc.gov/nusers/status/AvailStats/FY08.

[12] Top500 supercomputers site. http://www.top500.org.

[13] The asci q system: 30 teraops capability at los alamos national laboratory, 2002.

[14] Simulation of hafnium gate material, February 2007. Available online.

[15] Bulent Abali, Craig B. Stunkel, Jay Herring, Mohammad Banikazemi, Dhabaleswar K. Panda, Cevdet Aykanat, and Yucel Aydogan. Adaptive routing on the new switch chip for ibm sp systems. *J. Parallel Distrib. Comput.*, 61(9):1148–1179, 2001.

[16] Thomas William Ainsworth and Timothy Mark Pinkston. On characterizing performance of the cell broadband engine element interconnect bus. In *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, pages 18–29, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Marina Alonso, Juan-Miguel Martínez, Vicente Santonja, Pedro López, and José Duato. Power saving in regular interconnection networks built with high-degree switches. *Parallel and Distributed Processing Symposium, International*, 1:5b, 2005.

[18] Bob Amos, Sanjay Deshpande, Mike Mayfield, and Frank OConnell. Rs/6000 sp 375mhz power3 smp high node, 2000.

[19] Semiconductor Industry Association. International technology roadmap for semiconductors: 1999 edition. Disponible en `http://www.itrs.org`, 1999.

[20] E. Bakker, J. van Leeuwer, and R.B. Tan. Linear interval routing. *Algorithms review*, 2:45–61, 1991.

[21] James Balfour and William J. Dally. Design tradeoffs for tiled cmp on-chip networks. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 187–198, New York, NY, USA, 2006. ACM.

[22] A. Banerjee, R. Mullins, and S. Moore. A power and energy exploration of network-on-chip architectures. *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pages 163–172, May 2007.

[23] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. Qnoc: Qos architecture and design process for network on chip. *J. Syst. Archit.*, 50(2-3):105–128, 2004.

[24] Rajendra V. Boppana and Suresh Chalasani. A comparison of adaptive wormhole routing algorithms. *SIGARCH Comput. Archit. News*, 21(2):351–360, 1993.

[25] A. Broder, M. Fischer, R. Dolev, and B. Simons. Efficient fault-tolerant routings in networks. *Proc. of the 16th annual ACM Symp. on Theory of Computing*, 1984.

[26] Suresh Chalasani and Rajendra Boppana. Fault-tolerance with multimodule routers. *High-Performance Computer Architecture, International Symposium on*, 0:201, 1996.

[27] Suresh Chalasani and Rajendra V. Boppana. Fault-tolerant wormhole routing in tori. In *ICS '94: Proceedings of the 8th international conference on Supercomputing*, pages 146–155, New York, NY, USA, 1994. ACM.

[28] S. Chalsani, C. Raghavendra, and A. Varma. Fault-tolerant routing in min based supercomputers. *Proc. of the 4th Int. Conf. on Supercomputing*, 1990.

[29] Srini Chari. A total cost of ownership study (tco) comparing the ibm bluegene/p with other cluster systems for high performancecomputing. White paper, IBM and Cabot Partners, June. Available online (13 pages).

[30] F. T. Chong, J. Thomas, and F. Knight Jr. Design and performance of multipath min architectures. *Proc. of the 4th annual ACM Symp. on Parallel Algorithms and Architectures*, 1992.

[31] Matteo Dall'Osso, Gianluca Biccari, Luca Giovannini, Davide Bertozzi, and Luca Benini. xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor socs. In *ICCD '03: Proceedings*

of the 21st International Conference on Computer Design, page 536, Washington, DC, USA, 2003. IEEE Computer Society.

[32] W. J. Dally. Virtual-channel flow control. *Proceedings of the 17th annual International Symposium on Computer Architecture*, pages 60–68, 1990.

[33] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[34] W.J. Dally. Express cubes: improving the performance of k-ary n -cube interconnection networks. *Computers, IEEE Transactions on*, 40(9):1016–1023, Sep 1991.

[35] Andre DeHon, Tom Knight, and Henry Minsky. Fault tolerant design for multistage routing networks. Technical report, Cambridge, MA, USA, 1990.

[36] Zhu Ding, Raymond R. Hoare, Alex K. Jones, and Rami Melhem. Level-wise scheduling algorithm for fat tree interconnection networks. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 96, New York, NY, USA, 2006. ACM.

[37] J. Duato and P. López. Bandwidth requirements for wormhole switches: A simple and efficient design. In *2nd Euromicro Workshop on Parallel and Distributed Processing (2nd EWPDP)*, IEEE Computer Society Press, Mlaga, Espaa, 1994.

[38] J. Duato, A. Robles, F. Silla, and R. Beivide. A comparison of router architectures for virtual cut-through and wormhole switching in a now environment. *J. Parallel Distrib. Comput.*, 61(2):224–253, 2001.

[39] José Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Trans. Parallel Distrib. Syst.*, 4(12):1320–1331, 1993.

[40] José Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[41] C.B. Stunkel et al. The sp2 communication subsystem. *Technical report, IBM T.J. Watson Research Center*, August 1994.

[42] Mark J. Karol et al. Input versus output queueing on a space-division packet switch. *IEEE Transactions on Communications*, COM-35(12):1347–1356, 1987.

[43] N. Kamiura et al. Design of a fault-tolerant multistage interconnection network with parallel duplicated switches. *Proc. of the 15th IEEE Int. Symp. on Defect and Fault-Tolerance in VLSI Systems*, 2000.

[44] J. Flich, M.P. Malumbres, P. López, and J. Duato. Improving routing performance in myrinet networks. *Parallel and Distributed Processing Symposium, International*, 0:27, 2000.

[45] W. Denzel D. Craddock N. Ni W. Rooney T. Engbersen R. Luijten R. Krishnamurthy (IBM) J. Duato G. Pfister, M. Guzat. Solving hot spot contention using infiniband architecture congestion control,. In *High Performance Interconnects for Distributed Computing (HPI-DC)*, 2005.

[46] Israel Gazit and Miroslaw Malek. Fault tolerance capabilities in multistage network-based multicomputer systems. *IEEE Trans. Comput.*, 37(7):788–798, 1988.

[47] Patrick Geoffray and Torsten Hoefler. Adaptive routing strategies for modern high performance networks. In *HOTI '08: Proceedings of the 2008 16th IEEE Symposium on High Performance Interconnects*, pages 165–172, Washington, DC, USA, 2008. IEEE Computer Society.

[48] F. Gilabert, C. Gómez, M.E. Gómez, P. López, and J. Duato. A new deterministic routing algorithm for fat-trees. October 2007.

[49] F. Gilabert, C. Gómez, M.E. Gómez, P. López, and J. Duato. On the feasibility of fat-tree topologies for netowrks-on-chip. In *XX Jornadas de paralelismo*, September 2008.

[50] F. Gilabert, M.E. Gómez, P. López, and J. Duato. On the infuence of the selection function on the performance of fat-trees. In *Euro-Par*, Dresden, Germany, August 2006.

[51] Francisco Gilabert, Simone Medardoni, Davide Bertozzi, Luca Benini, María Engracia Gómez, Pedro López, and José Duato. Exploring high-dimensional topologies for noc design through an integrated analysis and synthesis framework. *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, 0:107–116, 2008.

[52] Christopher J. Glass and Lionel M. Ni. Fault-tolerant wormhole routing in meshes without virtual channels. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):620–636, 1996.

[53] Marius Gligor, Nicolas Fournel, and Frdric Ptrot. Adaptive dynamic voltage and frequency scaling algorithm for symmetric multiprocessor architecture. *Digital Systems Design, Euromicro Symposium on*, 0:613–616, 2009.

[54] C. Gómez, F. Gilabert, M.E. Gómez, P. López, and J. Duato. Deterministic versus adaptive routing in fat-trees. In *Workshop on Communication Architecture on Clusters, as a part of IPDPS'07*, page 235, March 2007.

[55] C. Gómez, F. Gilabert, M.E. Gómez, P. López, and J. Duato. Beyond fat–tree: Unidirectional load–balanced multistage interconnection network. *Computer Architecture Letters*, 7(2):49–52, July-Dec. 2008.

[56] C. Gómez, F. Gilabert, M.E. Gómez, P. López, and J. Duato. Ruft: Reduced unidirectional fat–tree. In *XX Jornadas de paralelismo*, September 2008.

[57] C. Gómez, F. Gilabert, M.E. Gómez, P. López, and J. Duato. Ruft: Simplifying the fat-tree topology. In *14th IEEE International Conference on Parallel and Distributed Systems, 2008. ICPADS '08.*, pages 153–160, Dec. 2008.

[58] C. Gómez, M.E. Gómez, P. López, and J. Duato. A dynamic and compact fault-tolerant strategy for fat-tree. In *IFIP International Conference on Network and Parallel Computing*. IFIP, October 2006.

[59] C. Gómez, M.E. Gómez, P. López, and J. Duato. Ft^{2}ei: A compact fault-tolerant routing strategy for fat-trees with exclusion intervals. In *XVII Jornadas de Paralelismo*, September 2006.

[60] C. Gómez, M.E. Gómez, P. López, and J. Duato. An efficient fault-tolerant routing methodology for fat-tree interconnection networks. In *Fifth International Symposium on Parallel and Distributed Processing and Applications (ISPA07)*, pages 509–522, August 2007.

[61] M. E. Gómez, P. López, and J. Duato. A memory-effective routing strategy for regular interconnection networks. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 41.2, Washington, DC, USA, 2005. IEEE Computer Society.

[62] M. E. Gómez, P. López, and J. Duato. Fir: an efficient routing strategy for tori and meshes. *J. Parallel Distrib. Comput.*, 66(7):907–921, 2006.

[63] M.E. Gómez, N.A. Nordbotten, J. Duato, J. Flich, P. López, A. Robles, O. Lysne, and T. Skeie. A routing methodology for achieving fault-tolerance in direct networks. 55(4):400–415, 2005.

[64] Ronald I. Greenberg and Charles E. Leiserson. Randomized routing on fat-trees. In *Advances in Computing Research*, pages 345–374. JAI Press, 1996.

[65] Hennessy and Patterson. Computer architecture: A quantitative approach, 4^{a} edición. *Morgan Kauffman*, 2006.

[66] Anna R. Karlin, Greg Nelson, and Hisao Tamaki. On the fault tolerance of the butterfly. In *In Proc. of the 26th ACM Symp. on Theory of Computing (STOC)*, pages 125–133, 1994.

[67] J. Kim, J. Balfour, and W.J. Dally. Flattened butterfly topology for on-chip networks. *Computer Architecture Letters*, 6(2):37–40, July-December 2007.

[68] John Kim, William J. Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *ISCA '07: Proceedings*

of the 34th annual international symposium on Computer architecture, pages 126–137, New York, NY, USA, 2007. ACM.

[69] Ken Koch. Roadrunner platform overview. Roadrunner Technical Seminar Series, June. Available online.

[70] S. Konstantinidou. The selective extra stage butterfly. *Transactions on Very Large-Scale Integration Systems*, 1993.

[71] V. P. Kumar and S. M. Reddy. Design and analysis of fault-tolerant multistage interconnection networks with low link complexity. *SIGARCH Comput. Archit. News*, 13(3):376–386, 1985.

[72] J-C. Laprie. Dependable computing and fault tolerance : Concepts and terminology. In *Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on*, pages 2–, Jun 1995.

[73] M. Lecine. Proceedings of the sf's terascale computing system. In *7th Workshop on Distributed Supercomputing(SOS7)*, 2003.

[74] T. H. Lee and J. J. Chou. Some directed graph theorems for testing the dynamic full access property of multistage interconnection networks. *IEEE TENCON*, 1993.

[75] F. Thomson Leighton, Bruce M. Maggs, and Ramesh K. Sitamaran. On the fault tolerance of some popular bounded-degree networks. In *SIAM Journal on Computing*, pages 542–552, 1995.

[76] C. E. Leiserson. Fat-trees: Universal networks hardware-efficient supercomputing. *IEEE Trans. on Computers*, 34(10), 1985.

[77] R. Libeskind-Hadas and E. Brandt. Origin-based fault-tolerant routing in the mesh. *High-Performance Computer Architecture, International Symposium on*, 0:102, 1995.

[78] Xuan-Yi Lin, Yeh-Ching Chung, and Tai-Yi Huang. A multiple lid routing scheme for fat-tree-based infiniband networks. *Parallel and Distributed Processing Symposium, International*, 1:11a, 2004.

[79] Tao Liu, Qiuyang Li, and Yulu Yang. Advanced baseline: A new min with fault-tolerance characteristic. *International Symposium on Parallel Architectures, Algorithms, and Networks*, 0:0275, 2002.

[80] D. Ludovici, F. Gilabert, C. Gómez, M. E. Gómez, P. López, and G. Gaydadjiev. Buttery vs. unidirectional fat-trees for networks-on-chip: not a mere permutation of outputs. In *3rd Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip, The 4th International Conference on High Performance and Embedded Architectures and Compilers*, January 2009.

[81] D. Ludovici, F. Gilabert, S. Medardoni, C. Gómez, M.E. Gómez, P. López, G.N. Gaydadjiev, and D. Bertozzi. Assessing fat-tree topologies for regular network-on-chip design under nanoscale technology constraints. In *Design, Automation and Test in Europe Conference and Exhibition, 2009. DATE '09.*, pages 562–565, April 2009.

[82] Olav Lysne and José Duato. Fast dynamic reconfiguration in irregular networks. *Parallel Processing, International Conference on*, 0:449, 2000.

[83] Anne MacFarland. ebay infrastructure – a prototype for the future. *The Clipper Group Observer*, November, 2006.

[84] M.M.K. Martin, M.D. Hill, and D.A. Wood. Token coherence: decoupling performance and correctness. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 182–193, June 2003.

[85] J. C. Martínez, J. Flich, A. Robles, P. López, and J. Duato. Supporting adaptive routing in iba switches. *J. Syst. Archit.*, 49(10-11):441–456, 2003.

[86] J. C. Martínez, J. Flich, A. Robles, P. Lǿpez, J. Duato, and Michihiro Koibuchi. In-order packet delivery in interconnection networks using adaptive routing. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 101, Washington, DC, USA, 2005. IEEE Computer Society.

[87] S. Medardoni, D. Bertozzi, L. Benini, and E. Macii. Control and datapath decoupling in the design of a noc switch: area, power and performance implications. *System-on-Chip, 2007 International Symposium on*, pages 1–4, Nov. 2007.

[88] Giovanni De Micheli and Luca Benini. Networks on chips: Technology and tools. *Morgan Kauffman*, 2006.

[89] Michael Miller. How google works. Technical report, 2007.

[90] José Miguel Montañana. *Efficient Mechanisms to Provide Fault Tolerance in Interconnection Networks for PC Clusters.* PhD thesis, Universidad Politécnica de Valencia, July 2008.

[91] J. Morrison. The asci q system at los alamos. In *7th Workshop on Distributed Supercomputing(SOS7)*, 2003.

[92] Trevor Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, pages 52 – 58, Abril 2001.

[93] Shubhendu S. Mukherjee, Peter Bannon, Steven Lang, Aaron Spink, and David Webb. The alpha 21364 network architecture. *IEEE Micro*, 22(1):26–35, 2002.

[94] Y. Mun and H. Y. Youn. On performance evaluation of fault-tolerant multistage interconnection networks. *Proc. of the 1992 ACM/SIGAPP Symp. on Applied Computing*, 1992.

[95] Cisco Networks. Introduction to gigabit ethernet. Available online: `http://www.cisco.com/en/US/tech/tk389/tk214/tech_brief09186a0080091a8a.html`.

[96] Sotiris E. Nikoletseas, Grammati E. Pantziou, Panagiotis Psycharis, and Paul G. Spirakis. On the fault tolerance of fat-trees. In *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*, pages 208–217, London, UK, 1997. Springer-Verlag.

[97] S. F. Nugent. The ipsc/2 direct-connect communications technology. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 51–60, New York, NY, USA, 1988. ACM.

[98] Sunish Parikh and Thomas E. Martinez. Dual processors, hyper-threading technology, and multi-core systems. Available at `http://www.intel.com/cd/ids/developer/asmo-na/eng/200677.htm`.

[99] Sudeep Pasricha and Nikil Dutt. On–chip communication architectures: System on chip. *Morgan Kauffman*, 2008.

[100] W.W. Peterson and D.T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, Jan. 1961.

[101] F. Petrini and M. Vanneschi. k-ary n-tress: High performance networks for massively parallel architecture. *IEEE Micro*, 15, Feb. 1995.

[102] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, 2002.

[103] Timothy Mark Pinkston, Ruoming Pang, and José Duato. Deadlock-free dynamic reconfiguration schemes for increased network dependability. *IEEE Trans. Parallel Distrib. Syst.*, 14(8):780–794, 2003.

[104] Dhiraj K. Pradhan, editor. *Fault-tolerant computer system design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[105] V. Puente, J. A. Gregorio, R. Beivide, and F. Vallejo. A low cost fault tolerant packet routing for parallel computers. *Parallel and Distributed Processing Symposium, International*, 0:45a, 2003.

[106] V. Puente and J.A. Gregorio. Immucube: Scalable fault-tolerant routing for k-ary n-cube networks. *Parallel and Distributed Systems, IEEE Transactions on*, 18(6):776–788, June 2007.

[107] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. De Micheli, and L. Benini. Bringing nocs to 65 nm. *IEEE Micro*, 27(5):75–85, Sept.-Oct. 2007.

[108] D. Reed. High-end computing: The challenge of scale. *Director's Colloquium*, May 2004.

[109] Crispin Gómez Requena, Maria Engracia Gómez Requena, Pedro Juan López Rodriguéz, and José Francisco Duato Marín. Ft2ei: A dynamic fault-tolerant routing methodology for fat trees with exclusion intervals.

[110] C. Ruemmler and J. Wilkes. Unix disk access patterns. *Winter Usenix Conference*, 1993.

[111] N. Santoro and R. Khatib. Routing without routing tables. *Tech. report SCS-TR-6, School of Computer Science, Carleton University*, 1982.

[112] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Sa']-ferthwaite, and Charles P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9, 1991.

[113] S.L. Scott and G. Thorson. The cray t3e network: Adaptive routing in a high performance 3d torus. *Proceedings of the Symposium on High Performance Interconnects*, August 1996.

[114] M. Seager. Operational machines: Asci white. In *7th Workshop on Distributed Supercomputing(SOS7)*, 2003.

[115] F.O. Sem-Jacobsen, T. Skeie, O. Lysne, and J. Duato. Dynamic fault tolerance with misrouting in fat trees. *Proc. of International Conference on Parallel Processing*, 2006.

[116] F.O. Sem-Jacobsen, T. Skeie, O. Lysne, O. Torudbakken, E. Rongved, and B.Johnsen. Siamese-twin: A dynamically fault-tolerant fat-tree. *Proc. Int. Parallel and Distributed Processing Symp.*, 2005.

[117] J. Sengupta and P. Bansal. Fault-tolerant routing in irregular mins. *IEEE Region 10 Int. Conf. on Global connectivity in Energy, Computer, Communication and Control*, 2, 1998.

[118] J. Sengupta and P. Bansal. High speed dynamic fault-tolerance. In *Electrical and Electronic Technology, 2001. TENCON. Proceedings of*

IEEE Region 10 International Conference on, volume 2, pages 669–675 vol.2, 2001.

[119] Li Shang, Li-Shiuan Peh, and N.K. Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 91–102, Feb. 2003.

[120] Stephen Shankland. Asci white overshadows supercomputer, 1999.

[121] N. Sharma. Fault-tolerance of a min using hybrid redundancy. *Proc. of the 27th Annual Simulation Symp.*, 1994.

[122] J.P. Shen and J.P. Hayes. Fault-tolerance of dynamic-full-access interconnection networks. *IEEE Transactions on Computers*, 33(3):241–248, 1984.

[123] Tor Skeie. Handling multiple faults in wormhole mesh networks. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, pages 1076–1088, London, UK, 1998. Springer-Verlag.

[124] C.B. Stunkel, D.G. Shea, D.G. Grice, P.H. Hochschild, and M. Tsao. The sp1 high-performance switch. *Proc. of the Scalable High-Performance Computing Conf.*, 1994.

[125] R. Suzuki, S. Fukumoto, and K. Iwasakio. Adaptive checkpointing for timewarp technique with a limited number of checkpoints. *Int. Conf. on Distributed Computing Systems Workshops*, 2002.

[126] IBM BG/L Team. An overview of bluegene/l supercomputer. In *ACM Supercomputing Conference*, 2002.

[127] Nian-Feng Tzeng, Pen-Chung Yew, and Chun-Qi Zhu. A fault-tolerant scheme for multistage interconnection networks. *SIGARCH Comput. Archit. News*, 13(3):368–375, 1985.

[128] M. Valerio, L. Moser, and P. Melliar-Smith. Fault-tolerant orthogonal fat-trees as interconnection networks. *Proc. 1st Int. Conf. on Algorithms and Architectures for Parallel Processing*, 1995.

[129] A. Varma and C. Raghavendra. Fault-tolerant routing in multistage interconnection networks. *IEEE Trans. on Comput.*, 38(3):385–393, 1992.

[130] A. Vishnu, M. Koop, A. Moody, A. R. Mamidala, S. Narravula, and D. K. Panda. Hot-spot avoidance with multi-pathing over infiniband: An mpi perspective. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 479–486, Washington, DC, USA, 2007. IEEE Computer Society.

[131] Hangsheng Wang, Li-Shiuan Peh, and S. Malik. Power-driven design of router microarchitectures in on-chip networks. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 105–116, Dec. 2003.

[132] P.T. Wolkotte, G.J.M. Smit, G.K. Rauwerda, and L.T. Smit. An energy-efficient reconfigurable circuit-switched network-on-chip. *Proceedings on the 19th IEEE International Parallel and Distributed Processing Symposium*, pages 155a–155a, April 2005.

[133] Chuan-Lin Wu and Tse-Yun Feng. On a class of multistage interconnection networks. *Computers, IEEE Transactions on*, C-29(8):694–702, Aug. 1980.