Topology Agnostic Methods for Routing, Reconfiguration and Virtualization of Interconnection Networks

Åshild Grønstad Solheim

February 14, 2012

Acknowledgements

A number of people have supported me during the work with this PhD thesis. First and foremost, the advice, constructive criticism and encouragement provided by my supervisors, Olav Lysne and Tor Skeie, have been invaluable. I am deeply grateful for their willingness to share knowledge and ideas; for their friendliness and patience; and for their open door policy. I would also like to thank Ingebjørg Theiss – my third supervisor for the first 18 months at Simula Research Laboratory – for giving me a thorough introduction to the research area of interconnection networks.

My colleagues and friends at Simula Research Laboratory are an exceptionally likeable, interesting and knowledgeable group of people from various parts of the world, and a list of those of my colleagues who mean a great deal to me would have contained too many names to be included here. I will, however, mention two of my colleagues in particular. Their contributions have significantly improved the quality of the research papers upon which this thesis is based, as well as the quality of the thesis itself. Thomas Sødring has been a close collaboration partner, inspiring colleague and supportive friend. I thank him for useful discussions over numerous cups of tea; for being prepared to share his constant stream of ideas; for co-authoring research papers; and for developing main parts of our simulator models. (In addition, on behalf of my entire family and myself, I am grateful for having obtained the recipe for Thomas's delicious pink cheesecake.) Sven-Arne Reinemo has always been willing to answer my questions, provide useful comments and give helpful advice, and he is also a co-author of two of the research papers upon which this thesis is based. I would also like to thank the remaining co-authors for their contributions, and the members of the IT support team for always being friendly and ready to help.

This thesis could not have been written without the funding from Simula Research Laboratory.

My husband and best friend, Pål Grønstad Solheim, noticed the advertisement for a vacant PhD position at Simula Research Laboratory, encouraged me to apply, and has since been most enthusiastic and supportive. As always, my parents, Anne Synnøve and Jens Jørgen Grønstad, have been very important supporters. Fortunately, my family and friends have shown understanding for the necessity of working long hours while pursuing a PhD.

Contents

Chapter 1

Introduction

Modern computing systems, such as supercomputers, data centers and multicore processor integrated circuit chips, generally require e cient communication between their di erent system units; tolerance towards component faults; flexibility to expand or merge; and a high utilization of their resources. Interconnection networks are used in a variety of such computing systems in order to enable communication between their diverse system units. Supercomputers, such as those ranked by the Top500 list [237], typically host traditional high performance computing applications, examples of which include simulations of medical, meteorological and geological processes. Important tasks for an interconnection network of a supercomputer include the provisioning of low latency and high throughput communication between a set of computing nodes that cooperate in executing a parallel application. Data centers, such as those operated by Google [85] and Amazon [9], are often large facilities which include huge collections of computing and storage resources. Such systems face a number of challenges, some of which are related to reliability, resource utilization and energy-e ciency. This set of challenges is also highly relevant for a multicore (also known as manycore) processor environment, where an interconnection network embedded in the integrated circuit chip links cores to each other as well as to other units such as cache memory. (An example of a multicore chip is Intel's [106] terascale prototype processor [100, 244] which has 80 cores.) In general, the various units of a computing system can be connected in a number of ways, and interconnection networks can thus assume many di erent structures (topologies).

Most traditional routing algorithms are relatively simple mechanisms designed for a specific regular network structure. A main topic of this thesis is topology agnostic routing, which does not assume a particular network structure. Topology agnostic routing algorithms were initially motivated by interconnection network technologies – such as Autonet [204] and Myrinet [31] – that allowed the units of a computing system to be connected in an arbitrary (irregular) manner. When compared to routing algorithms tailored to specific topologies, early topology agnostic routing algorithms (such as the one proposed in [204]) typically achieved an inferior performance. More recent topology agnostic routing algorithms perform significantly better than their predecessors, however.

Topology agnostic routing algorithms are essential for the operation of systems that include interconnection networks with irregular topologies. Furthermore, such algorithms play an important part in the provisioning of fault tolerance in systems that include interconnection networks with regular topologies. As no particular network structure is assumed, topology agnostic routing algorithms are also usable for the non-regular topologies that may result from faults in originally regular topologies.

For a single network component (such as a switch or communication channel) the expected time between failures may be relatively long. Nevertheless, the risk of a faulty network component increases when computing systems grow larger. A faulty network component could be present at the start-up of a computing system, or occur while the system is in operation. The former situation requires static fault handling, whereas the latter situation requires dynamic fault handling and is in general more challenging. An attractive quality of topology agnostic routing algorithms is their inherent static fault tolerance. Unplanned topology changes, such as network component faults, are not the only type of topology change that may occur over the lifetime of a computing system. Topology agnostic routing algorithms also provide flexibility with respect to planned topology changes, such as extensions of existing interconnection networks or mergers of computing systems with di erent interconnection network solutions.

Unplanned or planned changes in the topology of an interconnection network could disrupt communication between the various units of a computing system. A new routing function is needed if – as a result of a topology change – an existing routing function can no longer provide connectivity between the communicating system units. The change-over from one routing function to another is referred to as reconfiguration, and is a main topic of this thesis. An e cient reconfiguration strategy is important for the maintenance of a predictable network service after the occurrence of a topology change.

Due to strict requirements on high system throughput and low packet latency, interconnection networks in modern communication systems must in general support lossless communication. Thus, in order to avoid packet loss due to a lack of bu er space, flow control is exercised over each communication channel. Such a flow control mechanism entails a risk of packet deadlock. (In a deadlocked set of packets none of the packets can advance until another packet in the set advances.) A deadlock generally has a devastating e ect on the performance of a computing system. Deadlock recovery methods, such as [13, 116], aim to resolve an established deadlock. Unfortunately, for many applications deadlock recovery methods may cause a system to stall for an unacceptably long period of time. Thus, a deadlock must in general be avoided. The requirement of deadlock avoidance is a major challenge in the design of routing and reconfiguration algorithms for interconnection networks. On the purpose to avoid deadlock, routing and reconfiguration algorithms typically enforce restrictions on packet transmission, and some of these restrictions may significantly reduce the performance of a computing system.

Applications running in multiprocessor systems such as supercomputers, data centers or multicore processor integrated circuit chips are typically parallel, which means that separate tasks are executed concurrently on a set of di erent computing nodes. Such a set of computing nodes – often referred to as a partition – is selected and assigned by a processor allocation algorithm. As processor allocation involves aggregation of a set of physical computing nodes into a virtual processor that is assigned to execute a parallel application, it is a kind of virtualization. It is also a main topic of this thesis. E cient processor allocation is an important challenge for multiprocessor systems that host a number of di erent parallel applications at a time. A high utilization of a system's computing resources; short queuing times and running times for applications; as well as isolation of tra c between partitions (tra c-containment) are examples of desirable qualities of a processor allocation algorithm. Some of these qualities are hard to combine. In particular, many processor allocation algorithms achieve tra c-containment only at the cost of a significantly reduced utilization of a system's computing resources.

1.1 Contributions

Investigation and proposal of new or improved solutions to topology agnostic routing and reconfiguration of interconnection networks are main objectives of this thesis. In addition, topology agnostic routing and reconfiguration algorithms are utilized in the development of new and flexible approaches to processor allocation. We aim to present versatile solutions that can be used for the interconnection networks of a number of di erent computing systems, including supercomputers, data centers and multicore processor integrated circuit chips.

The main focus of this thesis is on performance issues. Nevertheless, e cient methods for routing, reconfiguration and processor allocation are also important for the realization of energy-e cient computing systems. A manifestation of the overall increased awareness of environmental challenges in high performance computing is the emergence of the Green500 list [88] which ranks supercomputers according to their energy-e ciency.

1.1.1 Routing

The Advanced Switching Interconnect (ASI) specification [17] did not prescribe a particular routing algorithm. On the purpose to select a suitable routing algorithm for the ASI technology [150], we perform a systematic evaluation of a number of existing routing algorithms. We state a set of criteria that a routing algorithm for ASI should comply with, and show that one of the existing topology agnostic routing algorithms fulfils all of the criteria. This routing algorithm – Layered Shortest Path (LASH) [144, 218] – is recommended to be used with ASI. We demonstrate the abilities of LASH as a static fault tolerance mechanism for mesh and torus topologies. Moreover, we introduce an optimization which allows LASH to achieve as high performance for mesh topologies as one of the established routing algorithms tailored for mesh topologies achieves. In addition, we demonstrate how the optimization allows LASH to maintain its qualities as a static fault tolerance mechanism. After our study was conducted, the ASI specification process was discontinued. Nevertheless, the ASI technology is now incorporated in Dolphin Express [122], and the results of our study are thus still of interest for this technology. In addition, we explain how most of our considerations regarding routing in ASI are also relevant for routing in InfiniBand [105].

1.1.2 Reconfiguration

Reconfiguration is a deadlock prone process, and some existing reconfiguration strategies include deadlock avoidance mechanisms that significantly reduce the network service o ered to running applications. We propose a new dynamic reconfiguration algorithm, RecTOR, which supports a replacement of the routing function without causing performance penalties. RecTOR is conceptually simple, and targets systems that apply the flexible and topology agnostic Transition-Oriented Routing (TOR) algorithm [200]. As RecTOR does not require complex network switches, it is also simple to realize.

Overlapping Reconfiguration (OR) [139, 140] is a versatile and highly e cient reconfiguration algorithm that was previously only usable in distributed routing systems. We introduce an adaptation that allows using OR also in source routing systems (such as Dolphin Express). Furthermore, we investigate how the performance of OR is influenced by di erent levels of synchronization of the initiation of a reconfiguration process. In addition, we propose an optimization of OR which,

during a reconfiguration of a source routing system, aims to provide a better network service to the running applications.

1.1.3 Processor allocation (virtualization)

Processor allocation strategies that guarantee tra c-containment commonly pose strict requirements on the shape of partitions, and thus achieve only a limited utilization of a system's computing resources. We introduce two new approaches to tra c-contained processor allocation which are more flexible, and which are also conceptually simple. Our first approach is a new processor allocation algorithm – UDFlex – for systems that use the Up*/Down* routing algorithm [204]. Our second approach is a framework for developing new processor allocation algorithms. Both approaches utilize the properties of a topology agnostic routing algorithm in order to enforce tra c-containment within arbitrarily shaped partitions. Consequently, our approaches achieve a high resource utilization as well as isolation of tra c between di erent applications. To di erent degrees, both of our approaches depend on reconfiguration. When compared to the use of a traditional processor allocation strategy (such as one that allocates sub-mesh shaped partitions in a mesh or torus topology) the use of either UDFlex or our framework is expected to entail a certain amount of allocation and communication overhead. We assess the amount of such overhead that can be tolerated before the advantages of UDFlex over sub-mesh allocating strategies are neutralized. In addition, for a large set of arbitrarily shaped partitions, each executing a communication intensive application, we investigate costs and benefits of tra c-containment (as prescribed by our framework) versus non-tra c-containment.

1.2 Thesis organization

Chapter 2 provides background information on various topics within the area of interconnection networks that are relevant to this thesis. Examples include topology, flow control, deadlock, fault models, routing, reconfiguration and topology agnosticism. In addition, processor allocation and virtualization are introduced in Chapter 2. Chapter 3 presents the research methods used in our studies and provides a general description of our simulator models. (The particular setups of our simulation experiments are explained in Chapters 4, 5 and 6 for each of the individual studies.) Chapter 4 presents our endeavours to realize e cient routing for the ASI/Dolphin Express technology. Our contributions to the field of reconfiguration are found in Chapter 5, whereas Chapter 6 presents our studies on processor allocation. Each of Chapters 4, 5 and 6 also includes a discussion on related work and our contribution; a critical view on our own research; and possible paths for future research. Finally, Chapter 7 summarizes our conclusions and plans for future research.

1.3 Publications

The research results presented in Chapters 4, 5 and 6 of this thesis are mainly based on the following publications.

- Å. G. Solheim, O. Lysne, T. Skeie, T. Sødring, I. Theiss, and I. Johnson. Routing for the ASI Fabric Manager. *IEEE Communications Magazine*, 44(7):39–44, July 2006.

- Å. G. Solheim, O. Lysne, A. Bermúdez, R. Casado, T. Sødring, T. Skeie, and A. Robles-Gómez. E cient and deadlock-free reconfiguration for source routed networks. In *9th Workshop on Communication Architecture for Clusters*, 2009.
- Å. G. Solheim, O. Lysne, and T. Skeie. RecTOR: A new and e cient method for dynamic network reconfiguration. In *15th International Euro-Par Conference*, pages 1052–1064, 2009.
- Å. G. Solheim, O. Lysne, T. Sødring, T. Skeie, and J. A. Libak. Routing-contained virtualization based on Up*/Down* forwarding. In *14th International Conference on High Performance Computing*, pages 500–513, 2007.
- Å. G. Solheim, O. Lysne, T. Skeie, T. Sødring, and S.-A. Reinemo. A framework for routing and resource allocation in network virtualization. In *16th International Conference on High Performance Computing*, pages 129–139, 2009.
- O. Lysne, S.-A. Reinemo, T. Skeie, Å. G. Solheim, T. Sødring, L. P. Huse, and B. D. Johnsen. Interconnection networks: Architectural challenges for utility computing data centers. *IEEE Computer*, 41(9):62–69, September 2008.

Chapter 2

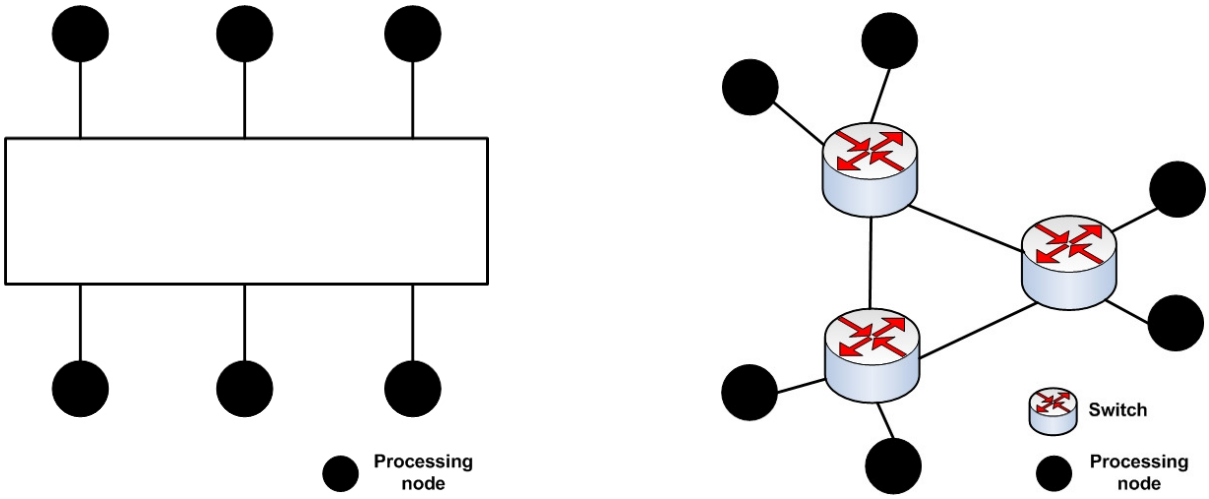
Interconnection networks

This chapter provides background information on various topics within the area of interconnection networks that are of relevance to this thesis. The term *interconnection network* normally refers to a high capacity communication network of limited physical extent that provides high throughput, low latency and lossless communication. This is also the definition assumed in this thesis (although the two main textbooks on interconnection networks [57,68] do not distinguish between interconnection networks and computer networks in general).

Interconnection networks are used in a variety of computing systems in order to enable communication between their diverse system units. For instance, such networks are used to interconnect computing nodes in a cluster or supercomputer; processing units, memory modules and input/output devices within a computer; ports and processing elements within a router or switch; cores and caches on a multicore chip; and computing nodes, storage devices and gateways towards external networks in a data center. For simplicity, we collectively refer to the diverse communicating system units as *processing nodes*.

The extent of interconnection networks typically spans from small networks on integrated circuit chips (NoCs) up to networks as large as local-area networks. Often, interconnection networks are classified according to their structure, and the main categories used by [68] are shared-medium, direct, indirect and hybrid networks. Shared-medium networks share a common multidrop bus; both direct and indirect networks are point-to-point switched structures; whereas hybrid networks contain elements both from the shared-medium and point-to-point switched networks. A multidrop bus is a common but simple interconnection network where all the processing nodes share a single communication medium, the bus. Only one processing node at a time can utilize the bus for message transmission, and a message is broadcast to all the processing nodes connected to the bus. Thus, a bus interconnection network prohibits parallel transmission of messages. This limits performance as the size of computing systems increases. A point-to-point switched interconnection network, on the other hand, encourages parallel transmission of messages, and thereby supports scalability and higher performance. In such a network, a physical communication channel connects two nodes, where a node is either a processing node, a switch, or a combined node (which contains both processing and switching elements). A message may traverse one or more intermediate nodes on its path from a source to destination node. Figures 2.1(a) and 2.1(b) illustrate a bus and a point-to-point switched interconnection network, respectively.

Historically, most interconnection networks used a shared communication medium. Nowadays, however, point-to-point switched networks are widespread. The focus of this thesis is on



(a) Bus network.

(b) Point-to-point switched network.

Figure 2.1: Examples of a bus and point-to-point switched interconnection network.

point-to-point switched networks, not on shared-medium or hybrid networks. A number of different network technologies exist that support the construction of point-to-point switched interconnection networks, including Ethernet [102, 206], InfiniBand [105] and Dolphin Express [122]. Point-to-point switched networks can be assembled in a number of di erent manners, and a variety of strategies exist for transportation of data through such networks. An interconnection network's structure, flow control and switching strategy, as well as its routing algorithm, are factors which may have a significant influence on the performance of a computing system. Sections 2.1, 2.2 and 2.5 provide background information on each of these issues.

The risk of packet deadlock is an important challenge for interconnection networks that provide lossless communication. Section 2.3 discusses how the deadlock risk arises from dependencies between the bu ers of di erent communication channels in such networks.

Reliable interconnection networks are essential for the operation of current high-performance computing systems. The ability to e ciently reconfigure an interconnection network is an important challenge in the e ort to support a reliable network service. A reconfiguration is needed in order to restore a connected routing function when changes in the structure of an interconnection network have caused a disconnection of the present routing function. Such changes could be the result of component faults, and Section 2.4 gives an overview of the most relevant fault models for interconnection networks. Furthermore, Section 2.6 introduces the concept of reconfiguration, which is one of the main topics of this thesis.

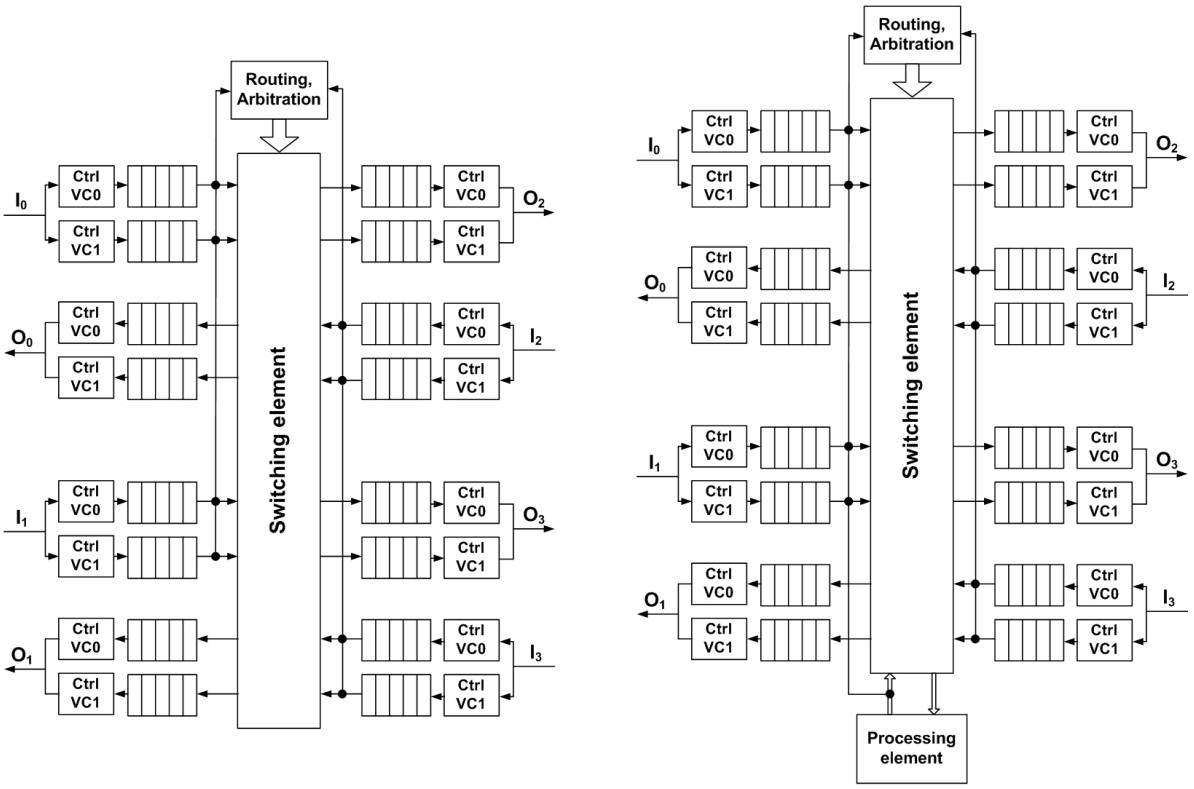
Another main topic of this thesis – processor allocation (virtualization) – is presented in Section 2.7. Processor allocation involves assignment of sets of processing nodes to execute parallel jobs that run concurrently in a multiprocessor system such as a supercomputer, multicore chip or data center. The processor allocation strategy has impact on the running and queuing times of individual jobs, as well as on the utilization of resources and throughput of jobs in a system. Processing nodes are connected by an interconnection network; many existing processor allocation strategies are conscious of network issues such as topology and routing (see e.g. [251, 255]); and the methods developed in this thesis are routing aware (but topology agnostic). Therefore, we chose to include the background information on processor allocation into this chapter on interconnection networks.

The main objective of this thesis is to investigate and propose methods for routing, reconfiguration and processor allocation that do not assume a particular topology for the interconnection network. Therefore, Section 2.8 provides a discussion on topology agnostic methods, including an overview of existing strategies.

A large part of this chapter is based on information found in [57, 68]. In addition, information has been derived from [171].

2.1 Topology

The nodes and communication channels of a point-to-point switched interconnection network can be connected in a variety of patterns, and the structure of a network is referred to as the network's *topology*. A node is either a switch, a processing node or a combined node, where a combined node comprises a switching element and one or more processing elements.



(a) Switch. (b) Combined node.

Figure 2.2: A typical switch and combined node architecture.

Figure 2.2(a) gives an example of a switch architecture. Switches may either be input buffered, output buffered or both. In this thesis, unless otherwise stated, we assume that switches contain both input (ingress) buffers and output (egress) buffers. Figure 2.2(b) illustrates a corresponding combined node with one processing element (details regarding buffering and control between the processing and switching elements are not shown). Both the switch and combined node have four bidirectional ports, where the input physical channels are labelled $I_0 - I_3$ and the output physical channels are labelled $O_0 - O_3$. Each physical channel has two virtual channels (VCs) [54], labelled *VC0* and *VC1*, with a separate control unit (*Ctrl*). The routing and arbitration unit decides

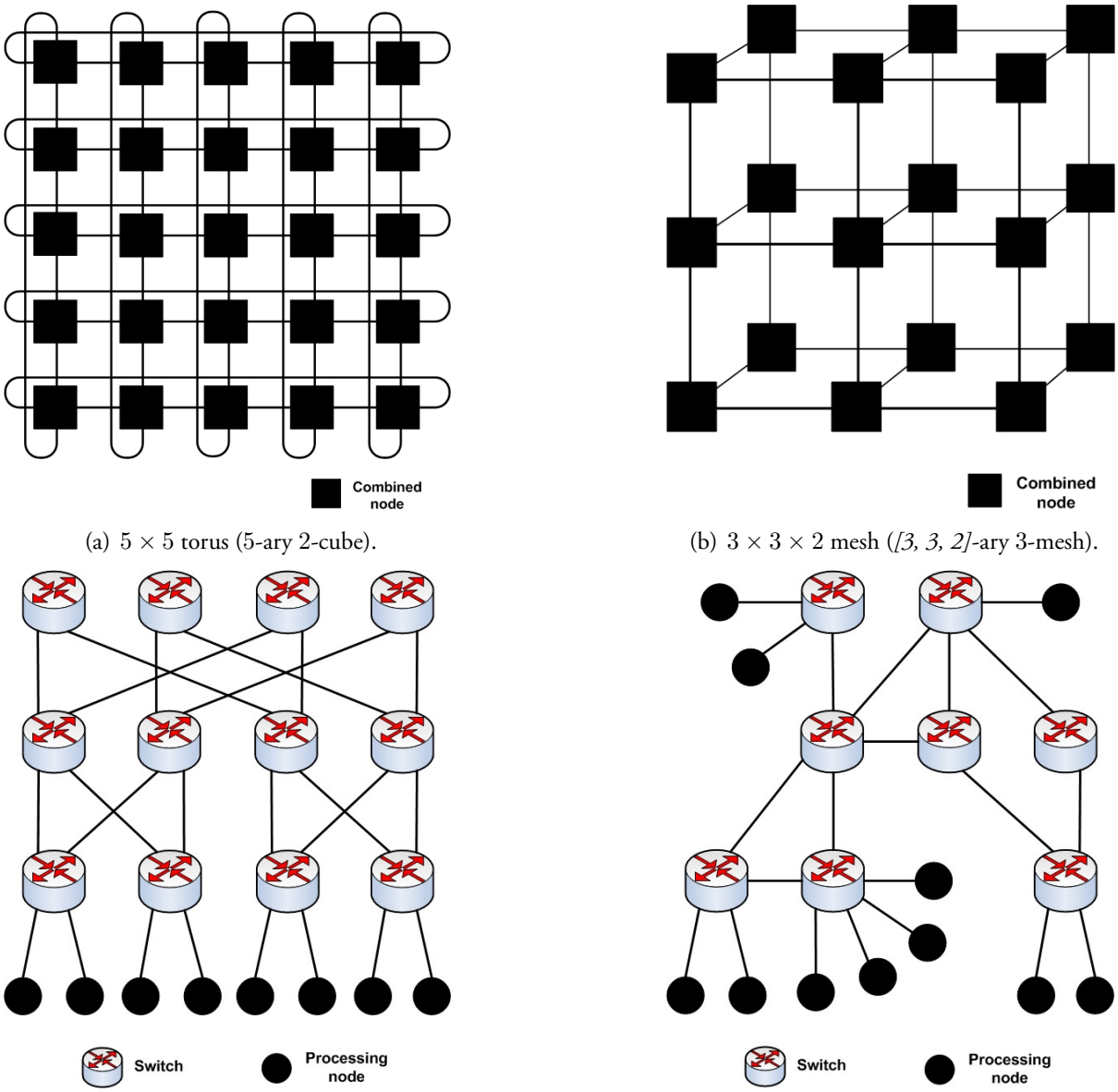
which ingress to egress connections are set up by the switching element. Some switches implement *virtual output queuing*, which means that an input buffer is organized in such a way that packets bound for different output channels are enqueued separately. The purpose of such a buffer organization is to avoid *head-of-line blocking*. Head-of-line blocking occurs when a packet bound for one output channel blocks subsequent packets bound for other vacant output channels. (Virtual output queuing is not illustrated in Figure 2.2.)

The topology of an interconnection network is often visualized as a graph. The mapping of an interconnection network I onto a graph G is then expressed formally as $I = G(N, C)$, where the vertexes N and the edges C of the graph represent, respectively, the network's switching elements and the communication channels that connect these switching elements. The graph may be directed or undirected, depending on whether the communication channels are unidirectional or bidirectional. The graph representation eases the reasoning about various issues related to the topology of an interconnection network, such as connectivity, path lengths, path diversity and bisection bandwidth (all of which will be discussed in this section). Furthermore, the graph representation is advantageous as it enables the use of graph theory in the development of solutions to many of the challenges faced by interconnection networks – such as deadlock avoidance, routing and reconfiguration (all of which will be discussed in later sections).

Interconnection networks can assume a number of different topologies, and Figure 2.3 illustrates a few of the alternatives. Torus topologies are often called k-ary n-cubes (sometimes a mesh is also referred to as a k-ary n-cube or k-ary n-mesh). Such topologies consist of n dimensions with k switching elements in each dimension, where the radix, k, can be different for each dimension. Figures 2.3(a) and 2.3(b) depict a 5×5 torus (5-ary 2-cube) and $3 \times 3 \times 2$ mesh ($[3, 3, 2]$-ary 3-mesh), respectively. Meshes and tori are described in [68] as *strictly orthogonal* topologies, as the switching elements are arranged in an n-dimensional grid with k nodes in each dimension, and every switching element is an intersection between all dimensions. Assume that in each dimension each switching element is assigned a coordinate i, where $0 \leq i \leq k-1$. Then, a mesh is assembled by connecting switching element number i with the switching elements numbered $i-1$ and $i+1$, for each value of i and for each dimension. These connections are also used for the torus topology. In addition, a torus utilizes so called wraparound channels, which means that, in each dimension, the switching elements numbered 0 and $k-1$ are also interconnected. The wraparound channels effectively make a torus topology a set of interconnected rings.

Multistage interconnection networks are a group of networks that consist of several stages of switches where processing nodes are connected only to the first and final stages of switches (possibly only on one side of the network if it is bidirectional). Examples of multistage interconnection networks include fat-trees [126], Clos [48], Benês [24] and Delta [169] networks. Some of these subgroups of multistage interconnection networks overlap. For instance, according to [176], a k-ary n-tree [176] is both a class of fat-trees and a Delta network. A k-ary n-tree is constructed by interconnecting n stages of switches such that, except in the first and final stage, each switch connects to k switches in the previous stage and k switches in the next stage of the tree. A 2-ary 3-tree is depicted in Figure 2.3(c).

Multistage interconnection networks are often used for large high capacity systems, and, in general, they are expensive due to the high number of switches relative to the number of processing nodes. In addition, long cables are in general required. Expansion of such topologies either requires switches with a higher number of ports or a large number of additional switches. Torus and mesh



(a) 5×5 torus (5-ary 2-cube).

(b) $3 \times 3 \times 2$ mesh (*[3, 3, 2]*-ary 3-mesh).

(c) Fat-tree (2-ary 3-tree).

(d) Irregular.

Figure 2.3: Example topologies.

topologies, on the other hand, can be expanded without the requirement of larger switches. Furthermore, cable lengths can be kept short, and simple routing algorithms can be implemented in hardware (without the need for routing tables). The Top500 list of supercomputers [237] shows that, for the most powerful supercomputers, fat-trees and tori are common interconnection network topologies. Currently, many data centers are based on Ethernet [206], and, as this technology by default uses a spanning tree protocol for routing, tree topologies are frequently used for the interconnection network. Arguments for fat-trees as an appropriate topology for data centers based on Ethernet are presented in [8]. For on-chip interconnection networks, two-dimensional meshes and tori have been suggested as suitable topologies (see e.g. [56, 124]), and those were evaluated in [167] together with a few other candidate topologies.

In the interconnection network literature, network topologies with nodes and communication channels arranged according to one of a set of standard structures are commonly denoted *regular*.

Network topologies with no such standard structure, on the other hand, are commonly denoted *irregular*. We adopt this convention.[1] Examples of regular network topologies include meshes, tori, hypercubes (2-ary n-cubes), fat-trees, Delta, Clos and Benês networks.

A network of workstations [12] is a computer cluster consisting of a set of commodity computers interconnected by a number of switches and communication channels. The switches and communication channels are typically arranged in an arbitrary pattern. Thus, a network of workstations commonly has an irregular topology. Typically, for an irregular network topology, some of the switches have one or more processing nodes attached, whereas others are solely connected to other switches. Irregular network topologies are easily expandable as no particular structure must be maintained. Some issues, such as routing which is described in more detail in Section 2.5, are in general more challenging for an irregular topology than for a regular topology. Figure 2.3(d) gives an example of an irregular network topology. A non-regular (or semi-irregular) topology may also be the result of faults in switching elements or communication channels of an initially regular network topology.

Point-to-point switched interconnection networks are commonly categorized as either *direct* or *indirect* networks. All the nodes of a direct network are combined nodes (see Figure 2.2(b)) which include both switching and processing elements. In an indirect network, on the other hand, a switch and processing node are separate entities connected by a communication channel. Typical examples of indirect networks are multistage interconnection networks (where processing nodes are connected only to the first and final stages of switches) and networks of workstations (where individual computers are connected by switches). Torus, mesh, ring and hypercube topologies, on the other hand, are often realized using combined nodes, and are thus often direct networks. However, all direct networks can be implemented as indirect networks by replacing each combined node with a switch node and a processing node [57]. Thus, this thesis will not further emphasize the distinction between direct and indirect networks, and in the following a "switch" could also refer to the switching element of a combined node.

A topology is *connected* if at least one path exists from every source to every destination processing node. Faults in switches or communication channels could disconnect a previously connected topology. *Path diversity*, which quantifies the existence of alternative paths between sources and destinations in a network, is an important characteristic of a topology for several reasons, e.g. for tra c balancing and fault tolerance. Assume that m disjoint paths exist from a source processing node s to a destination processing node d. Then, if a flexible routing strategy is used and in-order packet delivery is not required, tra c balancing could be achieved by distributing messages across the m paths. Congestion might result if several di erent sources and destinations share a section of their paths. Such congestion could be prevented if alternative paths were used such that fewer sources and destinations shared sections of the paths. Assume that a fault in a switch or communication channel destroys a path p_0 from s to d. Then, if p_0 was the only available path from s to d, the topology has been disconnected by the fault, and s and d can no longer communicate. If, on the other hand, m disjoint paths existed from s to d before the fault, another path $(p_1 \dots p_{m-1})$ could be taken into use after the destruction of p_0.

The minimum length of a path from s to d is another important property of a network's topology. Path lengths are decided by the number of intermediate switches that must be traversed, and

[1] Even though according to the strict definition given in [68] a network topology is regular only if every switch has the same number of neighbours – which e.g. includes tori but excludes meshes.

the *diameter* of a topology is the length of the longest minimal path.

The number of communication channels that connect a switch to other switches is referred to as the *node degree*. A fully connected topology of a significant size, where each switch is connected to all other switches, is an example of a topology with a high node degree. A ring, on the other hand, is an example of a topology with a low node degree, as each switch is connected to only two other switches. When comparing two equally large network topologies where one has a high node degree and the other has a low node degree, the former most likely has a shorter diameter than the latter.

A bisection of the topology results from a supposed removal of the smallest set S_{min} of communication channels that would cut the topology into two equally large halves. The *bisection bandwidth* is the aggregated transmission capacity of the communication channels in S_{min}, and quantifies the amount of data that can be transferred from one part of the topology to the other within a certain time frame. As opposed to the bisection bandwidth, the *e ective bisection bandwidth* introduced in [97] takes the routing function into consideration.

Physically, interconnection networks can be implemented in a number of di erent ways, ranging from small networks on a silicon die, via larger networks spanning a number of circuit boards, to large networks interconnecting racks or containers of processing nodes and switches in a data center. Numerous restrictions and tradeo s apply for the physical implementation of an interconnection network. A few of them are related to issues such as the quality of the silicon chip manufacturing process, the pin count of a chip, the length and width of the communication channels on a circuit board, and the selection of the optical or electrical cables for connecting racks or containers. The physical layout of an interconnection network is an important challenge. It is, however, not within the scope of this thesis.

2.2 Flow control and switching

Due to strict requirements on high system throughput and low packet latency, packet loss and subsequent retransmission are in general not acceptable for an interconnection network.

Thus, flow control mechanisms are needed in order to ensure that data transfer is not started unless su cient bu er space is available at the receiving end. Such mechanisms are needed both for the transfer of data through a single switch and for the transfer of data from one switching/processing node to another.

This thesis refers to the mechanisms used for controlling data transfer from an ingress to an egress bu er within a switch as *switching* mechanisms (recall that we assume input and output bu ered switches), whereas the mechanisms used between two separate nodes are referred to as *flow control* mechanisms.

Switching and flow control are closely intertwined, which is illustrated by the di erent approaches taken to explain these concepts by the two main textbooks on interconnection networks [57, 68]. Whereas [57] refers to both the intra and inter node mechanisms as flow control (and does not even include the word "switching" in its index), switching is an emphasized term in [68].

A *message* is the unit of information input from a source processing node for transportation through the interconnection network and delivery to a destination processing node. The interconnection network segments a message into several *packets* if it is longer than the maximum allowed

packet size. A packet may be further segmented into several flow control units (*flits*[2]), which, as the name implies, is the unit of information on which flow control is enforced. A flit can be segmented into several physical units (*phits*), where one phit represents the width of a physical communication channel – that is, the amount of information that can be transmitted on a physical communication channel per clock cycle.

2.2.1 Switching mechanisms

Virtual cut-through [114] and wormhole [54] switching are common switching mechanisms for current interconnection networks. Like store-and-forward switching, and unlike circuit switching, virtual cut-through and wormhole switching do not reserve resources in advance of the data transmission.[3] On its path from a source to destination processing node, a packet must share the available bu er space of an intermediate switch with other packets that traverse the same switch. However, the virtual cut-through, wormhole and store-and-forward switching mechanisms di er with respect to the point in time when data transmission from an ingress to egress bu er of a switch can be started. Each packet carries the routing information in its header. Nevertheless, with store-and-forward switching the entire packet must have been received by an ingress bu er before the decision on which egress bu er to forward the packet to is taken. Virtual cut-through and wormhole switching, on the other hand, support reduced latency by making the routing decision as soon as the routing information contained in the packet header has been received. Thus, provided that su cient egress bu er space is available and that the packet is selected by the arbitration algorithm, the head of the packet can be forwarded through the switch before the tail of the packet has reached the ingress bu er.

Virtual cut-through, wormhole and store-and-forward switching also di er with respect to the flit size. Both virtual-cut through and store-and-forward switching require that the flit size equals the packet size. Then, when the head of a packet is blocked, the tail catches up and the entire packet is stored in the same bu er. Wormhole switching (which supports small and cheap bu ers and is currently a popular choice for on-chip networks), on the other hand, does not assume a maximum packet size, and each bu er typically does not have space for an entire packet. Thus, while the head of a packet is blocked, the packet may span several switches. Due to these di erent characteristics, ensuring deadlock-freedom is more challenging for wormhole switching than for virtual cut-through and store-and-forward switching. The deadlock issue will be further discussed in Section 2.3.

2.2.2 Flow control mechanisms

The most commonly used flow control mechanisms for interconnection networks ensure that an upstream switch or processing node defers transmission of flits until a su cient amount of bu er space is available to store the flits in a downstream node. This requires that a downstream node informs an upstream node about bu er space availability. The two most common flow control mechanisms for interconnection networks are called *credit-based* and *on/o* flow control [57].

With credit-based flow control the upstream node maintains a credit counter which indicates the amount of free bu er space in the downstream node, and thus the number of flits that can

[2] Also known as flow control digits.

[3] See e.g. [68] for a description of store-and-forward and circuit switching.

currently be transmitted. The credit counter is decremented according to the number of flits transmitted from the upstream node to the downstream node. The credit counter is incremented upon reception of credit updates from the downstream node, which are sent in order to communicate to the upstream node that bu er space has become available. Thus, with credit-based flow control, a significant amount of information is normally transmitted merely to update the credit counter.

On/o flow control is a simpler mechanism that reduces the transmitted amount of control information concerning available bu er space in the downstream node. The upstream node maintains a flag to indicate whether or not flits can currently be transmitted. If the value of the flag is *on* the upstream node can transmit flits, whereas transmission must be deferred if the value of the flag is *o* . The downstream node need only inform the upstream node when the value of the flag must be changed. The flag must be turned o if the amount of available bu er space decreases below a certain threshold. Likewise, the flag must be turned on if the amount of available bu er space increases above a certain threshold. The threshold values that decide when data transmission is switched on and o respectively, are not necessarily the same.

A number of VCs may share a single physical communication channel. In that case, all VCs transmit packets over the same physical communication channel, whereas each VC has a separate set of bu ers and a separate flow control mechanism [52] (as illustrated in Figure 2.2). This means that the upstream node keeps state information (a credit counter or flag) for each VC, and the downstream node transmits updates with regard to bu er availability for each VC. Thus, VCs enable other packets to bypass a blocked packet, and are useful e.g. for implementation of deadlock avoidance and service di erentiation mechanisms.

In this thesis, a *virtual layer* is a virtual network that consists of all the switches and processing nodes of the physical network. In addition, the virtual layer includes a subset of the VCs such that, for each physical communication channel, one VC in each direction belongs to the virtual layer. (A virtual layer is sometimes referred to as a *layer* for short.)

The flow control mechanism has an important impact on the performance of the interconnection network. A bad flow control mechanism could keep a communication channel idle for part of the time, and thus cause increased packet latency and decreased system throughput. In order to support a constant flow of data on a communication channel, the bu er size – and, for on/o flow control, the on/o threshold values – must be carefully set, considering such parameters as channel bandwidth, round-trip time and flit length.

Optimistic flow control mechanisms, such as ACK/NACK (acknowledgement/negative acknowledgement), are not commonly used for interconnection networks due to ine cient use of bu ers and communication channels. With ACK/NACK the upstream node does not keep track of free bu er space in the downstream node, and simply transmits a flit at any time. When bu er space is not available, the downstream node discards the flit and returns a NACK. Such a mechanism requires retransmission of the discarded flits and, as flits may be received out of order, reordering of the received flits in the downstream node.

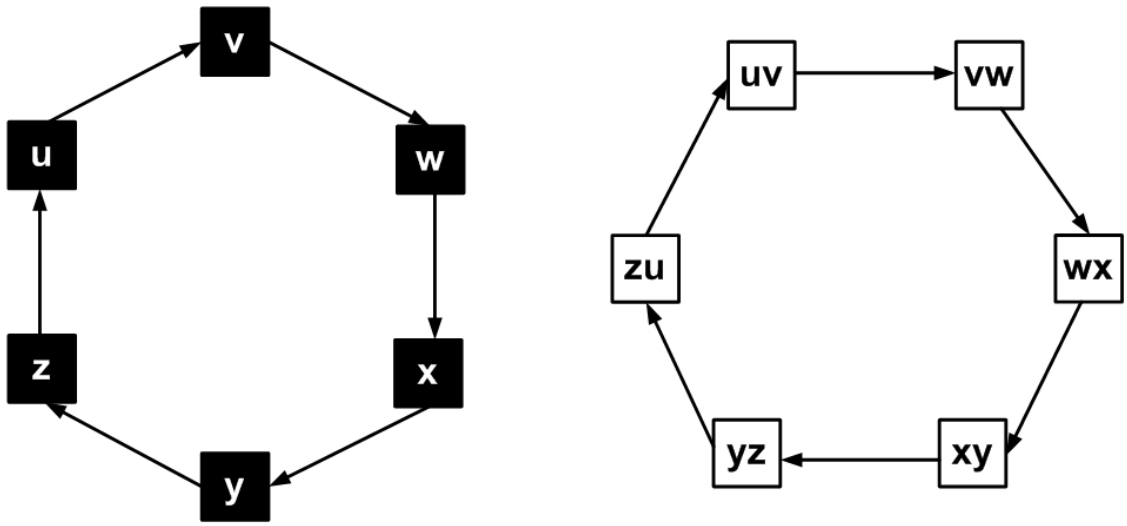
2.3 Channel dependency and deadlock

The flow control mechanism enforced on each communication channel in order to support lossless communication prevents a packet from moving until su cient bu er space is available on the receiving end of a communication channel. This causes a risk of deadlock as several packets may

be mutually waiting for each other to move. In a deadlocked set of packets none of the packets in the set can advance until another packet in the set advances. All the packets in the set are thus blocked forever, and the blockage spreads as packets in the deadlocked set block subsequently arriving packets. Such a situation is not consistent with a successful operation of an interconnection network.

Another potential problem caused by lossless flow control arises in the case of congestion. As the flow control mechanism does not allow a packet to advance until free bu er space is available on the receiving end of a communication channel, saturation trees [178] may form. That is, the congestion spreads out in the network, possibly also a ecting tra c not heading towards the original point of congestion. Congestion management mechanisms are studied e.g. in [66, 87, 92]. Although congestion control is an important topic for interconnection networks, it is not a main topic of this thesis.

In the following, we first give an introduction to the theory concerning deadlock, before two alternative approaches to handling this problem are presented. In this thesis, we do not focus on high-level deadlocks that may result from a lack of bu er management for request-reply protocols used e.g. for memory read or write operations.



(a) Ring network.

(b) Channel dependency graph.

Figure 2.4: A 6-node unidirectional ring network and the corresponding channel dependency graph.

If a flit can use a communication channel c_b immediately after a communication channel c_a there is a direct *channel dependency* [55] from c_a to c_b. A channel dependency expresses that the advancement of a flit from c_a to the next channel c_b depends on available bu er space in c_b. A *channel dependency graph* [55], which visualizes all the direct channel dependencies of an interconnection network, is a useful aid for reasoning about deadlock-freedom. The vertexes and edges of this directed graph represent the communication channels of the interconnection network and the dependencies between these channels, respectively. For wormhole switching, a blocked packet might spread out over the bu ers of several consecutive switches. Thus, strictly speaking, there are

also direct dependencies between non-adjacent channels. These are normally not included in the channel dependency graph, however, as they do not provide new information for reasoning about deadlock.

Figure 2.4(a) shows an interconnection network with six combined nodes labelled *u* through *z* connected in a ring topology. The communication channels are unidirectional, packets are routed in the clockwise direction, and VCs are not supported. The corresponding channel dependency graph is shown in Figure 2.4(b), where each communication channel is labelled according to its source and sink nodes. Deadlock-freedom cannot be guaranteed for this interconnection network. Let us consider a situation where all the bu ers of the network are full. Assume that the first packet in the input bu er of node u, v, w, x, y, and z is destined for node v, w, x, y, z, and u, respectively. In this set of packets none of the packets can advance before one of the other packets advances. Thus, this example illustrates an established deadlock.

We will discuss various aspects of routing in Section 2.5. For now, we briefly mention two definitions. First, for *deterministic routing*, as opposed to *adaptive routing*, only one path is provided from a source to destination processing node. Second, a routing function is *connected* if it provides at least one path between every pair of source and destination processing nodes.

For deterministic routing, [55] showed that a necessary and su cient condition for deadlock-freedom is that the channel dependency graph is free of cycles. This also follows as a corollary of the theorems provided for adaptive routing in [64] for wormhole switched networks and in [65] for virtual cut-through and store-and-forward switched networks. These theorems state that an adaptive routing function R is deadlock-free given that a connected routing subfunction $R^{'} \subseteq R$ exists that does not have cycles in its *extended channel dependency graph* [64]. For virtual cut-through and store-and-forward switching, an extended channel dependency graph may include both direct dependencies and direct cross dependencies. For a wormhole switched network, where a blocked packet can occupy several bu ers, indirect dependencies and indirect cross dependencies may be present in addition to direct dependencies and direct cross dependencies. Let us refer to the channels $c_e \in R^{'}$ as escape channels, and the channels $c_{ne} \in R^{''}$, where $R^{''} = R - R^{'}$, as non-escape channels. Then, an indirect dependency [63] from one escape channel to another occurs when a packet holds one or more non-escape channels in between two escape channels. Indirect dependencies can be avoided by prohibiting subsequent use of a non-escape channel for a packet that has already used an escape channel. Cross dependencies [64] occur when the set of escape channels overlaps with the set of non-escape channels. According to [57], such overlap is rare in real networks, however. In summary, the theorems provided in [64, 65] express that the channel dependency graph for an adaptive routing function may contain cycles as long as escape channels, which are not part of any cycle, enable dissolution of potential deadlocks.

Two di erent groups of mechanisms are used to prevent a deadlock from ruining the performance of an interconnection network. *Deadlock avoidance* mechanisms, which are most commonly used, ensure that deadlocks cannot occur, whereas *deadlock recovery* mechanisms detect and resolve a deadlock as it occurs. (In addition, a third group of mechanisms – referred to as *deadlock prevention* – allocates the resources to be used in advance. This may significantly limit the utilization of a network's resources [68].)

In order to break cycles of channel dependencies, most deadlock avoidance mechanisms either use VCs or restrict the set of physical paths included in a routing function. Each of these approaches has possible drawbacks, and for both approaches load imbalance may be an issue. For the former

approach the number of VCs required in each switch may be prohibitive. The latter approach causes a reduced path diversity and may thus also cause reduced fault tolerance. Furthermore, for simple routing strategies, restrictions on physical paths alone may not be sufficient to avoid deadlock for topologies that include cycles. (This is further explained in Section 2.5.1 for torus topologies.) These drawbacks may be alleviated by using combined mechanisms that apply both VCs and a restricted routing function in order to achieve deadlock-freedom. Section 2.8.1 presents a number of existing topology agnostic routing algorithms, including their deadlock avoidance mechanisms. In addition, Section 2.5.1 provides examples of routing algorithms for meshes and tori, the topologies used in the performance evaluation experiments of this thesis.

Bubble flow control [35] takes a different approach to avoid deadlocks in k-ary n-cube topologies. It assumes that virtual cut-through switching is used and that each buffer has space for at least two packets. Effectively, k-ary n-cube topologies consist of a set of rings. Bubble flow control ensures that a packet is inserted into a ring only if, after the insertion, the ring will have buffer space for at least one more packet. This ensures that at any time the ring has at least one free buffer – a "bubble". Thus, at least one of the packets in the ring is always allowed to advance, which means that deadlock cannot occur. (Starvation is a challenge that should be addressed, however.)

Another approach is proposed in [72] for source routing networks. In order to avoid cycles of channel dependencies, the path of a packet may be split such that the packet is ejected from – and subsequently re-injected into – the network at an intermediate switch. The so-called in-transit buffers, where ejected packets are temporarily stored, are located in processing nodes. Thus, all switches must have one or more processing nodes attached.

Deadlock recovery mechanisms (such as [13, 116]) are only useful if deadlocks rarely occur. Moreover, running applications must be able to endure a deadlock. A deadlock can be detected by searching for a cycle of blocked packets. Simpler, but less precise, methods based on timeouts have also been proposed. A detected deadlock must be resolved, for instance by discarding one or more packets. This thesis assumes that deadlock must be avoided, and does not focus on deadlock recovery.

2.4 Fault models

A variety of component faults can occur in an interconnection network, and the faults can have a number of different reasons. In order to simplify the reasoning about faults in an interconnection network, a few categories or models of faults have been introduced [57, 68].

One of the categories is referred to as *transient faults*, and includes temporary faults such as bit errors on a physical channel. A metric commonly used to quantify the frequency of such errors is the bit-error rate. Bit errors can be detected and corrected, e.g. by cyclic redundancy checking [175] and forward error correction [94] schemes, respectively.

Another category of faults is the *permanent faults*, which, as the name implies, includes faults that are not temporary. An example of a permanent fault is a switch with a lasting error in the logic of one of its central units, such as the unit responsible for routing and arbitration illustrated in Figure 2.2. The frequency of permanent faults is commonly quantified by the metric mean-time between failures.

Byzantine faults [125] represent a category of faults which is normally difficult to handle. A Byzantine fault implies that the failing component does not halt or shut down. On the contrary, in

an erroneous and arbitrary way it proceeds operation.

This thesis addresses challenges related to routing and reconfiguration after the occurrence of faults. In particular, we focus on deadlock avoidance and performance issues. Our interest in faults is restricted to permanent failures of the physical communication channels and switches of an interconnection network. Faults may also occur in other components of a computing system, however, such as in processing nodes, power supplies, clocks and fans. We assume that transient faults, such as bit errors on a physical communication channel, are corrected locally and do not focus on this type of faults. If a specific network component produces transient faults more frequently than acceptable, the component may be shut down and the fault can then be considered permanent [57]. Our attention is restricted to permanent faults that can be described as *fail-stop* [203]. Basically, a fail-stop fault implies that some internal or external supervisory module detects the failure of a network component and subsequently informs neighbouring components. Furthermore, we assume the presence of robust fault detection systems which are able to detect a Byzantine fault, shut down the faulty network component, inform the neighbours, and thereby reduce the Byzantine fault to a fail-stop. In this thesis, we focus on both *static* and *dynamic* faults. The former category includes faults that are present at the start-up of a computing system, whereas the latter category includes faults that occur while a system is in operation. In general, handling dynamic faults is more challenging than handling static faults.

2.5 Routing

A routing algorithm is responsible for providing the path that a packet is to take from its source to destination processing node. Existing routing algorithms vary in a number of di erent respects – such as in the following: where the routing decision is taken; number of destinations for each packet; input parameters; deadlock avoidance mechanism; implementation (by programmable routing tables or hardware logic circuits); whether several alternative paths are provided (and how one of them is selected); whether a specific network topology is assumed; whether the paths provided correspond to the shortest paths in the physical topology; and whether in-order delivery of packets, load balancing and fault tolerance are supported.

Two separate units, a *routing function* and a *selection function*, constitute the routing algorithm. A routing function must be *connected*, that is, it must provide at least one path between every pair of source and destination processing nodes in the network. Otherwise, the routing function is *disconnected*, which means that some of the system's processing nodes cannot communicate. As input some routing functions accept a pair of nodes (either the source and destination processing nodes, or the current switch and the destination processing node). Others accept the current input channel of a switch and the destination processing node as input. As output some routing functions return a set of alternative paths that the packet could follow from its source to destination processing node, whereas others return a set of alternative output channels from the current switch. The selection function chooses one of the alternative paths or output channels, and thus decides the exact route of the packet.

This thesis is only concerned with unicast routing, where a packet is transmitted from one source to one destination, and not with multicast routing, where a packet can be transmitted from one source to several destinations.

A routing decision may be taken either once at the source processing node or incrementally

at each intermediate switch. Using the former approach, referred to as *source routing*, the entire path of a packet is set at the source processing node and included in the header of the packet. Using the latter approach, referred to as *distributed routing*, a packet merely carries the address of its destination processing node. Upon arrival at an input channel of an intermediate switch, this switch decides the output channel on which to forward the packet. For the switch illustrated in Figure 2.2, this decision would be taken by the routing and arbitration module.

Source and distributed routing have their own advantages and disadvantages. In general, the total amount of routing table information is higher in a source routing system than in a distributed routing system. In addition, with source routing, the overhead in each packet is normally larger as the entire path is included in the packet header. On the other hand, a source routing switch may be simpler and faster than a distributing routing switch. A packet may be forwarded more quickly by a source routing switch as the output channel is included in the packet header. A distributed routing switch must perform a lookup in its routing table, and possibly select one of several alternative output channels. As the path of a source routed packet cannot be changed at an intermediate switch, source routing is in some respects less flexible than distributed routing is. In other respects, source routing provides more flexibility than distributed routing does: Assume that two source processing nodes, s_0 and s_1, transmit packets towards a common destination processing node, d, and that these packets arrive on the same input channel of an intermediate switch i. Then, with distributed routing, the packets from s_0 and s_1 are bound to share the same set of paths from i to d. With source routing, on the other hand, the set of paths from s_0 to d is independent of the set of paths from s_1 to d. Dolphin Express [122] and InfiniBand [105] are examples of source and distributed routing interconnection network technologies, respectively.

Routing tables, included in either source processing nodes or intermediate switches, are flexible since they are programmable. At the cost of flexibility, some simple routing algorithms for regular network topologies can be implemented in hardware. Such an approach can provide fast routing without requiring storage space for routing information, and is often referred to as *algorithmic routing*. Dimension-Order Routing (DOR) [231] for mesh and torus topologies is an example of a routing algorithm that can be implemented in hardware. At the expense of some additional logic per switch port, Logic-Based Distributed Routing (LBDR) [73] enables hardware implementations of more flexible routing algorithms than DOR. LBDR targets on-chip networks that conform to a defined set of regular and semi-irregular mesh-based topologies. Flexible DOR (FDOR) [219] is another algorithmic routing method designed for on-chip networks. When compared to DOR, FDOR only requires a single additional configuration bit in each switch. Thus, FDOR has lower implementation costs, but in some respects also less flexibility, than LBDR has. Recently, [209] added multicast capability to FDOR as well as an increased flexibility with respect to the supported shapes of mesh-based topologies. These extensions entail a higher implementation cost, however.

A routing algorithm may be either *deterministic*, *oblivious* or *adaptive*. A deterministic algorithm (see e.g. [231]) provides only one path from a source to destination processing node. That is, the routing function returns only one path or output channel, and a selection function is not needed. The set of deterministic algorithms is a subset of the set of oblivious algorithms. Oblivious algorithms (see e.g. [160]) may support more than one path from a source to a destination, and for each packet one of the alternative paths is selected without consideration of network state. The selection function may for instance randomly choose one path/channel from the set of paths/channels returned by the routing function. Like oblivious algorithms, adaptive algorithms (see e.g. [53])

provide more than one possible path from a source to a destination. As opposed to oblivious algorithms, however, adaptive algorithms take network state into account when selecting one of the alternative paths. For instance are the alternative output channels' queue lengths – which represent the output channels' load – commonly used in the routing decision [57].

For some applications it is important that packets are delivered at the destination in the same order as they were sent from the source. The packets that must be delivered in order constitute a flow, and a flow identifier can be included in each packet. A deterministic routing algorithm delivers all packets in order as all packets take the same path from a source to a destination. Oblivious routing algorithms can ensure in-order delivery of packets that belong to the same flow by letting the selection function base its choice of path on the flow identifier instead of randomly choosing a path [57]. Adaptive routing algorithms which base the selection of the paths of individual packets on network state do not support in-order delivery of packets.

A routing algorithm may be shortest path, minimal, or non-minimal. This thesis differentiates between shortest path routing and minimal routing, although in the literature the two terms are sometimes used as synonyms. A shortest path routing algorithm provides a set of paths which are the shortest possible according to the physical topology. A minimal routing algorithm, on the other hand, provides a set of paths which are the shortest possible according to some routing restriction. An example of such a routing restriction is a prohibited down-link to up-link *turn* according to the Up*/Down* routing algorithm [204] described in Section 2.8.1. (A "turn" denotes a traversal from one communication channel to another, and for a switch with both input and output buffering this means a traversal from one of its input buffers to one of its output buffers.) A non-minimal routing algorithm is not restricted to providing shortest or minimal paths but may also provide longer paths. The motivation for non-minimal routing may be to better load balancing or to bypass congested communication channels.

Deadlock avoidance is an important concern for most of the routing algorithms proposed for interconnection networks. Section 2.3 explained that, in order to avoid deadlocks, routing restrictions are commonly applied, sometimes in combination with network resources such as VCs. Sections 2.5.1 and 2.8.1 present some of the most important routing algorithms proposed in the literature and their mechanisms for achieving deadlock-freedom.

For routing algorithms that allow *misrouting* (see e.g. [53]), another important issue is the avoidance of *livelock*. Misrouting is a routing decision that brings a packet further away from its destination processing node, for instance in order to stay away from a congested communication channel. A packet is livelocked if it moves around in the network indefinitely without reaching its destination. Livelock is not an issue in source routing systems as the entire path of a packet is decided at the source processing node. For a distributed routing system, a possible approach to livelock avoidance is to limit the number of times a packet can be misrouted in intermediate switches. This thesis is not concerned with the livelock issue, and the routing algorithms in focus do not use misrouting.

Load balancing is important for the performance of an interconnection network, as an imbalance may cause network congestion, increased packet latency and decreased system throughput. As only a single path is supported from a source to a destination, deterministic routing algorithms do not take advantage of any path diversity that the topology might offer in order to balance traffic load throughout the network. Thus, for non-uniform traffic patterns, deterministic routing can cause severe performance degradation. Oblivious routing algorithms, on the other hand, may achieve

load balancing by spreading tra c over the available paths, possibly according to some probability distribution. Examples of randomized oblivious routing algorithms that address load balancing include [215, 243]. Adaptive routing algorithms do not necessarily achieve better load balancing than oblivious routing algorithms do. Their path selection, which is often solely based on local knowledge of network state, may not favour global load balancing.

During the operation of an interconnection network, faults may occur in its communication channels and switches. Such faults may disconnect the routing function and disable communication between some of the processing nodes. As only one possible path is provided from a source to a destination, a deterministic routing algorithm, such as [231], is not fault-tolerant in its basic version. Oblivious or adaptive algorithms, on the other hand, may select an alternative path which is not a ected by a fault (if such a path exists).

Some fault-tolerant routing algorithms, such as [84, 95], are only able to handle faults statically – that is, they can handle faults that are present at the start-up of the system. Other algorithms, such as [33, 162, 258], can handle faults dynamically, which means that they support continued operation of the system without requiring a restart.

A semi-irregular network topology may be the result of faults in the switches and communication channels of an originally regular network topology. Such faults could occur while a system is in operation, or they could be present at the start-up of the system. For instance, for an on-chip network, which commonly uses strictly orthogonal topologies such as two-dimensional meshes, a semi-irregular topology could result from faults introduced during the production process. Many routing algorithms assume a specific topology or group of topologies. These algorithms are referred to as *topology specific* routing algorithms, and some of them are described in Section 2.5.1. For semi-irregular topologies that result from faults in regular topologies, such algorithms may not be able to calculate a connected routing function. A number of extensions have been proposed in order to introduce fault tolerance into topology specific routing algorithms, however. Such fault-tolerant algorithms (see e.g. [81, 162]) have the ability to route around a faulty area, and thereby maintain a connected routing function after the failure of some of the communication channels or switches of a network.

When routing around faulty regions, care must be taken in order to ensure deadlock-freedom, as new channel dependencies may be introduced. Deadlock may result if these new channel dependencies cause the formation of a cycle in the channel dependency graph of the routing function. A number of solutions, such as [33, 37, 39] depend on VCs for deadlock avoidance. Most of the fault-tolerant routing algorithms designed for a specific topology are limited with respect to the combinations of component failures that are supported. Requirements for successful operation may include the shape of faulty regions – such as for [33] (which handles convex faults in mesh topologies) and for [37,39] (which in addition handle some types of non-convex faults). In order to represent one of the supported fault-combinations, healthy switches and communication channels could be turned o [37].

Topology agnostic routing algorithms constitute a category of routing algorithms that do not assume a specific interconnection network topology. Such a routing algorithm can be used for any network topology, and is thus an obvious choice for a network with an irregular topology. Perhaps more importantly, topology agnostic routing algorithms also support fault tolerance in networks with regular topologies. Topology agnosticism is further discussed in Section 2.8.

A vast number of routing algorithms are available in the literature, and Section 2.5.1 provides

an overview of a few important algorithms for mesh and torus topologies (the topologies used in the performance evaluation experiments of this thesis). Section 2.8.1 presents some of the most important topology agnostic routing algorithms. In general, our focus is on the algorithms' approach to deadlock avoidance.

2.5.1 Selected algorithms for meshes and tori

Dimension-Order Routing (DOR) [231] is used in mesh and torus (k-ary n-cube) topologies. Recall from Section 2.1 that these topologies consist of n dimensions with k switches in each dimension (where k can be di erent for each dimension). By routing a packet in one dimension at a time, DOR ensures that a change of dimension cannot result in a cycle in the channel dependency graph. In mesh topologies this is su cient to avoid deadlock, as routing within one dimension cannot cause a cycle of channel dependencies. For a torus topology, on the other hand, each dimension constitutes a ring, and thus a cycle. Therefore, the use of DOR in torus topologies depends on an additional mechanism, such as VCs (see [55]) or Bubble flow control (see [35]), for the avoidance of deadlock. In a two-dimensional mesh or torus, the two dimensions are often called X and Y, and DOR is often referred to as either XY-routing or YX-routing, depending on which of the turns are allowed. For XY-routing, the turns from the X-dimension to Y-dimension are allowed, whereas the turns from the Y-dimension to X-dimension are prohibited.

In strictly orthogonal topologies, such as meshes, at least four turns are needed in order to close a cycle between two dimensions. We have seen that DOR prohibits two of these turns. The Turn Model [80] – which refers to the four routing directions of a two-dimensional mesh as *west*, *south*, *east* and *north*, and also defines a *positive* direction (east or north) and *negative* direction (west or south) within each dimension – supports path diversity and more flexible routing by only prohibiting one of the four turns. This forms the basis of algorithms such as West-First (where any turn from south, east or north towards the west direction is prohibited), North-Last (where any turn from the north direction towards west, south or east is prohibited), and Negative-First (where any turn from a positive to a negative direction is prohibited).

A number of fault-tolerant routing algorithms are based on DOR or the Turn Model. Some examples are found in [33,81,162,216]. Another example is the solution proposed in [95], where a number of healthy processing nodes in a mesh topology are sacrificed so that the remaining healthy processing nodes can communicate. Healthy switches are kept operative, however. Two rounds of DOR routing might be needed to deliver a packet to its destination, and a separate VC is required for each round.

We will not consider the details of other topology specific routing algorithms, as this group of algorithms is outside the main focus of attention of this thesis. Numerous other routing algorithms have been proposed for mesh and torus topologies, however (see e.g. [42, 134, 160, 214]).

2.6 Reconfiguration

In some cases the routing function of an interconnection network has to be replaced, and the change-over from one routing function to another is referred to as *reconfiguration*. In this thesis, reconfiguration is applied for two di erent purposes. First, it is used as an important fault tolerance mechanism for interconnection networks. Second, as part of our work on processor allocation,

reconfiguration is used in order to ensure isolation of tra c between di erent parts of an interconnection network (such that tra c that belongs to one job cannot interfere with tra c that belongs to other jobs). The latter application of reconfiguration is discussed in Chapter 6, and will not be further considered in this section. (Examples of other areas of application for reconfiguration are found in [155] and [77]. In [155] reconfiguration is used by a power saving strategy for computer clusters, whereas in [77] reconfigurable routing algorithms are utilized for load balancing purposes in on-chip environments.)

A change in the topology of an interconnection network could be a result of faults in one or more of the network's components. The risk of a fault in any of the switches or communication channels of a network increases as a computing system grows larger. On the other hand, a topology change could also be a result of planned system updates, where network components are removed or added.

Recall from Section 2.5 that a routing function is disconnected if, for any pair of processing nodes, packets sent from one of the nodes cannot reach the other node. The routing function of an interconnection network may be disconnected as a consequence of a topology change, even if the physical topology is still connected. In case network components have been removed (either due to a fault or a planned update), a loss of packets must be expected as some packets will attempt to cross missing network components. A number of network management functions are involved in order to restore a connected routing function. First, the change of topology must be detected and the resulting topology discovered. Then, a set of new paths, which constitutes a connected routing function, must be computed. Finally, the existing (disconnected) routing function must be replaced by the new routing function. An overview of the entire process is given in [25, 186]. The focus of this thesis is on the final step of the process – the reconfiguration. In the following we will refer to the routing function to be replaced as R_{old} and to its successor as R_{new}.

A main challenge related to reconfiguration is deadlock avoidance. Even though both R_{old} and R_{new} alone are deadlock-free, deadlock might occur during the transitional phase [67]. In a source routing system (where each packet is routed from its source to destination entirely according to one of the routing functions) cyclic channel dependencies and deadlock may form if packets that belong to R_{old} take turns that are not allowed in R_{new} or vice versa. For a distributed routing system, the problem of avoiding cyclic channel dependencies between packets that follow two di erent routing functions is principally the same as for a source routing system. Additional challenges may be experienced for a distributed routing system, however. As the switches of an interconnection network operate asynchronously, some of the switches may have taken R_{new} into use while other switches still use R_{old}. Thus, underway from its source to destination, a packet may be routed according to di erent routing functions in di erent switches. Assume that both R_{old} and R_{new} accept as input parameters the input channel of the current switch and the destination processing node. Also assume that a packet with destination d arrives at input channel c of an intermediate switch s. Then, if the routing function in use in s does not expect packets destined for d to arrive at c, the packet becomes unroutable. Furthermore, for distributed routing systems, the risk of packets looping is a challenge that must be addressed.

The various reconfiguration strategies proposed for interconnection networks include di erent mechanisms for deadlock avoidance, and are commonly divided into two main categories – *static* and *dynamic* reconfiguration. Most static reconfiguration methods, such as [31, 190, 204, 233], do not allow application tra c into the network during the reconfiguration. For this category

of methods, all packets routed according to R_{old} are drained from the network before any packet routed according to R_{new} is allowed into the network. Thus, the risk of deadlock is eliminated, as packets routed according to only one of the deadlock-free routing functions are present in the network at a time. With this approach, the network service becomes unavailable during the change of routing function, and such a high cost of deadlock-freedom could be unacceptable for some applications and unfortunate for others. Dynamic reconfiguration methods, such as [36, 140, 179, 185], aim at improving the network service availability during the change of routing function. This category of methods allows application tra c into the network during the reconfiguration, and therefore requires sophisticated deadlock avoidance mechanisms. Some of these deadlock avoidance mechanisms cause a temporary degradation of performance, however. When such mechanisms are applied, applications must tolerate issues such as a temporary decrease of throughput and increase of latency. An extensive theory for deadlock-free dynamic reconfiguration is introduced in [67]. Based on this theory, a methodology is presented in [141] for developing procedures for deadlock-free reconfiguration.

The necessity of a reconfiguration is indisputable if a routing function has been disconnected due to a change of the topology. In other cases, a connected routing function could be sustained by a fault-tolerant routing algorithm (as explained in Section 2.5). Fault-tolerant routing algorithms include mechanisms that allow packets to be routed around a faulty area. In some cases, however, disabling non-faulty network components may become necessary (see e.g. [37]). Permanent performance penalties could result from a sub-optimal routing function provided by a fault-tolerant routing algorithm, or from its deactivation of healthy network components. In such cases, a reconfiguration – which causes only temporary performance penalties – could be a better option. Reconfiguration can also be combined with fault-tolerant routing in order to reduce, or perhaps even avoid, a temporary loss of packets (due to missing network components) while an optimal routing function is being restored – see e.g. FRoots [236] which is a topology agnostic approach.

A number of reconfiguration methods are available in the literature, and some of the most important are the approaches discussed in Section 2.8.2.

2.7 Processor allocation and virtualization

In this thesis, a *parallel job* or *parallel application* denotes a program for which separate tasks are executed concurrently on a number of di erent processing nodes in a multiprocessor system – such as a computer cluster, data center or multicore chip. A parallel job typically executes on a subset of the processing nodes available in a multiprocessor system, and is more or less communication and computation intensive.

Two main categories of multiprocessor systems exist – shared memory and message-passing. A shared memory multiprocessor includes a single address space, and a set of processing nodes that cooperate in executing a parallel job communicates through operation on shared variables. This requires synchronized access to the shared variables, which can be ensured through a mechanism for mutual exclusion, such as monitors [96]. In a message-passing multiprocessor, on the other hand, each processing node has a separate address space, and a set of processing nodes that cooperate in executing a parallel job communicates (and implicitly synchronizes) by sending and receiving messages.

Amdahl's law [11] is commonly used to estimate the maximum speed-up that can be expected

from parallelization of a certain fraction of a sequential program onto a given number of processing nodes [171, 250]. Application programming interfaces such as Message Passing Interface (MPI) [158] and OpenMP [164] have been proposed in order to support development of parallel applications. For several reasons, development of a parallel application is more challenging than development of a sequential application [171]. A programmer must identify the parts of the application to be parallelized. E cient parallelization also involves considerations of such issues as the number of processing nodes to apply and the balance between computation and communication. Furthermore, synchronization of the access to shared variables, or exchange of messages between di erent processing nodes, must be explicitly handled by the programmer. Many research studies, such as [21, 30, 93], have attempted to develop compilers with support for automatic parallelization of sequential programs. Satisfactory solutions seem hard to obtain, however.

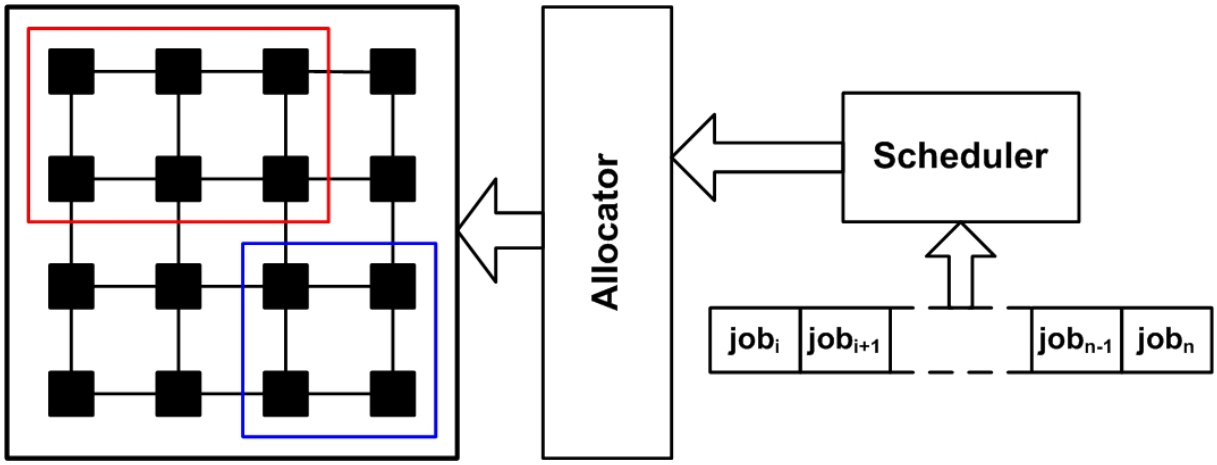


Figure 2.5: Resource management sub-system.

Processor allocation is the assignment of a set of nodes[4] with available capacity to execute a parallel job. Such a set of nodes will also be referred to as a *partition*. In a multiprocessor system some resource management sub-system is responsible for the allocation of nodes. Figure 2.5 illustrates such a sub-system with 16 nodes interconnected in a 4×4 mesh topology. Two parallel jobs are currently running (one on a 3×2 sub-mesh and the other on a 2×2 sub-mesh), while a queue holds a number of waiting jobs. Each of the waiting jobs has requirements on such aspects as number of nodes and running time. From the waiting queue a scheduler selects the next parallel job to be started. An allocator then selects a set of nodes to execute the job. In the literature, the terms processor *allocation* and *scheduling* are sometimes used interchangeably. In this thesis, however, the allocator and scheduler are consistently assumed to be responsible for separate tasks (in accordance with the explanation above). Furthermore, our focus is on allocation, not on scheduling. (A status report on the research area of scheduling was presented in [69].)

Another research area related to processor allocation that is not in the focus of this thesis is *task mapping*. A parallel job is often represented as a graph of separate tasks, where the edges of the graph represent communication between tasks. Task mapping involves assignment of the individual tasks of a job to the individual nodes of the partition allocated to the job. (In cases where an entire system is allocated to a job, the entire system constitutes a single partition.) Task

[4]For simplicity, we assume that a *node* refers to a combined node which contains both processing and switching elements.

mapping corresponds to the mapping problem discussed in [32]. Examples of studies that focus on task mapping include [26, 27]. We did not include a task mapper in Figure 2.5.

Processor allocation is a form of virtualization. Virtualization is a general term that denotes the abstraction of logical entities from physical entities. Roughly, virtualization can be used both to have one physical unit appear as several logical units, and to aggregate several physical units into one logical unit. In this thesis, we are mainly interested in the latter approach. We have adopted the space-sharing model commonly assumed in the literature on processor allocation. Space-sharing implies that a set of nodes is exclusively assigned to a single parallel job which runs uninterruptedly until completion. (Time-sharing, on the other hand, allows more than one job to be assigned to a particular node at a time.) According to [127], space-sharing was also applied in the Computational Plant (Cplant) [49] supercomputer at Sandia National Laboratories [201]. In a space-sharing multiprocessor system, processor allocation essentially implies aggregation of a set of physical processing nodes into a virtual processor which is assigned to execute a parallel job. Virtualization techniques are used in various parts of computing systems. It is found in processing (examples include some of the processors from Intel and AMD which o er hardware support for virtualization [10, 107]); storage (examples include RAID systems [170]); and communication channels (examples include VCs [54]). Although in recent years it has again received much attention, virtualization is not a new concept. For instance, in [180] – published in 1974 – a set of necessary and su cient conditions was stated for a third generation computer to support a virtual machine monitor[5], a software entity for controlling multiple virtual machines running on a single physical machine. Although virtual machines are important in processor allocation and parallel processing, for instance with respect to issues such as security and migration of jobs, their role is not in focus in this thesis.

A significant number of processor allocation strategies have been proposed for multiprocessor systems. Although hybrid strategies such as [230] have been introduced, the majority of the algorithms fall into one of two categories – they are either contiguous such as [113, 253, 262], or non-contiguous such as [34, 127, 137]. A contiguous algorithm designates a set of adjacent nodes to a parallel job, whereas a non-contiguous algorithm may designate a set of nodes that are not adjacent. There are advantages and disadvantages associated with either of the two categories. External fragmentation[6] is an inherent issue for contiguous algorithms that can be completely avoided by non-contiguous algorithms. It occurs when a su cient number of nodes are available, but the allocation attempt nevertheless fails due to some restriction. For instance, many allocation strategies for mesh topologies require that a region of available nodes constitutes a sub-mesh. Internal fragmentation occurs if more nodes than requested must be allocated to a job. For instance, in [130] the allocated area must be a quadratic sub-mesh for which the side lengths are powers of 2. Both external and internal fragmentation cause a reduced utilization of a system's processing resources.

Modern interconnection network technologies could use VCs to separate tra c. Nevertheless, when using a non-contiguous processor allocation strategy, contention for the capacity of a physical communication channel between messages that belong to concurrent jobs may be unavoidable. Such contention often has unfortunate consequences, both with respect to the performance of individual jobs and with respect to the performance of a computing system. It increases the time spent on communication for a parallel job, and thereby also increases the job's running time. Thus, subsequent jobs must await service for a longer time, and the system's throughput of jobs decreases.

[5] A virtual machine monitor is also referred to as a hypervisor.

[6] In the following, external fragmentation will often be referred to simply as fragmentation.

Some contiguous processor allocation strategies have a characteristic that we refer to as *tra c-containment* (or tra c isolation). For these allocation strategies, the set of nodes to be assigned to a job is selected in accordance with the underlying routing function such that no communication channels are shared between messages that belong to di erent jobs. With tra c-contained processor allocation, each job can be guaranteed a fraction of the interconnection network's capacity, regardless of the properties of concurrent jobs. Thus, if one parallel job introduces severe congestion within the interconnection network, other jobs should not be a ected. The importance of tra c-containment, both with respect to security requirements and avoidance of inter-job congestion, is emphasized in [14, 15]. In studies on processor allocation the concept of tra c-containment is often not discussed explicitly. Nevertheless, many strategies, like those that allocate sub-meshes in meshes (see e.g. [61, 253]), will be tra c-contained whenever the predominant routing algorithm, DOR, is used. (DOR was described in Section 2.5.1.)

The processor allocation strategies referred to above are typically proposed for o -chip environments – that is, not for systems found within integrated circuit chips. In the following paragraphs we will consider issues related to allocation of nodes (cores) in on-chip systems.

In [74], we identified a set of problems to be addressed for a network interconnecting a number of cores in a multicore chip. It was assumed that, in the design of a multicore chip, the main metrics to be optimized are performance, fault tolerance/yield[7] and power consumption/silicon area. As improvements to one of these metrics may have an opposite e ect on another metric, an optimization that comprises all of the metrics represents a challenge. For instance, inclusion of a sophisticated routing algorithm may improve connectivity between the on-chip components (such as cores and caches), and thereby increase the yield as more of the chips with manufacturing defects can be utilized. If the sophisticated routing algorithm requires advanced logic or large routing tables, however, the improvement might come at the cost of increased power consumption and silicon area. We argue in [74] that virtualization is a useful tool for supporting optimization of the three metrics, and that the following issues are the most important ones to address. First, partitioning mechanisms that maximize resource utilization and minimize contention between the tra c of concurrent jobs should be used. Second, coherency domains should be applied in order to enhance cache coherency. Third, in order to increase yield, connectivity should be improved for chips with production faults. Fourth, power consumption should be reduced by turning o idle components. Among these four issues, the first one is the most relevant for this thesis, and we recognize it from the above discussion on processor allocation in o -chip systems. This illustrates that, in many respects, the challenges related to processor allocation are similar for on-chip and o -chip environments.

Meshes and tori are relevant topologies for on-chip systems [56, 124]. Moreover, DOR is a simple routing algorithm that can be implemented in hardware, which makes it suitable for on-chip interconnection networks. Thus, some of the sub-mesh allocating strategies referred to above for o -chip systems could be adopted into on-chip systems. While ensuring tra c-containment when DOR is used, the allocation of sub-meshes will cause fragmentation in on-chip systems – as it does in o -chip systems – and thereby reduce the utilization of available cores. In [74] we discussed how the utilization of cores could be improved if restrictions regarding the shape of partitions were relaxed (this is also a main topic in Chapter 6). In such a case, tra c-containment could be sustained by the use of a more flexible routing algorithm than DOR, and a closer cooperation between the

[7] *Yield* denotes the fraction of a set of manufactured integrated circuit chips that is approved for use.

units responsible for allocation and routing would be required. For most o -chip systems, the implementation of a topology agnostic routing algorithm is straightforward. The most flexible routing algorithms depend on either source or distributed routing. In an on-chip environment, however, a routing algorithm that requires routing tables represents an increased demand for memory space, silicon area and power. Furthermore, some topology agnostic routing algorithms, such as [200, 217], also depend on VCs (and VC-transitions) for deadlock avoidance. For some on-chip systems, the requirements of such algorithms are expected to be unacceptable. In these cases, logic-based routing algorithms such as [73, 192, 209] could be considered. These algorithms support more flexible routing than DOR does, which is advantageous with respect to tra c-contained processor allocation as well as with respect to fault tolerance. In addition, [191] – an extension of [73] – applies a simple mechanism in order to implement tra c isolation between di erent network regions.

Additional challenges for resource management in computer clusters with multiprocessor multicore nodes are addressed in [20]. Issues to consider for such environments include locality, on-chip contention and o -chip contention. In general, finer granularity allocation than supported in traditional systems with fewer levels of processing units is required.

The research community has put relatively little e ort into developing topology agnostic solutions for processor allocation. For mesh and torus topologies, on the other hand, a number of processor allocation strategies can be found in the literature. Thus, as only a few topology agnostic allocation strategies are described in Section 2.8.3, we have included a larger number of allocation strategies for meshes and tori – the topologies used in the performance evaluation experiments of this thesis – in Section 2.7.1. Examples of studies that target processor allocation in another group of regular topologies, fat-trees, are found in [168, 210].

2.7.1 Selected strategies for meshes and tori

Section 2.7 explained that there are two main categories of processor allocation strategies – contiguous and non-contiguous algorithms. Non-contiguous processor allocation in general alleviates the fragmentation problem inherent in contiguous processor allocation. On the other hand, a main challenge for non-contiguous allocation algorithms is contention between the tra c of concurrent jobs. In order to minimize such contention, many non-contiguous allocation strategies attempt to limit the dispersal of the nodes assigned to a parallel job. Examples of non-contiguous allocation strategies for mesh and torus topologies that – using various metrics – aim at maximizing the concentration of allocated nodes include [23, 127, 146]. In [145], a number of metrics were suggested to quantify the dispersal of nodes, and a good correlation was demonstrated between these metrics and tra c contention. For an assigned set of nodes, the metrics consider such issues as the number of nodes and communication channels involved; aggregated distances from a center-node; and average, aggregated and maximum pairwise distance between nodes. In [137], the tradeo between fragmentation (for a strategy that allocates contiguous sub-meshes) and tra c contention (for various non-contiguous allocation strategies) was investigated, and the overall conclusion was in favour of non-contiguous processor allocation. Nevertheless, a main goal of this thesis is to contribute to the development of processor allocation strategies that support tra c-containment. Therefore, our main interest is in contiguous – rather than non-contiguous – processor allocation strategies.

A number of contiguous processor allocation algorithms have been proposed for mesh and torus (k-ary n-cube) topologies. Many of them restrict the allowed partitions to sub-meshes or

sub-cubes. Few of the studies ([251] is an exception) discuss that, when DOR is used and sub-meshes are allocated in a topology that has wraparound channels, the shortest path between two nodes may include intermediate nodes that are not part of the partition. (Tra c-containment is thus not supported.) In such cases, messages that are routed outside a partition risk interference with messages that belong to concurrent jobs. According to [14], this issue may be alleviated by the application of the particular torus topology introduced with the Blue Gene/L supercomputer [29]. This topology has additional communication channels, compared to an ordinary torus, and is referred to as a multi-toroidal topology.

First Fit and Best Fit [262] were proposed for two-dimensional meshes to solve problems related with the Two-dimensional Buddy [130] and Frame Sliding [45] processor allocation strategies. The applicability of Two-dimensional Buddy is restricted to square mesh systems, and partitions are restricted to square sub-meshes, where the lengths of the sides of the squares are powers of 2. If a parallel job requests a number of nodes that are not a power of 2, internal fragmentation follows as a higher number of nodes than required must be allocated. As a result of the sliding of frames in fixed strides through a mesh, Frame Sliding will not always recognize a free sub-mesh even if one is available. First Fit and Best Fit keep track of free and busy nodes. For each scheduled job, they determine which of the free nodes can constitute a bottom left corner of a free sub-mesh of the requested size $a \times b$. First Fit allocates the first free sub-mesh found, whereas Best Fit attempts to reduce external fragmentation by selecting the smallest free region for allocation (and thereby leaving larger free regions for future and possibly larger jobs). Two other allocation schemes that aim to reduce external fragmentation by well considered placements of new sub-meshes in two-dimensional mesh topologies are presented in [3].

In case a scheduled parallel job requests a rectangular region of $a \times b$ nodes, Adaptive Scan [61] may rotate the original request by 90 degrees and also search for a free sub-mesh $b \times a$. Adaptive Scan does not scan through every node in the mesh, and achieves a complete sub-mesh recognition capability as it does not fix the strides.

Flexfold [91] has complete sub-mesh recognition capability, and searches first for a sub-mesh of size $a \times b$ or $b \times a$. It may in addition, after consideration of a possible communication overhead, fold the originally requested sub-mesh, and search for a sub-mesh of size $\frac{a}{2} \times 2b$, $2a \times \frac{b}{2}$, $2b \times \frac{a}{2}$, or $\frac{b}{2} \times 2a$ (if a or b is an odd number some of these alternatives are of no interest).

The All Request Shapes (ARS) strategy [4] addresses the fragmentation issue by – in response to a request for n nodes – allowing allocation of a sub-mesh of any shape that includes n nodes and fits within the two-dimensional mesh topology.

In [43], another processor allocation strategy is proposed for two-dimensional mesh based systems. It places an allocation in a vacant sub-mesh that has a left border towards an occupied region or towards the edge of the mesh. This principle is also used by the Leapfrog method [253] which introduces a more e cient data structure for faster recognition of free sub-meshes in a mesh topology. In [253], analytical models concerning the execution cost of the allocation process and the probabilities of finding free sub-meshes under di erent load levels are presented. The focus of both [43] and [253] is more on allocation e ciency than on the fragmentation issue, and the results presented do not indicate a significant reduction of fragmentation when compared to e.g. Adaptive Scan. The Turning Busy List presented in [22] and the Compacting Free List presented in [1], which allocate sub-meshes in three-dimensional and two-dimensional mesh topologies respectively,

are other examples of strategies that aim to improve allocation e ciency.

In order to improve the performance of processor allocation strategies such as those presented in [45, 61, 253], [83] introduces an improved search order for identification of free sub-meshes, as well as two di erent approaches to job migration.

The main idea of both [261] and [232], which target two-dimensional meshes and three-dimensional tori respectively, is to reduce fragmentation by allocating adjacent partitions to jobs with similar finishing times (in order to make larger contiguous regions of nodes available to subsequent jobs).

The allocation strategies presented in [211] and [227] target mesh topologies and attempt to alleviate fragmentation by allowing partitions that are not sub-mesh shaped. L-shaped partitions are allowed in [211], whereas partitions of any shape are allowed in [227]. Neither [211] nor [227] discusses routing and deadlock, and neither strategy can support tra c-containment when DOR is used. However, [211] seems to presuppose that a set of paths can be identified within a partition. We observe that routing algorithms such as Logic-Based Distributed Routing (LBDR) [73] and Flexible DOR (FDOR) [209] could actually realize tra c-containment within an L-shaped partition. (Section 6.3 explains how LBDR can be used for a b-shaped topology – which corresponds to an L-shaped partition.)

In [181], multi-dimensional sub-meshes are allocated for systems based on three-dimensional tori. The strategy has complete sub-mesh recognition capability; does not restrict the orientation of sub-meshes; and may allocate sub-meshes across wraparound channels. Routing issues are not discussed in [181]. We observe, however, that the use of DOR cannot ensure tra c-containment since the shortest path between two nodes may traverse nodes outside the assigned sub-mesh.

The scan search scheme [44] allocates three-dimensional sub-meshes in torus topologies (where the sub-meshes may be allocated across wraparound channels). It has complete sub-mesh recognition capability, allows flexible orientation of the sub-meshes, but is not tra c-contained with DOR. A particular data structure is used to reduce the three-dimensional information on a torus to two-dimensional information, and this decreases the average allocation time when compared to [181].

The Extended Tree-Collapsing strategy [46] mainly addresses the internal fragmentation issue, and accepts requests for any r-ary m-cube partition in systems based on k-ary n-cube topologies. The scheme is a ected by significant external fragmentation, and cannot ensure tra c-containment with DOR (consider e.g. an r-ary m-cube partition with $r > \frac{k}{2}$).

The main idea of the Isomorphic allocation strategy [113] is to partition a k-ary n-cube in a series of recursive steps such that, after the ith step, the number of $\frac{k}{2^i}$-ary n-cube partitions is 2^n. The strategy is, however, not restricted to k-ary n-cube systems nor to the allocation of n-cubes. Several n-cubes can be merged in order to form a region of nodes of a di erent shape. The strategy cannot always ensure tra c-containment if DOR is used. Consider e.g. a system based on an 8-ary 3-cube topology, where the resource requests have the form $x \times y \times z$ where $1 \leq x, y, z \leq 8$. Then, if either of $x, y, z > \frac{8}{2}$, messages may be routed outside their partition due to the wraparound channels.

The k-ary Partner strategy [251] uses a tree structure to represent sub-cubes, and the tree is searched with an aim to localize free m-dimensional sub-cubes (slices) of a k-ary n-cube topology. The strategy supports tra c-containment with DOR, but does not have complete sub-cube recog-

nition capability. It may also be a ected by internal fragmentation, due to the requirement that the allocated sub-cubes have radix k.

For faulty meshes and tori the contiguous processor allocation strategies described above will, in general, not perform satisfactorily. Therefore, particular allocation strategies have been proposed for faulty topologies – see e.g. [115] that identifies virtual sub-meshes in a faulty mesh topology, and [40] that uses a distributed procedure to acquire intact sub-meshes from a faulty mesh or torus topology.

Traditional processor allocation strategies may encounter scalability issues as the size of systems increases. In [15] the approach taken by the Blue Gene/L supercomputer – which consists of $64 \times 32 \times 32$ nodes interconnected in a multi-toroidal topology – is presented. Scalability is achieved by grouping N nodes into one allocation unit (in [15] N is 512). Such a coarse granularity for allocations implies a risk of internal fragmentation and decreased resource utilization, however, unless the parallel jobs request a multiple of N nodes.

For on-chip environments based on two-dimensional mesh and torus topologies, [263] and [264] recently presented slightly optimized and adapted versions of some traditional sub-mesh allocating algorithms (such as First Fit [262], Adaptive Scan [61] and the schemes referred to as Quick Allocation [256] and Stack-Based Allocation [255, 259]). Moreover, in [265] energy consumption was evaluated for an on-chip allocator that runs some of these adapted processor allocation algorithms.

The allocation strategy proposed in [131] aims to promote heat balance in an on-chip system based on a mesh network topology. Thus, the temperature of the cores is taken into account when virtual sub-meshes are allocated.

2.8 Topology agnostic methods

This thesis is concerned with topology agnostic methods for routing and reconfiguration, as well as with the use of such methods as a means to achieve flexible and tra c-contained processor allocation. For interconnection networks with irregular topologies, most methods tailored for specific regular topologies are not applicable. Topology agnostic methods are therefore essential for the operation of such networks. Construction of arbitrary topologies are supported by most interconnection network technologies, such as InfiniBand [105] and Dolphin Express [122]. Nevertheless, regular topologies such as fat-trees and tori are heavily represented at the top of the Top500 list of supercomputers [237], and for on-chip networks two-dimensional meshes and tori are predominant [56, 124]. As topology agnostic methods do not assume a particular topology, they are also usable for any regular topology. In particular, topology agnostic methods play an important part in the provisioning of fault tolerance for regular topologies.

Routing algorithms developed for a specific topology will often not work correctly if the topology is altered by a fault in one or more of the switches or communication channels. A topology agnostic routing algorithm, on the other hand, does not assume a particular topology, and can therefore be used on a topology that are no longer regular due to faulty network components. This quality of topology agnostic routing algorithms is useful both with respect to static and dynamic fault handling.

Static fault tolerance is important for various types of computing systems. Given that the topology of an interconnection network is physically connected, a topology agnostic routing algorithm

is able to provide a connected routing function for the network. Thus, when using topology agnostic routing, static fault tolerance is inherently supported, and any fault present at the start-up of a system is trivially handled. As an example, consider an on-chip network interconnecting a number of cores. For such a system, faults that result from the chip production process are relatively common [171], and may alter an intended two-dimensional mesh or torus topology. The use of a topology agnostic routing algorithm can maximize connectivity and thereby possibly increase the number of useful cores on a chip. This could improve the yield of the production process, given that the overhead of a topology agnostic routing algorithm is acceptable (routing tables are typically required). Static fault handling by using topology agnostic routing may also be advantageous for computing systems that tolerate a restart in case a fault occurs in a network component.

Topology agnostic routing algorithms are also particularly useful for dynamic fault handling. In some cases, a connected routing function can not be sustained after the occurrence of a fault. Then, a new routing function is needed and a reconfiguration must be performed. A topology specific routing algorithm might not be able to provide a new connected routing function, or it might need to impose rigid restrictions, such as disabling non-faulty network components (see e.g. [37]). A topology agnostic routing algorithm, on the other hand, can provide a new routing function for any physically connected topology, including one that results from faults (regardless of the topology or routing function in use before the fault occurred). Reconfiguration processes are commonly motivated by topology changes in general and component faults in particular. Thus, reconfiguration strategies are typically topology agnostic. Both the theoretical framework presented in [67] and the methodology presented in [141] for development of deadlock-free dynamic reconfiguration strategies support topology agnosticism.

Reconfiguration might not be necessary if a connected routing function can be supported after the fault of a network component. An approach to fault-tolerant routing for InfiniBand is proposed in [154], where the deadlock-free and topology agnostic Transition-Oriented Routing [200] algorithm is used to compute a set of disjoint paths for each pair of source and destination processing nodes. If, during the operation of a system, a path is disrupted by a fault, an una ected path can be applied. A similar principle for fault-tolerant routing could be utilized by other interconnection network technologies, provided that migration of paths is supported. In addition, an alternative topology agnostic routing algorithm could be applied.

Much research e ort has been made to develop solutions for topology agnostic routing and reconfiguration, and some of the strategies available in the literature are described in Sections 2.8.1 and 2.8.2. Our contributions to these research areas are presented in Chapters 4 and 5.

Most of the processor allocation strategies found in the literature are developed for a particular network topology, and some of the strategies for meshes and tori were described in Section 2.7.1. Fragmentation – which reduces the utilization of the available computing resources of a system – is an inherent problem in contiguous processor allocation. Nevertheless, a topology agnostic processor allocation strategy could alleviate the problem of fragmentation by allowing increased flexibility with respect to the shape of contiguous partitions. (Recall that many of the traditional processor allocation strategies for mesh and torus topologies only support allocation of sub-mesh shaped partitions – see e.g. [253, 255].) Section 2.7.1 also described special strategies needed for processor allocation in systems with faulty mesh and torus topologies [40, 115]. A topology agnostic approach to processor allocation also has advantages with respect to fault tolerance, as no particular considerations are needed for a topology altered by some fault. Topology agnostic processor allo-

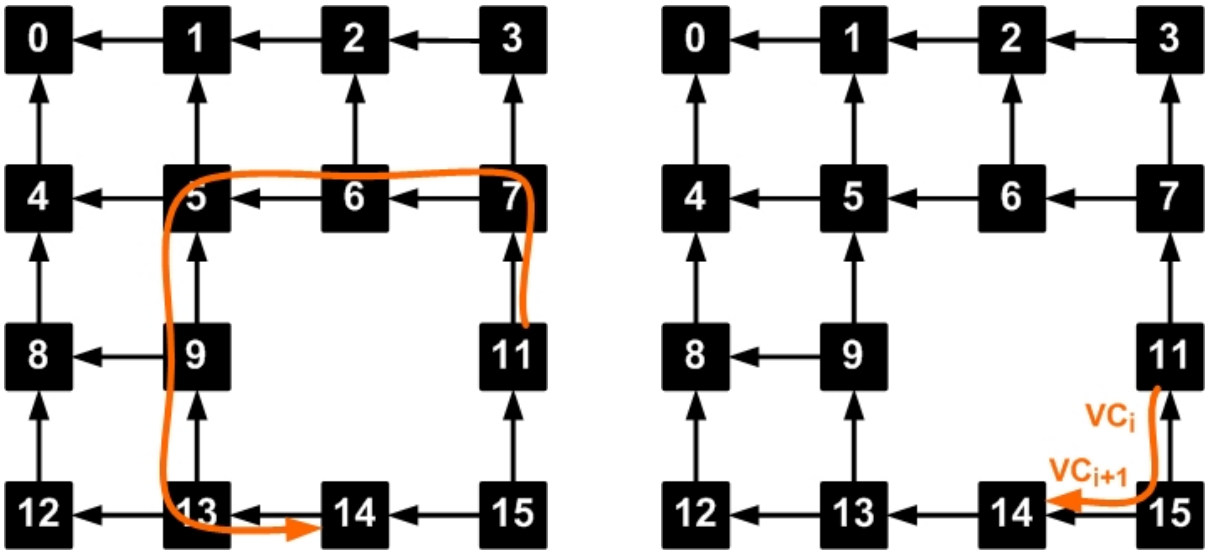
cation has not attracted much attention from the research community, however. Thus, only a few such strategies (some of them non-contiguous) are included in Section 2.8.3. Our contributions to the area of processor allocation are presented in Chapter 6, where both topology agnostic routing and reconfiguration are utilized in order to ensure containment of tra c within arbitrarily shaped partitions.

2.8.1 Selected routing algorithms

A number of topology agnostic routing algorithms avoid deadlock by prohibiting a set of turns in the interconnection network (and thereby breaking cycles of channel dependencies). The Up*/Down* algorithm [204] assigns up and down directions (called up-links and down-links, respectively) to all the communication channels of a network in order to form a directed acyclic graph (DAG). Communication channels and switches are represented in the DAG by edges and vertexes, respectively. The root node of the DAG is the one vertex that has only incoming up-links. The DAG can have one or more leaf nodes – vertexes with only outgoing up-links. Figure 2.6(a) illustrates an Up*/Down* DAG in a 4×4 mesh topology with one node (number 10) missing. In this DAG, node 0 is the root and node 15 is the only leaf. According to convention, an arrow indicates the direction of an up-link (and the opposite direction of a down-link). In a correct Up*/Down* DAG, all other nodes can reach the root node by following a sequence of one or more up-links, and the root node can reach all other nodes by following a sequence of one or more down-links. Up*/Down* avoids cycles of channel dependencies by prohibiting a turn from a down-link to an up-link. Thus, any legal route comprises zero or more up-links followed by zero or more down-links. One of the main advantages of Up*/Down*, in addition to being topology agnostic and conceptually simple, is that VCs are not needed for deadlock avoidance. On the other hand, the performance of Up*/Down* is limited by congestion that tends to form in the area around the root node, as the forwarding restrictions (the prohibited down-link to up-link turns) make this region a hotspot. Another, perhaps less critical, issue caused by the forwarding restrictions is that some of the legal paths could be longer than the shortest physical paths. For instance, consider the long path from node 11 to node 14 in Figure 2.6(a). In this case, the shortest path (through node 15) is prohibited as it would imply a down-link to up-link turn. In the original version of Up*/Down* [204], the DAG was based on a tree identified by a breadth-first search. Improved performance was achieved in [198] where the use of a depth-first search resulted in fewer forwarding restrictions. Moreover, new heuristics for calculating the DAG were introduced in [196], e.g. for selecting the best root node. These results were combined in [199].

Several other routing algorithms have emerged that also base deadlock avoidance on turn prohibition, while attempting to achieve improved performance by addressing the weaknesses of Up*/Down*. Belonging to this category of routing algorithms are Smart Routing [41], Flexible Routing (FX) [197], Multiple Roots (MRoots) [71, 143, 144], Left-Up First Turn (L-Turn) [117], Tree Based Turn Prohibition (TBTP) [174], Simple Cycle-Breaking (SCB) [129] and Segment-Based Routing [151].

Smart Routing aims at balancing tra c better than Up*/Down* does. It computes a channel dependency graph, and uses a greedy approach to break each individual cycle until an acyclic channel dependency graph is obtained. The assignment of prohibited turns depends on heuristic cost functions that attempt to minimize the average length of paths as well as maximize the dispersion



(a) Up*/Down*. (b) TOR.

Figure 2.6: Up*/Down* and TOR routing from node *11* to node *14* in a 4×4 mesh network with one node missing. (An arrow indicates the direction of an up-link.)

of the disallowed turns.

By breaking a cycle of channel dependencies in di erent locations for the clockwise and counterclockwise directions of the cycle, FX achieves better load balancing than Up*/Down* does.

We will refer to the strategy that was independently suggested in [143] and [71] (and later included in [144]) as MRoots. MRoots calculates a separate DAG for each virtual layer. (The concept of a virtual layer was explained in Section 2.2.2.) By distributing the root nodes of each DAG throughout the topology, MRoots improves load balancing and reduces congestion when compared to Up*/Down*.

L-Turn introduces two new directions, left and right, in addition to up and down, and defines a DAG which includes edges with directions left-up, right-up, left-down and right-down. Deadlock is avoided by prohibiting all turns towards a left-up channel. Moreover, an additional restriction is needed to break cycles of channel dependencies that do not include any left-up channel. When compared to Up*/Down*, L-Turn achieves better load balancing and shorter paths due to a better distribution of prohibited turns. A related routing algorithm called Right-Down Last Turn (R-Turn) was introduced in [112], and both L-Turn and R-Turn were further elaborated on in [111].

For TBTP, which was proposed for Gigabit Ethernet [206], guarantees are given that 50% is the maximum fraction of prohibited turns in any topology. When compared to Up*/Down*, TBTP is reported to disallow around 10% fewer turns. On the other hand, the newer SCB algorithm provides a tighter guarantee (33%) on the maximum fraction of prohibited turns.

Although its main strength is perhaps as a static fault tolerance mechanism for meshes and tori, Segment-Based Routing is usually considered a topology agnostic routing algorithm (see e.g. [75]). It divides the topology into smaller autonomous units, referred to as segments, such that the (bidirectional) turns to prohibit can be appointed independently for each segment.

Like the routing algorithms described above, the strategies recently presented in [135], [260] and [157] base deadlock-freedom on turn prohibition. The routing algorithm presented in [135] targets nano-scale on-chip environments and reduces the storage space required for routing in-

formation when compared to Up*/Down*. The Tree-Turn model presented in [260] assigns six di erent directions to the communication channels of an interconnection network – as opposed to Up*/Down* and L-Turn which assign two and four directions, respectively. According to [260], a routing algorithm based on the Tree-Turn model performs better than Up*/Down* and L-Turn perform. In [157] a methodology is presented for developing a class of routing algorithms referred to as zone-ordered label-based routing algorithms. This class of routing algorithms also includes existing algorithms such as Up*/Down* and L-Turn.

Another approach to deadlock-free topology agnostic routing depends on VCs for avoiding cycles of channel dependencies (in some cases, computation of a spanning tree or DAG is also needed). This group of algorithms includes such approaches as Layered Shortest Path (LASH) [144, 218], Transition-Oriented Routing (TOR) [200], LASH-TOR [217], and Descending Layers [119]. In general, shortest path routing can be supported, and the paths could for instance be computed in accordance with Dijkstra's shortest path algorithm [60].

LASH distributes the paths of an interconnection network over a number of virtual layers, such that none of the layers include any cycle of channel dependencies. This is the basic mechanism for ensuring a deadlock-free routing function. LASH is a main subject of Chapter 4, and a more thorough explanation of the algorithm is given there.

Whereas LASH forwards a packet within the same virtual layer all the way from its source to destination, LASH-TOR allows a transition from a layer l to the next higher layer, $l + 1$, wherever required to avoid a cycle of channel dependencies in l. Thus, compared with LASH, LASH-TOR normally achieves a reduction of the number of virtual layers needed for deadlock avoidance.

TOR, which plays an important role in Chapters 5 and 6, calculates a DAG in the same manner as Up*/Down* does. Its calculation of paths is, however, fundamentally di erent from that of Up*/Down*. TOR calculates a path without regard to the underlying DAG. For TOR, the purpose of the DAG is solely to identify the *breakpoints* – the turns where cycles of channel dependencies must be broken. As with Up*/Down*, the breakpoints are the down-link to up-link turns. TOR prevents deadlock by requiring that when a packet crosses a breakpoint (traverses from a down-link to an up-link) it makes a transition to the next higher VC. Figure 2.6 illustrates the di erences in path calculation between TOR and Up*/Down*, using the route from node 11 to node 14 as an example. For Up*/Down* (Figure 2.6(a)) the shortest path through node 15 is prohibited by a down-link to up-link turn. This turn is allowed for TOR (Figure 2.6(b)), but requires a VC-transition.

Descending Layers is based on a similar principle as TOR. The combination of deadlock avoidance and flexible path calculation is enabled by allowing a packet to transfer from a virtual layer k to the next lower layer $k - 1$ when a restricted turn is crossed. Alternatives to pure Up*/Down* for defining the turn-restrictions are discussed. Variants of Up*/Down* may be used, where di erent turn-restrictions are applied in di erent virtual layers in order to reduce the number of restricted turns. Another possibility is the use of L-Turn.

Many of the routing algorithms above are thoroughly discussed and evaluated in [75]. As opposed to these algorithms, Adaptive Trail [182] and Minimal Adaptive [212] are examples of topology agnostic routing algorithms that exclusively operate in an adaptive mode. The deadlock avoidance mechanism of the former is based on Eulerian trails, whereas the latter relies on escape channels.

2.8.2 Selected reconfiguration methods

As explained in Section 2.6, deadlock is a potential risk when a routing function is to be replaced while a system is in operation. The easiest solution is to stop accepting application tra c into the interconnection network while the routing function is being replaced – an approach taken by static reconfiguration strategies (see e.g. [31, 190, 204, 233]).

This thesis is more focused towards dynamic reconfiguration strategies. In an attempt to provide a better service to running applications, such strategies allow tra c to be injected into the network during a change-over from an existing routing function (R_{old}) to a succeeding routing function (R_{new}). Thus, dynamic reconfiguration strategies in general require sophisticated deadlock avoidance mechanisms, although approaches such as [152] which depends on deadlock recovery have also been proposed.

The strategies referred to as Partial Progressive Reconfiguration (PPR) [36] and Close Graph-based Reconfiguration (CGR) [185,187] are based on the Up*/Down* routing algorithm, and both restore a correct Up*/Down* DAG from a DAG that has been rendered incorrect by a topology change. PPR changes the direction of a subset of the edges of the DAG through a sequence of partial routing table updates, and is only useful in distributed routing systems. CGR restricts the new Up*/Down* DAG such that packets belonging to R_{old} and R_{new} can coexist in the network without causing deadlocks, and thereby supports an una ected network service during a reconfiguration process. Neither PPR nor CGR requires VCs to achieve deadlock-freedom. Both methods depend on relatively complicated procedures to establish R_{new}, however.

Double Scheme [179] avoids deadlock during a reconfiguration process by utilizing two sets of VCs in order to separate packets routed according to R_{old} from packets routed according to R_{new}. Each set of VCs accepts application tra c in turn while the other is being drained and reconfigured. Double Scheme can be used between any pair of routing algorithms. However, in order to avoid deadlock, Double Scheme generally requires a number of available VCs that resemble the sum of the number of VCs demanded by R_{old} and R_{new}.

Overlapping Reconfiguration (OR) [139, 140] is a versatile reconfiguration method that can be used between any pair of routing algorithms. It ensures in-order packet delivery, and does not depend on the availability of VCs. For deadlock avoidance OR depends on special packets, called *tokens*, which prevent ordinary packets that follow di erent routing functions from mixing. Originally, OR could only be applied in distributed routing systems. However, in Chapter 5 we propose a solution that also allows use of OR in source routing systems. OR has been categorized both as a dynamic reconfiguration method (in [140]) and as a static reconfiguration method with overlapping phases (in [139]). For this thesis, the functionality of OR is the important aspect, not its categorization. Nevertheless, according to the definitions of static and dynamic reconfiguration used in Section 2.6, OR falls into the dynamic reconfiguration category. (Section 2.6 emphasized whether or not application tra c is accepted into the network at all times.) A more in-depth description of OR is given in Chapter 5.

Some mechanisms, such as NetRec [19], Agent NetReconf [6, 7], LORE [235], and Skyline [138] attempt to restrict a reconfiguration process to part of the topology. NetRec, Agent NetReconf, and LORE are dynamic reconfiguration methods designed for distributed routing systems. Skyline is not a reconfiguration method in itself. It is rather an auxiliary mechanism for reconfiguration of Up*/Down* routed interconnection networks. Skyline singles out a confined

part of the topology where a complete reconfiguration must be performed, and can thereby support faster reconfiguration processes.

NetRec is not tied to a particular routing algorithm, and does not depend on the availability of VCs. With NetRec, the reconfiguration process a ects a confined region around each component fault. Included in this region are the immediate neighbours of a faulty component and the intermediate components traversed to reestablish the connections among the immediate neighbours. Within the region a number of messages are exchanged in order to update the routing tables, and during this update application tra c is not accepted.

Agent NetReconf is fundamentally similar to NetRec, but uses autonomous mobile agents rather than message exchange. Furthermore, its complexity is reduced. In [7], Agent NetReconf was enhanced to handle a situation where a new fault occurs during an ongoing reconfiguration.

LORE depends on virtual layers for deadlock avoidance while rerouting packets around the area defined as a ected by a fault. Like OR, LORE supports in-order packet delivery during a reconfiguration process.

The dynamic reconfiguration methods referred to above are general approaches that are not tied to a particular interconnection network technology. The approaches presented in [257] and [25], on the other hand, target the InfiniBand technology. In [257], a realization of Double Scheme in InfiniBand, solely utilizing standardized properties, is demonstrated. For an Up*/Down* routed interconnection network based on InfiniBand, [25] covers the entire management process, from the detection of a topology change up to and including the change of routing function. The solutions presented are in accordance with the InfiniBand specification, and the reconfiguration strategy included (which is referred to as "pseudo-dynamic") achieves deadlock-freedom by disallowing a limited set of turns.

2.8.3 Selected processor allocation strategies

For the contiguous version of the distributed Leak [194, 195] processor allocation strategy, the following allegory of leaking water is used to illustrate the main concept. An amount of water, equivalent to the requested number of nodes, leaks out from an unallocated starting node, through other unallocated nodes, until the requested number of nodes are covered by water. Thus, Leak supports partitions of arbitrary shapes, and thereby reduces the fragmentation problem. Tra c-containment is not ensured when DOR is used, however. (For the communication experiments of [194], XY-routing was used in combination with a non-contiguous version of Leak.) In [194, 195], Leak mainly seems to be directed towards mesh and torus topologies. The basic principles of the strategy resemble a traditional breadth-first search, however, and a processor allocation strategy applying a breadth-first search around an unallocated node would not be restricted to a particular topology.

To the best of our knowledge, few of the existing contiguous processor allocation strategies are independent of topology. Therefore, for lack of contiguous topology agnostic strategies, we also include non-contiguous topology agnostic strategies in the following paragraphs.

The Generic Algorithm presented in [123] strives for compact partitions. It is a polynomial time approximation of the NP-hard [79] problem of selecting a given number of nodes from a defined set of nodes, while minimizing some distance metric between the selected subset of nodes. The Generic Algorithm is not tied to a specific topology, but generally expects distances to satisfy

the triangle inequality. Non-contiguous partitions are allowed.

Random [137] is a non-contiguous processor allocation strategy that allocates the requested number of nodes to a parallel job without regard to their relative location. Thus, no attempt is made to achieve a compact partition. A likely result is a significant interference between the tra c of concurrent jobs. Random, as a fragmentation-free processor allocation strategy, is used in Chapter 6 to indicate the upper limit for resource utilization.

Chapter 3

Research methods

An important part of the development of a new solution for a particular problem (for instance within processor allocation, deadlock-free routing and reconfiguration – the research areas of interest in this thesis) is an evaluation of the performance of a new method relative to the performance of existing methods. Furthermore, an evaluation of a method's sensitivity towards certain parameters may be needed.

Three main alternative approaches exist for undertaking a performance evaluation – analytical modelling, simulation and measurement [108]. The design of an analytical model normally involves considerable simplifications, and when using analytical modelling the accuracy of the results of a performance evaluation is expected to be low [108]. The systems considered in this thesis are relatively large and complex, and due to an in general limited scalability of analytical models we do not expect such models to be useful for our studies. A performance evaluation based on measurements depends on access to a relevant computing system where new and existing solutions could be implemented and their performance measured. Some of the solutions investigated in this thesis depend on mechanisms that are not available in current hardware. Furthermore, the investments related to purchase and maintenance of the necessary amount of relevant hardware for interconnection network research are prohibitive. Thus, for the interconnection network research community performance evaluation by simulation is a common choice. We also chose simulation for the performance evaluations of this thesis, and our simulator models are described in Section 3.1.

Our simulation experiments were run on a Condor [50] cluster available at the University of Oslo [241]. Statistics gathered during the simulations were dumped to text files. These files were parsed by scripts written in the Perl programming language [205], and Gnuplot [82] was used to plot the results.

A number of di erent metrics were applied for the studies included in this thesis. The relevant metrics for each individual study are presented in Chapters 4, 5 and 6.

In order to get an overview of existing approaches to solve a given problem – including their strengths and limitations – a literature study is needed in most research projects. Relevant literature for this thesis have mainly been obtained from full text libraries such as IEEE Xplore [103], ACM Digital Library [5], SpringerLink [226], and CiteSeer^x [47]. In addition, we have used the Inspec [104] reference database. Most of these services are accessible via the University of Oslo Library – Science Library, Informatics [242]. The library has also been helpful in providing information that was not available on-line. Google Scholar [86] has been a useful tool in the search for research papers.

3.1 The simulator models

Two different discrete-event simulator models – the *communication simulator* and the *allocation simulator* – are used in the performance evaluations of this thesis. The communication simulator is used to model detailed packet transfer from a source processing node, via intermediate switches and communication channels (possibly including storage in ingress and egress buffers of the switches), towards a destination processing node. The allocation simulator is used to model the assignment of a set of nodes (a partition), from the pool of available nodes, to each job in a series of incoming jobs.

Both of the simulator models are developed in the J-Sim [240] environment. The models are written in the Java programming language [109], and are built on top of the lowest layer of J-Sim.[1] This layer includes an implementation of J-Sim's architecture of autonomous components and simulation runtime environment. A simulation experiment is initialized by a seed obtained by the random integer generator available in the java.util package.

Our simulator models support evaluation of interconnection networks with a number of different topologies. The solutions proposed in this thesis are topology agnostic and do not assume a particular topology. Mesh and torus topologies (with or without component faults) have been used in our simulation experiments.

Both the communication simulator and the allocation simulator measure time in *cycles* – an abstract time unit. However, an allocation simulation typically represents jobs that may run for seconds, minutes or hours, whereas a typical communication simulation could represent packet exchange over a few milliseconds. Some of the processor allocation experiments require evaluation of internal communication among nodes that cooperate in executing a job. The different time-scales of the two types of experiments make this task challenging. The completion of a significant number of jobs is needed to obtain representative results for the allocation experiments, and the exchange of a significant number of packets is needed to obtain representative results for the communication experiments. In order to overcome these challenges, the allocation simulator outputs a given number of snapshots of currently allocated partitions. These snapshots are evenly spread out over the data collection period, and are stored for subsequent input to the communication simulator. There, packet-exchange experiments are performed only for this set of snapshots.

Sections 3.1.1 and 3.1.2 present more details of the communication and allocation simulator models, respectively. (The detailed setup of each simulation experiment is described in Chapters 4, 5 and 6.) Validation and verification of the simulator models are discussed in Section 3.1.3.

3.1.1 The communication simulator

The basic building blocks of our communication simulator are switches and processing nodes that are connected by communication channels. For each experiment, the topology (including the size) of the interconnection network and the number of virtual channels (VCs) available are set. The mesh and torus topologies used in our experiments are implemented as indirect networks, where one processing node is connected to every switch. The communication simulator supports faults of both switch ports (communication channels) and entire switches, and various routing algorithms

[1] We do not use the network models offered by the higher layers of J-Sim, as these are mainly modelled on the Internet.

and reconfiguration strategies are available.

A transparent synthetic workload model is applied in the experiments. A normal approximation of the Poisson distribution is used to decide the interarrival time of packets. This means that the packet injection rate is decided by two parameters – mean and standard deviation – which are input to a normal distribution for which any output value below 1 is set to 1. Several tra c patterns (destination address distributions) are supported by the simulator model, and the tra c pattern to use is a parameter in each experiment. In all of our experiments, packets are exchanged solely between processing nodes belonging to the same partition. In many cases, such as for the routing and reconfiguration experiments presented in Chapters 4 and 5, the entire system constitutes a single partition. However, as described above, communication experiments for a set of concurrent partitions are also supported.

In all of our experiments, the total size of a packet, including header and payload, is 256 bytes. The size of the packet header is not specified, but assumed to be negligible compared to the size of the entire packet. A packet size of 256 bytes is supported by such technologies as Dolphin Express [122] (see e.g. [120]) and InfiniBand [105] (see e.g. [177]).

Virtual cut-through switching and credit-based flow control are implemented in the simulator model. Switches are non-blocking, which means that packets heading for di erent egress ports are not a hindrance to each other. In the cases where several packets are competing for the same egress port, priority is given to the packet that has the earliest arrival time in its ingress bu er. Thus, a fair scheduling policy is implemented.

A source processing node has a transmission queue per VC of finite size. This queue overflows when the network cannot deliver packets at the rate they are injected, and the size of the queue is set for each experiment.

For the packet size used, two cycles are needed to transmit an entire packet onto a communication channel. From the first bit of the header is transmitted onto the communication channel until the entire header is received in the bu er at the other end of the communication channel, two cycles are consumed. (Except, as discussed in Section 3.1.3, on the last leg of a packet's route – from the last switch towards the destination processing node – where only one cycle is consumed.) We assume high capacity processing nodes, such that packets are immediately removed from the network upon reaching their destination processing node.

The ingress and egress bu ers per VC of a switch port apply first in, first out (FIFO) ordering, and the size of these bu ers is a simulation parameter. Let us assume that an egress bu er, eB, has free space for at least one packet, and that no other packet is currently in transfer across the switch fabric from an ingress bu er, iB, towards eB. Then, from the header of a packet is available at the front of iB, one cycle is consumed for routing, arbitration and transfer of some initial bits of the packet across the switch fabric and into eB, where transmission onto the next communication channel could potentially start immediately. The transmission rate within a switch, from an ingress bu er onto the internal switch fabric, equals the transmission rate onto an external communication channel. This means that there is no speed-up within a switch, and that two cycles are needed for transmitting an entire packet.

The length of the data collection period, a number of cycles, is set for each experiment. For most of our experiments (an exception is found in Chapter 6), the steady-state behaviour of the system is of interest. Thus, for these experiments, data collected during the initial transient period (while the occupancy of bu ers is increasing and the system has not yet reached a balanced condition) should

not be included in the final statistics. Several approaches for deciding the length of the initial transient period are discussed in [172] and [108]. In our experiments, two di erent approaches are used. For the first experiments, mean latency was computed and collected for a moving window of 10 consecutive batches. Based on the formula for a least squares regression line given in [28], a line representing this data set was identified. We assumed that the first occurrence of a zero or negative value for the line's gradient indicated the end of the transient period. However, in some cases, this method seemed to suggest a duration of the initial transient period which was shorter than the actual transient period. Therefore, for subsequent experiments another approach was used. This approach simply involves setting a fixed length of the transient period which is significantly longer than any transient period found by the first approach. No data are collected after packet generation is stopped, while the final packets are drained from the interconnection network.

3.1.2 The allocation simulator

Main components of the allocation simulator are a job generator, a queue of infinite size of parallel jobs awaiting execution, a first-come, first-served scheduler, and an allocation module. For a 4×4 mesh topology, Figure 3.1 illustrates the homogeneous environment assumed in our experiments, where all the resources are equivalent and where single processors are the only type of resource eligible for allocation. For simplicity, combined nodes are assumed.

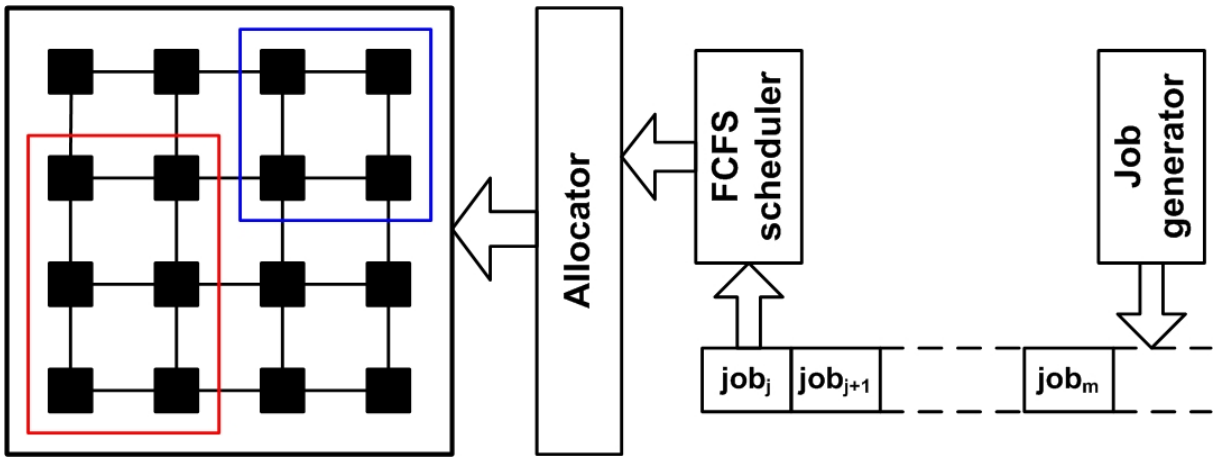


Figure 3.1: The allocation simulator model.

A synthetic workload model is used in the experiments. The interarrival times of jobs into the queue are exponentially distributed. Each job has certain resource requirements, which either include a number of nodes, $|R|$, or a defined sub-mesh of $a \times b$ nodes. The value of $|R|$ is drawn from a uniform distribution with minimum size $|R|_{min}$ and maximum size $|R|_{max}$. The values of a and b are drawn from separate uniform distributions with minimum sizes a_{min} and b_{min} and maximum sizes a_{max} and b_{max}, respectively.

Upon termination of a running job, or upon arrival of a new job (if the queue was previously empty), the scheduler selects the first job(s) in the queue as the next candidate(s) for assignment of nodes.

The allocation module runs one of the various processor allocation algorithms supported, and attempts to localize a set of free nodes that meets the resource requirements of the scheduled job.

We adopt the common assumption (see e.g. [127]) that, after a successful assignment of nodes to a job, the job runs uninterruptedly until completion on this set of nodes. The running time is drawn from an exponential distribution, and will be referred to as the job's *service time*.

In our experiments, the nodes are interconnected in a mesh or torus topology. The input *load* for a mesh or torus of width w and height h is $\frac{|R|_{mean} \times ST_{mean}}{w \times h \times IT_{mean}}$ (in accordance with [44]), where $|R|_{mean}$ is the mean number of nodes requested by the jobs, ST_{mean} is the mean service time of jobs, and IT_{mean} is the mean interarrival time of jobs.

Each experiment runs until a number of jobs, given by a simulation parameter, have been allocated and completed. Both the initial and final 10% of the observations are discarded in order to exclude data collected during initial and final transient periods from the results. However, a steady state can only be reached for input load levels less than or equal to 1. For higher load levels the arrival rate of jobs is higher than the departure rate, and the number of jobs in the infinite queue will increase until the generation of new jobs is stopped.

3.1.3 Validation and verification of the models

According to [108], validation of a simulator model involves an evaluation of how well the system being modelled is represented, and, thus, to what extent the design of the model is based on appropriate assumptions. Verification of a simulator model, on the other hand, is an assessment of whether its implementation is of su cient quality, that is, without serious programming errors [108].

Our models have been verified through incremental testing and debugging during the development process, where a number of serious and less serious programming errors were identified and corrected. However, after our experiments were performed, we became aware of one remaining implementation error in the communication simulator that has an influence on packet latency. For the final communication channel traversed by a packet, the time consumption is only one cycle (not two cycles as for the other communication channels) from the first bit of the header is transmitted from the port of the final switch until the header is received by the destination processing node. Due to this error, the latency registered for each packet is one cycle too short. However, as the period of time that each packet occupies the communication channel is correctly set, the arrival times of subsequent packets are not a ected. For our experiments, the relative behavior of di erent algorithms, as well as an algorithm's sensitivity towards certain parameters (which are una ected by the implementation error), are of greater interest than absolute figures for latency. In summary, the remaining implementation error is a constant error that does not a ect our conclusions.

Part of the process of validating our simulator models involved a comparison of their output with the output of other simulators. For instance, the relative performance of some existing routing algorithms, according to our communication simulator, was compared and found su ciently similar to the output of another simulator model[2] already in use at Simula Research Laboratory.

One version of the communication simulator was implemented solely for the purpose of improved modularity and parametrization, and is basically similar to the communication simulator applied in this thesis. For this version of the communication simulator, after a careful calibration of parameters a close match was established with throughput and latency results measured on a small

[2]This model was e.g. used in [161, 234].

InfiniBand cluster available at Simula Research Laboratory.

In a confidential report submitted to the EU-funded SIVSS project [202] in 2005, an adapted version of our communication simulator was compared to a simulator that was developed at Xyratex [254] in order to support the development of a new Advanced Switching Interconnect [17,150] switch. For the packet size used in our experiments, a high similarity between the throughput results of the two models was demonstrated.

For some existing processor allocation algorithms, such as First-Fit and Best-Fit [262], system utilization (or fragmentation) results found in the literature (see e.g. [137]) were compared and found similar to the results output from our allocation simulator.

In the following, we include a discussion about some of the decisions that were made, both regarding the development of the simulator models and the design of the experiments. Both the communication and allocation simulators use simple synthetic workload models. Compared to synthetic models, traces often constitute more realistic workloads, and could have further added credibility to some of our conclusions. However, care must be exercised when using traces in order to verify their relevance to the system under study. For instance, for the studies of processor allocation presented in Chapter 6, a trace of job arrivals, job sizes, etc. from one data center or supercomputer is not necessarily representative of the workload of another data center or supercomputer. To the best of our knowledge, representative traces do not exist for internal packet exchange within the arbitrarily shaped partitions that we consider. It was stated in [136] that the relative performance (the ranking) of the investigated allocation and scheduling algorithms was similar regardless of whether a synthetic workload model or a trace was applied. We are mainly interested in the relative performance of algorithms – not in absolute performance figures – and do not expect that the use of traces would have significantly altered our conclusions.

Although multistage topologies such as fat-trees are widespread in several types of systems, we also believe that meshes and tori are relevant topologies for our experiments. For the processor allocation studies, the choice of mesh and torus topologies allows a comparison between our new allocation strategies and a number of allocation strategies previously proposed for these topologies. Mesh and torus topologies have advantages as networks can be expanded without requiring larger switches, and communication channels can be kept short. Such topologies are often preferred for on-chip networks [56, 124]. Furthermore, a significant share of the machines at the top of the Top500 list of supercomputers [237] are based on torus topologies. Many current data centers are based on the Ethernet technology [102, 206], and tree topologies are commonly used as they match the spanning tree protocol [207] which is Ethernet's original routing algorithm. As suggested in [183], a simple modification of Ethernet's flow control mechanism will enable use of the topology agnostic routing algorithm LASH [144,218]. The use of LASH and other flexible routing algorithms could facilitate a more widespread use of mesh and torus topologies in data centers based on Ethernet.

In our communication simulator, two cycles are required for transmission of a 256-byte packet onto a communication channel. Thus, the transmission rate of a communication channel is 128 bytes (1024 bits) per cycle. The corresponding transmission rate, tr, in gigabits per second depends on the length of a cycle, cl, in nanoseconds (ns) as follows: $tr = \frac{1024}{cl}$. Thus, roughly, in order to model for instance the data rate of the 10 Gigabit Ethernet technology [102], a cl of 102.4 ns could be assumed. On the other hand, for the absolute figures (e.g. for packet latency)

output from our communication simulator to be fully representative of a particular interconnection network technology, the time consumption in various parts of the system should have been further calibrated. As previously mentioned, our main focus is on relative performance rather than on absolute figures of performance. Issues such as the acceptable running time of experiments and the required accuracy of results were considered when the level of detail to include in our simulator models was decided. Such considerations are generally needed in modelling activities.

The interface between the allocation simulator and the communication simulator comprises a given number of snapshots of current partitions, which are output from the allocation simulator for subsequent input to the communication simulator. Ideally, communication experiments should be executed for every partition. Due to the di erent time scales of the two simulator models, and the number of jobs that should be studied, such experiments would not finish within an acceptable period of time. The chosen solution is a compromise that allows us to run long and detailed simulations of the exchange of packets for a set of snapshots. The main unfavourable consequence of this arrangement is perhaps that, for the results output from the allocation simulator, any communication overhead is not reflected in the service time of a job, and neither in the queuing time of subsequent jobs.

Using the already available ProcSimity simulator tool [252] could perhaps have been a reasonable alternative to developing our own allocation simulator. In retrospect, this alternative could have been more thoroughly considered. (Although we do not have indications that our conclusions would have been influenced by the use of a di erent simulator tool.)

Our studies of processor allocation focus on the e ect of the allocation strategy, in particular on metrics such as system utilization and fragmentation. We are aware that, given an adequate mix of job sizes, backfilling scheduling could also reduce fragmentation and improve the utilization of a system's resources. A backfilling scheduler allows smaller jobs to bypass larger jobs in the queue under such conditions that the first job in the queue is not delayed – as in the Extensible Argonne scheduling system (EASY) [133, 220] – or – as in more conservative approaches – that none of the jobs ahead are delayed. A comparison of these two variants of backfilling is provided in [159], whereas [70] discusses various theoretical and practical approaches to scheduling. Nevertheless, in order to emphasize the e ect of the allocation strategy, we apply a simple first-come, first-served scheduler in our experiments.

As recommended in [108, 172], an experiment should be stopped when the desired level of accuracy of the results (normally based on the width of confidence intervals) is ensured. However, neither of our two simulator models includes an on-line calculation of confidence intervals that could help deciding the running time of a simulation. Thus, as previously mentioned, each experiment is stopped after the execution of a predetermined number of cycles or jobs. Thereafter, we have to control, o -line, that the results have su cient accuracy.

Chapter 4

Routing

An important aspect of the operation of an interconnection network is the ability to identify a routing function that allows e cient transport of packets from their sources to their destinations. This chapter is mainly based on a study [224] that aims at enabling e cient routing in a specific interconnection network technology – Advanced Switching Interconnect (ASI) [17, 150]. ASI is a source routing technology that extends PCI Express [173]. PCI Express only supports communication within a system that consists of a single master processing node (which controls the communication) and a number of slave nodes. ASI addresses this limitation, and enables communication between multiple autonomous processing nodes.

No particular routing algorithm was declared in the ASI specification [17]. A main goal of our study is to identify a suitable routing algorithm for ASI among a large number of routing algorithms available in the literature. Section 4.1 states a set of requirements for a routing algorithm to be used with ASI. We demonstrate how one routing algorithm, Layered Shortest Path (LASH) [144, 218], matches all of these requirements, and we recommend that LASH is used with ASI.

Section 4.2 discusses a number of issues related to realization of LASH in ASI, and Sections 4.4 and 4.5 include a performance evaluation. For instance, we investigate some static fault tolerance characteristics of LASH. (This investigation is based on a confidential report submitted to the EU-funded SIVSS project [202] in 2005.) Moreover, we propose and evaluate an optimization of LASH with respect to route calculation for mesh topologies.

At the time our study was conducted, the ASI technology attracted much attention. Later, the ASI specification process was discontinued. The results of our study are still of interest, however, as the ASI technology is now incorporated in Dolphin Express [122].[1] In addition, most of our considerations regarding routing in ASI are also relevant for routing in InfiniBand [105]. In Section 4.3 we discuss how the requirements for a routing algorithm to be used with ASI are also reasonable guidelines for selecting a routing algorithm to be used with InfiniBand. Moreover, we discuss how a number of issues related to realization of LASH in ASI are also relevant for realization of LASH in InfiniBand. These considerations – and the conclusion that LASH is also a highly suitable routing algorithm for InfiniBand – had a significant impact on the inclusion of LASH in the OpenFabrics Enterprise Distribution [163] software stack for InfiniBand.

Section 4.6 discusses related work and our contribution; Section 4.7 includes critical comments regarding our own work; and Section 4.8 presents some remaining issues for future research.

[1] The technology is referred to as ASI throughout this chapter.

4.1 Requirements to a routing algorithm for ASI

As ASI uses source routing (except for multicast), the routing algorithm must support a deterministic routing approach. This requirement excludes purely adaptive algorithms such as Adaptive Trail [182] and Minimal Adaptive [212], but includes algorithms such as Up*/Down* [199, 204] and Transition-Oriented Routing (TOR) [200], which in principle support both deterministic and adaptive routing.

The rest of this section presents and discusses the following set of additional requirements that a routing algorithm for ASI should fulfil. The routing algorithm should be *deadlock-free*, support *shortest path routing*, and require *no transitions from one virtual layer to another*. It should *not assume any particular topology*, but nevertheless be *highly e cient for regular topologies*. We give reason for each of these requirements, and demonstrate how all but one of the candidate routing algorithms fall short of one or more of the requirements.

4.1.1 No assumptions on topology

ASI itself does not limit the choice of topology for an interconnection network, and neither should its routing algorithm introduce such a limitation. Thus, a routing algorithm for ASI should be topology agnostic.

Other arguments in favour of using a topology agnostic routing algorithm for ASI are related to fault tolerance and incremental scalability. A topology agnostic routing algorithm can calculate a connected routing function for any network topology, including an originally regular topology with faulty switches or communication channels – given that the resulting topology is physically connected. A topology specific routing algorithm, on the other hand, may not be usable for a non-regular topology that results from faulty network components. Furthermore, a topology agnostic routing algorithm allows flexibility with respect to an extension of a network or a join of di erent networks, whereas a topology specific routing algorithm generally requires that the regularity of a network topology is strictly maintained.

The requirement of topology agnosticism excludes such routing algorithms as Dimension-Order Routing (DOR) [231] and those based on the Turn Model [80]. In [75] a thorough overview and evaluation of topology agnostic routing algorithms are given. The most relevant topology agnostic routing algorithms include Up*/Down*, Left-Up First Turn (L-Turn) [117], Flexible Routing (FX) [197], Multiple Roots (MRoots) [71, 143, 144], Segment-Based Routing [151][2], Smart Routing [41], LASH, TOR, Descending Layers [119], and LASH-TOR [217]. All of these algorithms were described in Section 2.8.1.

4.1.2 Deadlock-freedom

The deadlock issue is one of the main concerns in the design of a routing algorithm for an interconnection network. Section 2.3 explained that interconnection networks are prone to deadlocks due to their support of lossless communication (which typically requires enforcement of flow control across each communication channel).

[2]Segment-Based Routing was published after our study [224] was performed, but is nevertheless included in this thesis.

According to [68], deadlock recovery mechanisms (see e.g [13, 116]), which discover and resolve – rather than avoid – deadlocks, are only usable if deadlocks occur infrequently and can be endured. For ASI – an interconnection network technology designed to support various higher layer protocols in diverse communication, storage and embedded systems – such an assumption on the frequency and acceptability of deadlocks should not be made. A routing algorithm for ASI should neither rely on mechanisms for deadlock prevention. In order to prevent deadlocks such mechanisms allocate the resources to be used in advance, which may significantly limit the utilization of a network's resources [68]. Thus, a routing algorithm for ASI should comply with the principles presented in [55], where it is stated that, for a deterministic routing function, an acyclic channel dependency graph is a prerequisite for deadlock-freedom.

All of the topology agnostic routing algorithms listed in Section 4.1.1 are deadlock-free. Thus, the requirement on deadlock-freedom alone does not exclude any of these algorithms. As explained in Section 2.8.1, these algorithms take di erent approaches to deadlock-freedom. Up*/Down*, L-Turn and MRoots base the calculation of deadlock-free paths on a spanning tree of the topology; LASH and LASH-TOR use virtual channels (VCs); whereas TOR and Descending Layers use both a spanning tree and VCs. Segment-Based Routing divides the topology into segments and applies restrictions on turns independently for each segment, whereas Smart Routing uses heuristics and greedily breaks cycles in the channel dependency graph.

4.1.3 Shortest path routing

In order to e ciently utilize an interconnection network's resources, switch-based technologies such as ASI should apply shortest path routing. Thus, the routing function should ensure that the path provided from a source processing node s to a destination processing node d is no longer than the shortest distance from s to d in the physical topology.

Among the topology agnostic routing algorithms listed in Section 4.1.1, LASH, TOR, Descending Layers and LASH-TOR can support shortest path routing – given that a su cient number of VCs are available. Up*/Down*, L-Turn, FX, MRoots, Segment-Based Routing and Smart Routing, on the other hand, do not support shortest path routing.

4.1.4 No transitions from one virtual layer to another

Recall from Section 2.2.2 that a *virtual layer* is a virtual network that consists of all the switches and processing nodes of the physical network. In addition, the virtual layer includes a subset of the VCs such that, for each physical communication channel, one VC in each direction belongs to the virtual layer. (Also recall that a virtual layer is sometimes referred to as a *layer* for short.)

Up to 16 VCs for unicast tra c (8 ordered-only and 8 bypass-capable) are supported by the ASI specification. In ASI, rather than letting each packet hold explicit information about VC-use, each communication channel has a tra c class (TC) to VC mapping. A TC value is included in the packet at the time of its creation and kept during its lifetime. For each communication channel to traverse on a packet's path from its source to destination, the VC to be used is decided by the communication channel's TC to VC mapping. This means that ASI cannot support deadlock avoidance mechanisms where packets must perform transitions from one virtual layer to another. This excludes routing algorithms such as LASH-TOR, TOR and Descending Layers.

4.1.5 High e ciency for regular topologies

ASI supports arbitrary interconnection network topologies. Nevertheless, regular network topologies such as meshes, tori and fat-trees may in many cases be preferred. Thus, for a particular regular topology, the topology agnostic routing algorithm chosen for ASI should ideally achieve similar performance as a routing algorithm tailored to the topology. Furthermore, the chosen topology agnostic routing algorithm should perform well for non-regular topologies that may result from failing switches or communication channels of originally regular topologies.

Among the alternative topology agnostic routing algorithms listed in Section 4.1.1, LASH is the only algorithm that has not been excluded by any of the previous requirements. Results presented in [224] show that, for a uniform tra c pattern, LASH (in its original version) achieves a significantly lower throughput than DOR achieves for a 6×6 mesh topology. However, the di erence in performance between DOR and LASH decreases for a pairwise tra c pattern, and both for a neighbour and hotspot tra c pattern LASH performs slightly better than DOR. In Section 4.2.5 we introduce an optimization of LASH for mesh topologies which enables LASH to select the same routes as DOR, and thus to achieve equal performance for any tra c pattern. The performance evaluation presented in Sections 4.4 and 4.5 considers the optimization of LASH for mesh topologies as well as some static fault tolerance qualities of LASH for mesh and torus topologies.

We conclude that LASH is a suitable routing algorithm for ASI. Issues related to realization of LASH in ASI are discussed in Section 4.2.

4.2 Realization of LASH in ASI

Section 4.1 demonstrates that, whereas a number of existing algorithms do not meet the requirements stated for a routing algorithm to be used with ASI, one algorithm – LASH – fulfils all of these requirements. Thus, we recommend LASH as the routing algorithm to be used with ASI.

For our purposes, LASH is a deterministic shortest path routing algorithm within the category of Layered Routing [144]. Layered Routing is an approach that uses virtual layers to obtain topology agnostic, deadlock-free routing in interconnection networks.

In the following, we explain how LASH works, how it can be implemented in ASI, and elaborate on how it fulfils the requirements stated in Section 4.1. We propose a service di erentiation mechanism, an optimized route selection for mesh topologies, and study some static fault tolerance qualities of LASH.

In order to compute the routing function, LASH first explores the network topology and identifies the shortest paths between all pairs of source and destination switches[3]. Second, to ensure deadlock-freedom, these shortest paths are distributed among virtual layers, such that no layer contains any cyclic channel dependency. One virtual layer is available initially, and additional layers are put to use on demand. A packet is forwarded in the same virtual layer all the way from its source to its destination.

LASH is applicable for source routing interconnection network technologies such as ASI, where the complete path from the source to destination is decided by the source processing node and included in the header of each packet. (LASH is also applicable for interconnection network tech-

[3] One or more source processing nodes are attached to a *source switch*, whereas one or more destination processing nodes are attached to a *destination switch*.

ASI ROUTING REQUIREMENT	LASH PROPERTY
N	T
D -	E - -
S	U D ’
N	A
H	O DOR

Table 4.1: How LASH fulfils the requirements to a routing algorithm for ASI.

nologies based on distributed routing, such as InfiniBand.) Table 4.1 summarizes how LASH fulfils the requirements to a routing algorithm for ASI that were stated in Section 4.1.

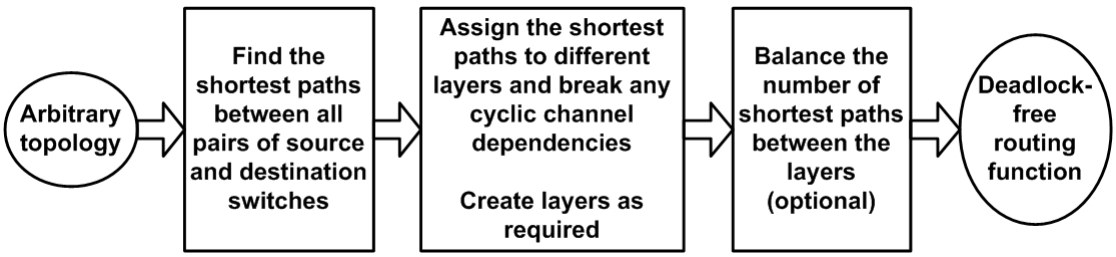


Figure 4.1: The LASH routing algorithm.

The use of LASH in ASI requires implementation of the following three fundamental steps that are also outlined in Figure 4.1:

1. LASH determines the shortest paths between all pairs of source and destination switches, using a suitable algorithm such as Dijkstra’s [60]. There may be several alternative shortest paths from a source to a destination. In order to balance the tra c load, LASH may randomly select one of the paths, or a more advanced tra c balancing concept could be applied. (See e.g. [41, 118, 196] for examples of tra c balancing algorithms.)

2. LASH ensures deadlock-freedom by analyzing channel dependencies in order to ascertain which of the paths can be placed in the same virtual layer. For each shortest path, LASH searches for a virtual layer where the path may be included without introducing a cyclic channel dependency. If a virtual layer is found, the path is assigned to it. Otherwise, a new virtual layer is created, and the path is included there. Section 4.2.2 discusses how the resulting number of virtual layers depends on such factors as topology and network size.

3. At this point LASH has settled the number of virtual layers required. However, the numbers of paths assigned to each virtual layer may di er. Without a layer balancing mechanism, densely filled layers may experience performance degradation compared to scarcely filled layers. In order to achieve as high performance as possible, the last (and optional) step of the LASH algorithm balances the number of paths in each virtual layer. This ensures similar

levels of head-of-line blocking within the layers, and is feasible since most of the paths can be assigned to more than one layer.

4.2.1 Complexity

In the selection of a routing algorithm, its computational complexity is an important concern. According to [144], for an interconnection network that includes N switches, the complexity of LASH is $N^2 \times l \times N$, where N^2 is the number of pairs of source and destination switches; l is the number of virtual layers; and N comes from inspection of cyclic channel dependencies. Tests performed in [144] indicated that, for current computers, the relatively high complexity of LASH is not an issue for interconnection networks consisting of less than 256 switches. We note that the size of ASI fabrics is limited by the length of a packet's routing structure (turn pool) of only 31 bits [17].

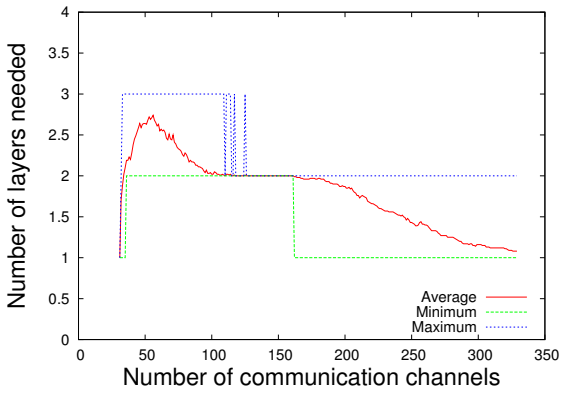
However, if the complexity of LASH should become a problem, an alternative mode of operation of LASH o ers reduced complexity at the expense of a slightly increased number of layers. Here, LASH reduces the requirement of a single shortest path from each source to each destination, allowing for a simultaneous evaluation of a tree of shortest paths from a single source to multiple destinations. In [218], this mode is referred to as *directed acyclic graph granularity*, whereas the mode explained above is referred to as *single source-destination granularity*.

4.2.2 Need for virtual layers

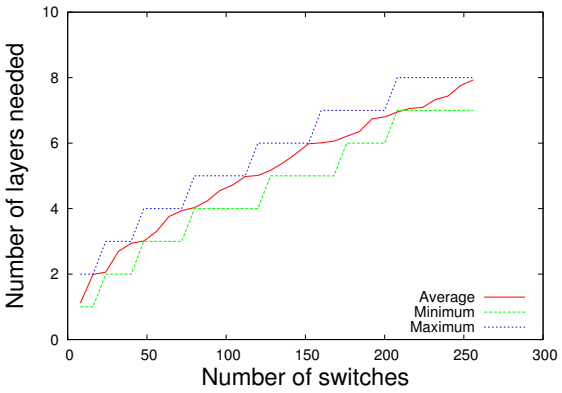
Interconnection network technologies traditionally provide VCs for three main purposes: quality of service, deadlock-free routing, and improved network performance (due to reduced head-of-line blocking). ASI provides a limited number of VCs, and, according to [17], these primarily seem to be intended for the provisioning of quality of service. Thus, occupation of an unreasonable amount of these resources on the purpose to achieve deadlock-freedom is not acceptable if di erent levels of service are needed.

The number of virtual layers required by LASH mainly depends on the topology, size and connectivity of an interconnection network. For irregular topologies, the need for layers has been thoroughly investigated in [144, 218, 234]. For an interconnection network consisting of N switches, the theoretical upper bound on LASH's layer requirements given in [218, 234] is as high as $\frac{N}{2}$. However, it is indicated that a tighter upper bound may exist, and the simulation experiments included in [144, 218, 234] suggest than the practical number of virtual layers needed by LASH is moderate. Figure 4.2(a) [234] depicts how the need for layers varies with the number of communication channels for an irregular network consisting of 32 switches. When the number of communication channels is around twice the number of switches, the requirement for virtual layers is at the highest. For this degree of connectivity, Figure 4.2(b) [234] shows the number of virtual layers required as the number of switches increases. It was stated in [234] that the need for virtual layers appears to increase logarithmically as the number of switches grows – and, thus, that a modest number of layers are needed, even for large irregular interconnection networks.

Let us now turn our attention to some well known regular topologies. For a fat-tree, being representative of multistage networks, there are no cyclic channel dependencies between the shortest paths in the absence of faults. Hence, LASH requires only a single layer.



(a) The need for virtual layers as connectivity increases (32 switches).



(b) The need for virtual layers as the number of switches, N, increases. The number of communication channels is $2N$.

Figure 4.2: The need for virtual layers as a function of connectivity and size of irregular topologies (numbers obtained from [234]).

For tori, as with irregular topologies, the need for layers generally increases as the size of a network grows. Our simulation experiments indicated that for a 6×6 torus LASH needs at most 4 layers; for an 8×8 torus 5 layers are normally required (6 layers are required in rare cases); whereas for a 12×12 torus 8 layers are normally required (9 layers are required in rare cases). For a two-dimensional mesh of arbitrary size, on the other hand, our simulations show that LASH requires at most 2 layers.

The virtual layer requirements stated above for some regular topologies are based on the assumption that the switches are interconnected via arbitrary port numbers. In Section 4.2.5, we suggest an optimization for mesh topologies that enables LASH to select the same paths as DOR. When using this optimization, LASH requires only one virtual layer in a mesh topology.

For interconnection networks with originally regular topologies, failing switches and communication channels may have an impact on LASH's requirements of virtual layers. In [132], only a moderate increase (or no increase at all) in the required number of layers was found when LASH's layer requirements were analyzed for a 18×18 torus with 0, 1, 2 and 3% of its communication channels failing. For up to 8 switch faults, our own simulation experiments indicated that at most 4 virtual layers are needed for an 8×8 and 16×16 mesh (where 2 layers are needed in the fault-free case). For an 8×8 torus, at most 6 layers are needed (which is not a higher layer-requirement than occasionally observed in the fault-free case).

4.2.3 Deadlock-freedom and quality of service

When using LASH, a packet traverses an ASI fabric with a fixed TC-assignment. Section 4.1.4 described how each communication channel in the ASI fabric has a TC to VC mapping. It must be ensured that the TC to VC mappings are consistent with LASH's usage of layers. In order to maintain deadlock-freedom, each switch in the fabric must support a minimum number of VCs equal to the number of virtual layers required by LASH.

ASI specifies TCs for quality of service provisioning, and the level of service given to a packet depends on the packet's TC-value. In order to provide service di erentiation when LASH is used,

VIRTUAL LAYER	PRIORITY-CLASS
0	0
1	0
2	1
3	1
4	2
5	2
6	3
7	3

Table 4.2: Example of a layer to priority-class mapping in a case where LASH needs two virtual layers for deadlock avoidance and eight VCs are available.

we suggest to introduce a layer to priority-class mapping. In the cases where LASH needs less than half of the available number of VCs to ensure deadlock-freedom, at least two di erent priority levels can be provided for service di erentiation. However, in the cases where LASH settles for a single layer, no such novel mappings are required for service di erentiation.

As an example, let us assume that in a system where eight VCs are available LASH needs two virtual layers in order to ensure deadlock-freedom. Then, four di erent priority levels can be supported. Table 4.2 illustrates a possible layer to priority-class mapping for this case.

4.2.4 Static fault tolerance

Topology agnostic routing algorithms, such as LASH, do not assume that an interconnection network has a specific topology. An important consequence is that a topology agnostic routing algorithm can compute a connected routing function for a topology that results from faults in switches and communication channels of an originally regular topology (provided that the resulting topology is physically connected). If such faults are present at system start-up, no special precautions are required for calculation and application of the routing function. Thus, topology agnostic routing algorithms in general, and LASH in particular, inherently provide static fault tolerance.

As a static fault tolerance mechanism, an important quality of a topology agnostic routing algorithm is the ability to sustain an acceptable performance as switches and communication channels of an interconnection network fail. A gradual deterioration of performance – rather than a collapse – as an increasing number of faults occur is often referred to as a *graceful* performance degradation. Section 4.1.5 stated that a routing algorithm for ASI should also perform well for a non-regular topology that may result from failing network components in an originally regular topology. For a set of static fault scenarios, Sections 4.5.1 and 4.5.3 investigate how the performance of LASH is a ected by an increasing number of failing switches in a mesh or torus topology.

4.2.5 Route optimization for mesh

Section 4.1.5 stated that, for a particular regular topology, the topology agnostic routing algorithm chosen for ASI should ideally achieve similar performance as a routing algorithm tailored to the topology. Even though LASH is a topology agnostic routing algorithm, it is possible to configure LASH and connect a mesh in such a way that the selected paths correspond to the paths found

by DOR. In that case, a single virtual layer will su ce to ensure deadlock-freedom for a fault-free mesh.

The optimization strategy can be generalized to meshes of any dimension. Let us for the purpose of illustration consider a mesh with two dimensions, X and Y. Then, the main idea is, for every visited switch, to search for a path in the X-dimension before the Y-dimension. The first path found corresponds to the DOR (XY) path in a fault-free mesh. Assume that the four lowest port numbers in every switch are 0, 1, 2 and 3. Then, the optimization could, for instance, be implemented in ASI as follows. Connect the two-dimensional mesh such that the negative X-port is labelled 0, the positive X-port is labelled 1, the negative Y-port is labelled 2, and the positive Y-port is labelled 3. Configure LASH such that a shortest path from a source to a destination is searched for in increasing port number order in every visited switch, and select the first path found. This path complies with DOR (XY) in a fault-free mesh. (A search in decreasing port number order, on the other hand, would find a path that complies with YX.) With this configuration the topology agnosticism of LASH is kept, and LASH can still be used with other topologies – including a mesh topology with faulty switches or communication channels. (In that case a single virtual layer may no longer be su cient, however.) Sections 4.5.2 and 4.5.3 evaluate the route optimization.

4.3 Relevance for InfiniBand

Most of our considerations regarding routing in ASI are also relevant for routing in InfiniBand. Section 4.3.1 discusses how the requirements for a routing algorithm to be used with ASI are also reasonable guidelines for selecting a routing algorithm to be used with InfiniBand. Moreover, Section 4.3.2 discusses how a number of issues related to realization of LASH in ASI are also relevant for realization of LASH in InfiniBand.

4.3.1 Guidelines for selecting a routing algorithm

In Section 4.1 we argued that a routing algorithm for ASI should support deterministic routing, be topology agnostic and deadlock-free; provide shortest paths and high e ciency for regular topologies; and not require transitions from one virtual layer to another.

ASI is a source routing technology and does not support adaptive routing. Although InfiniBand is a distributed routing technology, adaptive routing is not supported in the current InfiniBand specification. Mechanisms to enable adaptive routing in InfiniBand have been proposed, however (see e.g. [148]).

A routing algorithm for InfiniBand should be topology agnostic for the same reasons as a routing algorithm for ASI should be topology agnostic: The InfiniBand technology itself does not limit the choice of topology for an interconnection network, and a topology agnostic routing algorithm is advantageous with respect to fault tolerance and incremental scalability. (Nevertheless, fat-trees and tori are much used topologies for current interconnection networks based on InfiniBand, and topology specific routing algorithms are commonly applied.)

A routing algorithm for InfiniBand should be deadlock-free. Recall that the usability of deadlock recovery mechanisms depends on an assumption that deadlocks occur infrequently and that running applications are able to endure a deadlock. Such assumptions are unreasonable for InfiniBand which – like ASI – supports di erent higher layer protocols in various types of computing

systems. Deadlock prevention mechanisms should neither be applied as such mechanisms may significantly limit the utilization of an interconnection network's resources.

E cient utilization of an interconnection network's resources is also a main argument for using shortest path routing in switch-based technologies such as InfiniBand (and ASI).

A fixed service level (SL) is assigned to a packet in InfiniBand (in like manner as a fixed TC value is assigned to a packet in ASI). The InfiniBand specification supports 16 SLs and 16 virtual channels (called virtual lanes – VLs). Each InfiniBand switch holds an SL to VL mapping, and the VL onto which a packet will be forwarded is found by a lookup based on the packet's SL, the input physical channel used, and the output physical channel to be used. As opposed to ASI, InfiniBand allows transitions from one virtual layer to another. Nevertheless, the use of a routing algorithm that depends on such transitions are challenging in InfiniBand. In [200] a thorough explanation is given of how mapping conflicts may occur – and be resolved – when using the TOR algorithm with InfiniBand. (Recall that TOR depends on virtual layer transitions for deadlock avoidance.) A mapping conflict may arise in a switch between packets that carry the same SL; that arrive at the switch in di erent VLs of the same input physical channel; and that will depart the switch in di erent VLs of the same output physical channel. Extra SLs are consumed in order to resolve mapping conflicts, and shortest path routing can only be supported if a su cient number of SLs are available. Results presented in [200] demonstrate how the number of required SLs grows as the size of interconnection networks (with irregular topologies) increases. The InfiniBand specification supports only a limited number of SLs. These are also intended for quality of service provisioning (so that a packet is given a level of service in accordance with the SL value included in its header). Thus, using a routing algorithm that consumes a large share of the available SLs may obstruct the support of service di erentiation. As explained in [200], mapping conflicts could have been avoided if the InfiniBand specification also included the input VL in the set of factors that decides the output VL. This would have improved InfiniBand's support of routing algorithms that depend on virtual layer transitions for deadlock avoidance.

As many computing systems are based on interconnection networks with regular topologies, a (topology agnostic) routing algorithm to be used with InfiniBand should be highly e cient for regular topologies as well as for non-regular topologies that may result from faults in originally regular topologies.

Section 4.1 showed how one of the evaluated routing algorithms – LASH – fulfils all of the requirements stated for an algorithm to be used with ASI. The discussion presented in this section shows that these requirements are also reasonable guidelines for a routing algorithm to be used with InfiniBand. Thus, we conclude that LASH is also a highly suitable routing algorithm for InfiniBand.

4.3.2 Realization of LASH in InfiniBand

In Section 4.2 we discussed a number of issues related to realization of LASH in ASI: computational complexity; need for virtual layers; deadlock-freedom and quality of service; static fault tolerance abilities; and a route optimization for mesh topologies.

The computational complexity of a routing algorithm is equally important for the InfiniBand technology as for the ASI technology. The complexity of LASH is relatively high – $O(N^3 \times l)$, where N is the number of switches in a network and l is the number of virtual layers required.

In comparison, the computational complexity of the TOR algorithm [200] that was proposed for InfiniBand is $O(N^3)$ – or $O(N^2)$ if a simpler mechanism is applied to balance tra c. Section 4.2.2 explained that the number of virtual layers required by LASH is normally moderate. Thus, the complexity of LASH is practically the same as the complexity of TOR (with its original tra c balancing mechanism). Whereas the size of an interconnection network based on ASI is limited by the length of a packet's routing structure, InfiniBand can support large networks. (The number of processing nodes on an InfiniBand network is limited by the 16 bit local identifier packet header field.) If the complexity of LASH should become a problem for a large InfiniBand network, recall from Section 4.2.1 that an alternative mode of operation exists for LASH. This mode of operation o ers reduced computational complexity at the expense of a slightly increased number of virtual layers.

Section 4.2.2 explained that the number of virtual layers required by LASH mainly depends on the topology, size and connectivity of an interconnection network. Recall that for irregular topologies, as well as for tori, the need for virtual layers generally increases as the size of networks grows. The main di erence between InfiniBand and ASI with respect to LASH's virtual layer requirements is related to ASI's lack of support for large networks.

We explained in Section 4.2.3 that, in order to ensure deadlock-freedom in a network based on ASI, the number of available VCs cannot be less than the number of virtual layers required by LASH. Moreover, the TC to VC mappings must be consistent with LASH's usage of layers. Corresponding requirements apply to the number of available VLs and to the SL to VL mappings of a network based on InfiniBand.

In order to provide service di erentiation when LASH is used with InfiniBand, a virtual layer to priority-class mapping similar to the one suggested for ASI (see Section 4.2.3) could be introduced. Then, at least two di erent priority levels could be provided for InfiniBand in the cases where LASH needs less than half of the available number of VLs.

LASH is a topology agnostic routing algorithm and does therefore inherently provide static fault tolerance. Sections 4.5.1 and 4.5.3 investigate how the performance of LASH is a ected by an increasing number of static switch faults in mesh and torus topologies. The results presented are relevant both for InfiniBand and ASI.

Section 4.2.5 describes a route optimization for mesh topologies that allows LASH to identify the same set of paths as the DOR algorithm identifies. This route optimization is equally relevant for InfiniBand and ASI.

4.4 Experiment setup

We have suggested LASH as an appropriate routing algorithm for ASI. In order to evaluate the performance of LASH in an ASI fabric, an ASI simulator model was built on top of the generic communication simulator described in Section 3.1.1. The ASI simulator model implements the main characteristics of ASI, such as its bu er organization and flow control, as well as its lack of support for transitions from one virtual layer to another. However, in order to obtain results that are of relevance also outside ASI, ASI's limitation on the length of a packet's routing structure (turn pool) of 31 bits is ignored. This enables simulation of medium and large sized topologies.

Our considerations regarding routing in ASI were subsequently found to be highly relevant for routing in InfiniBand, and we concluded that LASH is also a suitable routing algorithm for In-

finiBand (see Section 4.3). We explained above that our ASI simulator model implements ASI's bu er organization and flow control, and does not support transitions from one virtual layer to another. The particular bypass-capable VCs of ASI are not used in our experiments, and – like ASI – InfiniBand uses credit-based flow control. Furthermore, LASH does not require transitions from one virtual layer to another. Thus, we believe that the results of our evaluation of the performance of LASH in ASI are also representative of the performance of LASH in InfiniBand. The most important objection is perhaps that – as explained in Section 2.5 – for a distributed routing technology (such as InfiniBand) packets transmitted from di erent source processing nodes towards a common destination processing node are bound to share the same path from any common intermediate switch towards the destination, whereas for a source routing technology (such as ASI) the paths are independent. Nevertheless, when the optimization of LASH is used, the same set of paths is computed by a source routing technology and a distributed routing technology.

We have studied LASH's need for virtual layers under the assumption that a layer l_{sd} (from a source s to a destination d) and a layer l_{ds} (for the return path from d to s) can be chosen independently. Some technologies, such as implementations of InfiniBand, demand that $l_{sd} = l_{ds}$, and this requirement was also stated in the ASI specification. According to [132], when demanding $l_{sd} = l_{ds}$ an increase should be expected in LASH's layer-requirement, especially for topologies such as tori which consist of numbers of rings. We notice, however, that in the cases where LASH needs only a single layer for $l_{sd} \neq l_{ds}$ (such as when the optimization described in Section 4.2.5 is used in a mesh), a single layer also su ces for $l_{sd} = l_{ds}$.

We consider an 8×8 and 16×16 mesh topology, as well as an 8×8 torus topology. In addition to the fault-free case, four di erent static fault-scenarios are simulated, where 1, 2, 4 and 6 randomly selected switches are faulty at system start-up.[4] In the following we refer to a processing node connected to a non-faulty switch as *active*.

The two metrics used in this study quantify the average throughput per active processing node and the average packet latency, and are referred to as Thr_n and Lat, respectively. $Thr_n = \frac{packets}{m}$, where $packets$ are the total number of packets received over the data collection period of 100 000 cycles by the m active processing nodes in a system. $Lat = \frac{\sum Lat_{pkt}}{packets}$, where Lat_{pkt} is the latency for a single packet. That is, the time that elapses from when the packet is generated and injected into the transmission queue of the source processing node until the packet header is received by the destination processing node. Each experiment is repeated 30 times (with a di erent seed), and the mean values of these repetitions are presented together with their 95% confidence intervals.

We study two di erent tra c patterns – a uniform destination address distribution, and a hotspot tra c pattern where 50% of the packets are destined for a randomly selected hotspot node whereas the remaining 50% of the packets are uniformly distributed.

The transmission queue of a source processing node has space for 12 packets per VC, and overflows when the network cannot deliver packets at the rate they are injected. Both an ingress and egress bu er of a switch port can hold 6 packets per VC, and an ingress VC implements virtual output queuing.

An initial set of simulation experiments was performed in order to determine the maximum

[4] A test is included to ensure that the faults do not disconnect the topology.

numbers of virtual layers needed by LASH for the regular mesh and torus topologies as well as for the non-regular topologies that result from switch faults. The output of these experiments (which was also briefly summarized in Section 4.2.2) was used as input to the subsequent experiments.

The first of our main sets of experiments considers LASH's qualities as a static fault tolerance mechanism. The route optimization suggested in Section 4.2.5 is not applied, and LASH randomly selects one path from the set of possible shortest paths from a source to a destination. For these experiments, at most 4 virtual layers are required for an 8×8 and 16×16 mesh, whereas for an 8×8 torus at most 6 virtual layers are required. These numbers of layers are at the disposal of LASH for all levels of switch faults (including the fault-free case).

The second set of experiments compares LASH to DOR. In these experiments, the same number of VCs as required by LASH is also made available for DOR. Without the optimization described in Section 4.2.5, LASH needs 2 virtual layers in the mesh topologies. When the optimization is applied, only a single virtual layer is needed in the fault-free case. However, as indicated in Section 4.2.5, a single virtual layer may no longer be su cient in the presence of faults. The third set of experiments evaluates LASH's qualities as a static fault tolerance mechanism when route optimization is applied. In this case, our experiments suggest that 2 virtual layers are su cient for the levels of switch faults under study.

4.5 Results

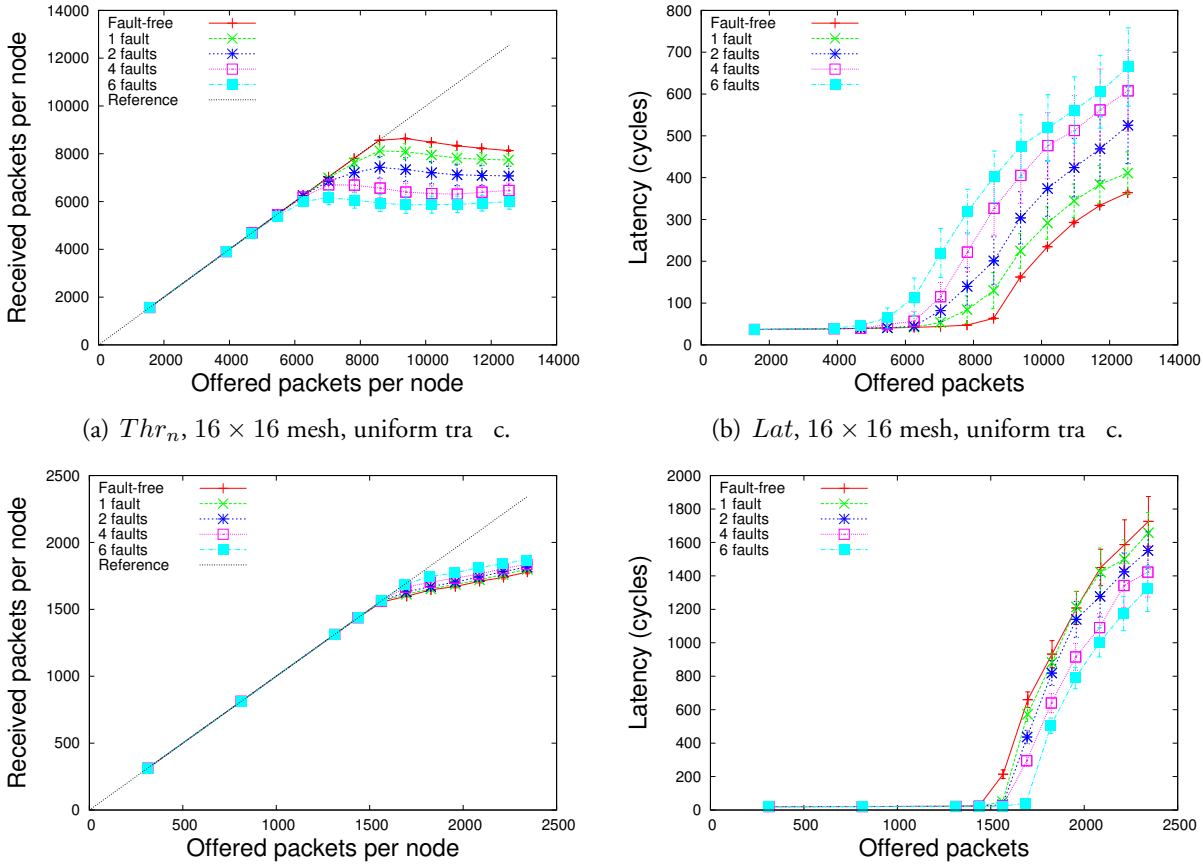
Being a topology agnostic routing algorithm, LASH inherently supports static fault tolerance. In Section 4.5.1 we present a study on how the performance of LASH is a ected by an increasing number of static switch faults in an initially regular mesh or torus topology. Moreover, Section 4.5.2 evaluates the optimization of LASH proposed for mesh topologies in Section 4.2.5, which allows the paths computed by LASH to comply with the paths computed by DOR. Finally, Section 4.5.3 considers how this optimization a ects some fault tolerance properties of LASH. We cannot show the results from all of our experiments, but have selected and included a set of representative plots.

4.5.1 Static fault tolerance

In this section, we present a study of how the performance of LASH is a ected by an increasing number of switch faults in an originally fault-free mesh or torus topology. We consider the fault-free case, as well as 1, 2, 4 and 6 randomly selected switch faults. We apply a set of tra c load levels, ranging from an insignificant load up to well above network saturation.

For a uniform tra c pattern, Figures 4.3(a) and 4.3(b) show Thr_n and Lat, respectively, for a 16×16 mesh. Figure 4.3(a) shows that, as the number of failing switches increases, the knee-point (saturation point) of the Thr_n curve gradually occurs at a lower tra c load level (per active processing node), and we observe a virtually proportional decrease in the maximum reached Thr_n. Figure 4.3(b) shows a corresponding development for Lat, which, above the knee-point, gradually increases as additional switches become faulty. Thus, the observed performance degradation is graceful.

For hotspot tra c the results are di erent. Figure 4.3(c) shows that Thr_n increases slightly as a higher number of switches fail in an originally 8×8 mesh. Likewise, Figure 4.3(d) demonstrates a decrease in Lat as switches disappear.



(a) Thr_n, 16×16 mesh, uniform tra c.

(b) Lat, 16×16 mesh, uniform tra c.

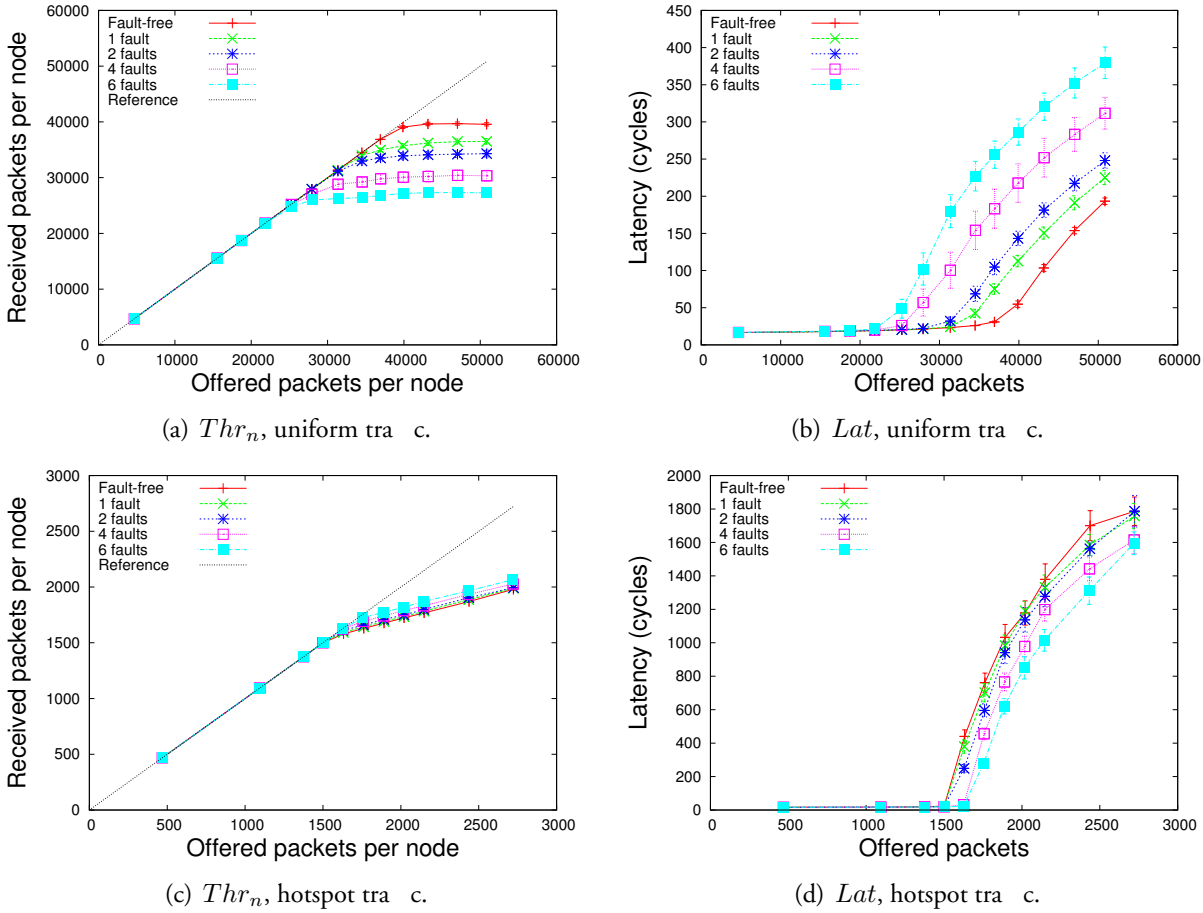
(c) Thr_n, 8×8 mesh, hotspot tra c.

(d) Lat, 8×8 mesh, hotspot tra c.

Figure 4.3: Using LASH: E ects of an increasing number of switch faults for mesh topologies. 4 virtual layers are used.

Figure 4.4 confirms that, both for uniform and hotspot tra c, the results for an 8×8 torus are similar to the results for the mesh topologies. In the following, we attempt to explain the di erences in the results observed for the hotspot and uniform tra c patterns.

A comparison of Figure 4.4(a) with Figure 4.4(c) illustrates that saturation occurs at a significantly lower tra c load level for hotspot tra c than for uniform tra c. This is expected as, for the hotspot tra c pattern, 50% of the packets are directed towards a single destination, which may cause significant congestion already for a relatively low tra c load. For the uniform tra c pattern, on the other hand, the packets are evenly distributed towards all destinations, and a higher tra c load is sustainable. When switches disappear, the total capacity of the interconnection network is reduced, although the actual reduction depends on the location of the faults. Thus, for uniform tra c, the observed (graceful) performance degradation is reasonable. Also, when switches fail, the processing nodes attached to them become inactive, resulting in a decreased tra c load in the network. This may alleviate congestion which, particularly for a hotspot tra c pattern, is an impending risk. For hotspot tra c, the positive e ects of alleviated congestion resulting from switch faults seem to compensate for the negative e ects of reduced network capacity. This explains the increase in performance observed for hotspot tra c as additional switches fail.



(a) Thr_n, uniform tra c.

(b) Lat, uniform tra c.

(c) Thr_n, hotspot tra c.

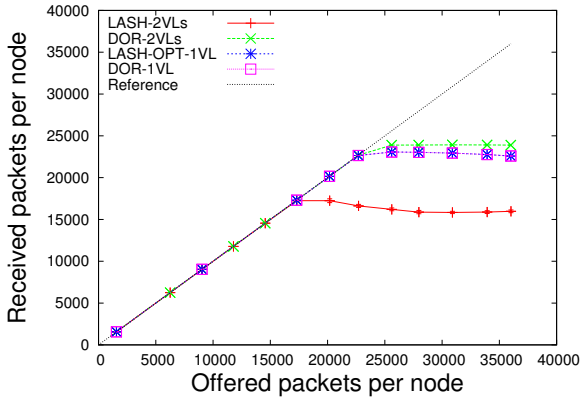
(d) Lat, hotspot tra c.

Figure 4.4: Using LASH: E ects of an increasing number of switch faults for an 8×8 torus. 6 virtual layers are used.

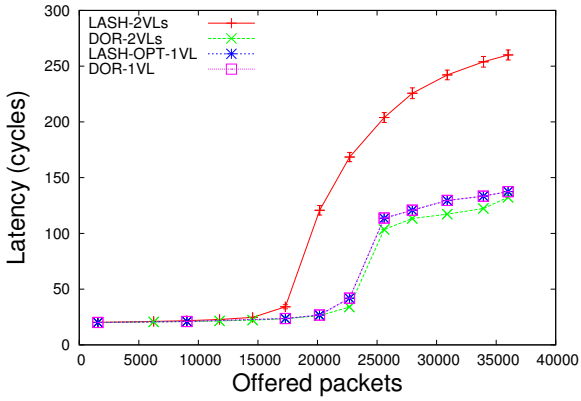
4.5.2 Route optimization

In this section, we compare the performance of LASH – with and without the route optimization described in Section 4.2.5 – with the performance of DOR. First, we let LASH randomly select one path from a set of possible paths from a source to a destination. In this case two virtual layers are needed, and we refer to LASH as "LASH-2VLs". Second, the route optimization is applied for LASH. Then, only a single virtual layer is needed, and for this scenario we refer to LASH as "LASH-OPT-1VL". Likewise, we refer to DOR, when granted two virtual layers, as "DOR-2VLs" and, when granted only a single layer, as "DOR-1VL".

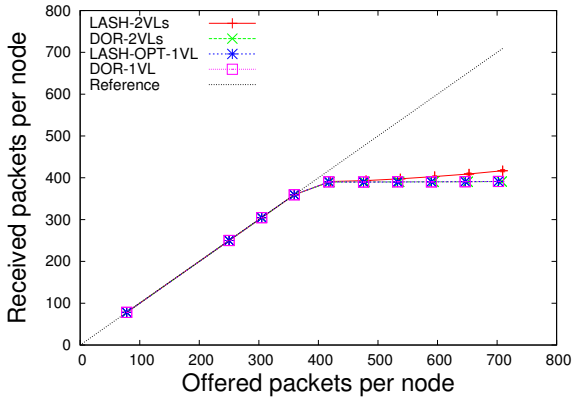
For the uniform tra c pattern, Figures 4.5(a) and 4.5(b) show Thr_n and Lat, respectively, for LASH-2VLs, DOR-2VLs, LASH-OPT-1VL and DOR-1VL in a fault-free 8×8 mesh topology. We observe how DOR-2VLs achieves significantly higher Thr_n and significantly lower Lat than LASH-2VLs does (remember that route optimization is not applied for LASH). This is not surprising, as DOR is known to achieve very good performance for uniformly distributed tra c. The route optimization allows LASH to select the exact same set of paths as DOR uses – the XY-paths in these experiments. Thus, as demonstrated in Figures 4.5(a) and 4.5(b), LASH-OPT-1VL achieves the same performance as DOR-1VL achieves for uniformly distributed tra c. We observe how the performance of LASH-OPT-1VL and DOR-1VL is slightly inferior to that of DOR-2VL. The reason is that, when compared to using only a single virtual layer, the use of two virtual layers re-



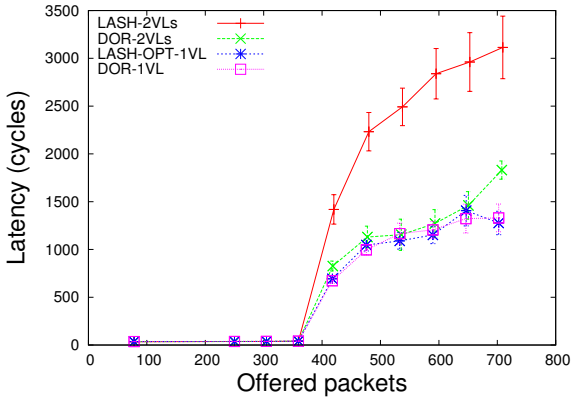
(a) Thr_n, 8×8 mesh, uniform tra c.



(b) Lat, 8×8 mesh, uniform tra c.



(c) Thr_n, 16×16 mesh, hotspot tra c.



(d) Lat, 16×16 mesh, hotspot tra c.

Figure 4.5: A comparison of LASH with DOR. LASH is shown with route optimization (*LASH-OPT-1VL*) and without route optimization (*LASH-2VLs*), for which 1 and 2 virtual layers are needed, respectively.

duces head-of-line blocking within each virtual layer. With the optimization, a single virtual layer is su cient for LASH to ensure deadlock-freedom in a fault-free mesh topology. Nevertheless, more virtual layers may be granted in order to boost performance.

For the hotspot tra c pattern, Figures 4.5(c) and 4.5(d) show Thr_n and Lat, respectively, for LASH-2VLs, DOR-2VLs, LASH-OPT-1VL and DOR-1VL in a fault-free 16×16 mesh topology. We observe that Thr_n achieved by LASH-2VLs and DOR-2VLs are similar (although slightly higher for LASH-2VLs for the very highest tra c load levels). On the other hand, Lat is significantly higher for LASH-2VLs than for DOR-2VLs for tra c load levels above network saturation. This suggests that the route optimization for LASH is advantageous also for other tra c patterns than for uniformly distributed tra c. When comparing LASH-OPT-1VL and DOR-1VL with DOR-2VLs, we observe no di erence in Thr_n, but perhaps even a slightly lower Lat for the two former. Thus, in this case, an additional virtual layer does not improve performance. This suggests that head-of-line blocking within each virtual layer becomes an insignificant factor compared with the heavy network congestion resulting from the hotspot tra c pattern itself.

4.5.3 Static fault tolerance when using route optimization

Section 4.5.1 demonstrated that LASH is able to provide static fault tolerance in mesh and torus topologies, and that a graceful performance degradation results from an increasing number of switch faults. For fault-free mesh topologies, Section 4.5.2 showed that, by using the route optimization suggested in Section 4.2.5, LASH can provide the same set of paths as DOR, and thereby achieve the same performance. When compared to DOR, a significant advantage of LASH is that, even when the optimization is applied, it is usable – without special precautions – for a non-regular topology that may result from network component faults in a mesh topology.[5] In this section, we study how the performance of LASH with route optimization is a ected by an increasing number (1, 2, 4, 6) of switch faults.

For uniform tra c, Figures 4.6(a) and 4.6(b) show Thr_n and Lat, respectively, for an 8×8 mesh topology. When compared to the fault-free case, a significant performance degradation is observed when a single switch fault has occurred, both for Thr_n and Lat. In particular, the knee-point – which indicates network saturation – occurs at half the tra c load (per active processing node). However, as additional switch faults occur, the performance degradation is more gradual.

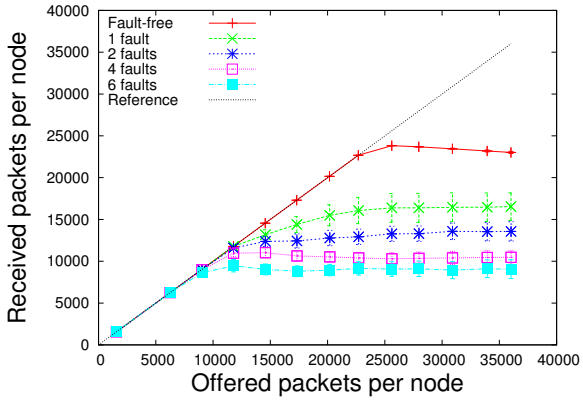
For hotspot tra c, Figures 4.6(c) and 4.6(d) show Thr_n and Lat, respectively, for a 16×16 mesh topology. For Thr_n, the di erences between the fault-free case and any of the fault-scenarios are negligible. Lat, on the other hand, tends to increase as more switches fail. (For tra c load levels well above saturation, relatively large variations can be observed, however.) Recall from Section 4.5.1 that, for the hotspot tra c pattern, there is a tradeo (with respect to performance) between the reduced network capacity and the alleviated congestion that result from switch faults. For the 16×16 mesh, Figure 4.6(d) indicates that reduced network capacity is the dominant factor when LASH's route optimization is applied. For the 8×8 mesh, on the other hand, alleviated congestion seems to be more dominant (see Figure 4.7(b)).

In the rest of this section, we refer to LASH with route optimization as "optimized LASH" and LASH without route optimization as "original LASH". For an 8×8 mesh topology, Figure 4.7 compares Thr_n for the optimized LASH with Thr_n for the original LASH, both for the fault-free case and in the presence of 1, 4 and 6 switch faults.[6] As previously stated, the original LASH needs at most 4 virtual layers for these scenarios. Thus, in order to get a fair comparison, the optimized LASH is also granted 4 virtual layers for the experiments presented here.

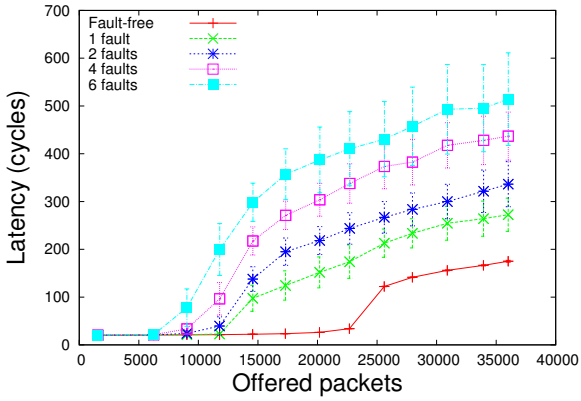
Figure 4.7(a) shows the results for the uniform tra c pattern. As we already know from Section 4.5.2, the optimized LASH achieves a significantly higher performance than the original LASH achieves in the fault-free case. However, already for a single switch fault, the saturation point occurs at a somewhat higher tra c load level (per active processing node) for the original LASH than for the optimized LASH. (Although for even higher load levels, the Thr_n curve for the optimized LASH seems to grow past the Thr_n curve for the original LASH.) In the cases of 4 and 6 switch faults, Thr_n for the original LASH stabilizes at a higher level than Thr_n for the optimized LASH, although the di erence is not dramatic. Thus, for systems where a tra c pattern resembling the uniform pattern is expected, the decision of whether to apply route optimization for LASH involves a tradeo between a higher performance in the fault-free case and a reduced performance in the presence of switch faults. However, the advantage of the route optimization in the fault-free

[5] We assume that the topology remains physically connected.

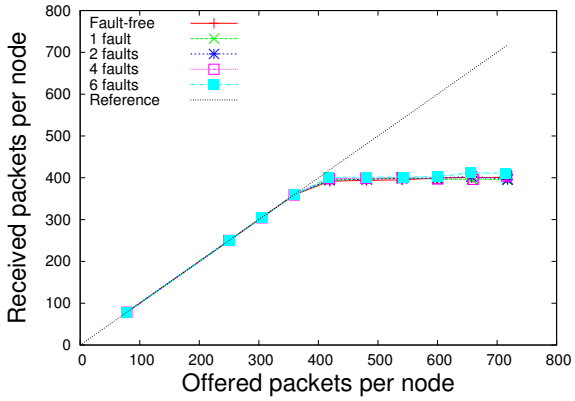
[6] For improved clarity, the curve representing 2 switch faults (which did not add further information) was not included in the plots.



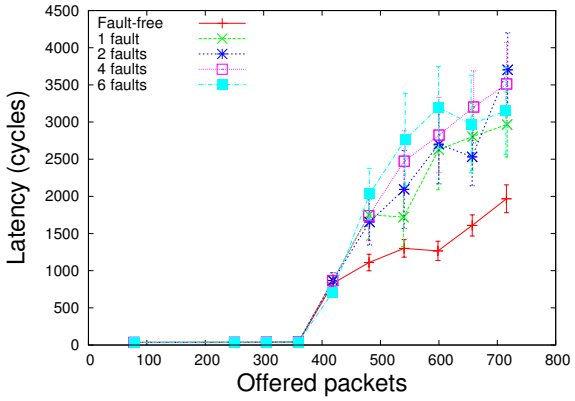
(a) Thr_n, 8×8 mesh, uniform tra c.



(b) Lat, 8×8 mesh, uniform tra c.



(c) Thr_n, 16×16 mesh, hotspot tra c.



(d) Lat, 16×16 mesh, hotspot tra c.

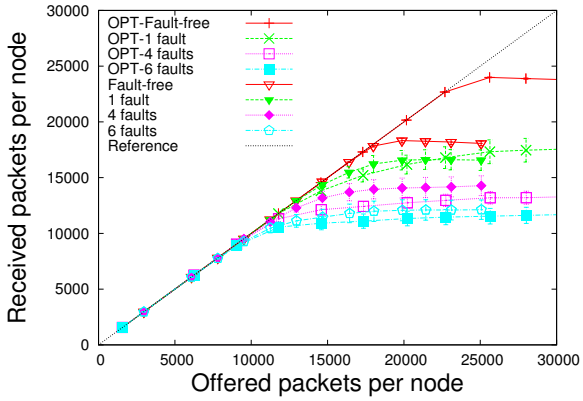
Figure 4.6: Using LASH with route optimization: E ects of an increasing number of switch faults for mesh topologies. 2 virtual layers are used.

case appears to be more significant than the disadvantage in the case of switch faults.

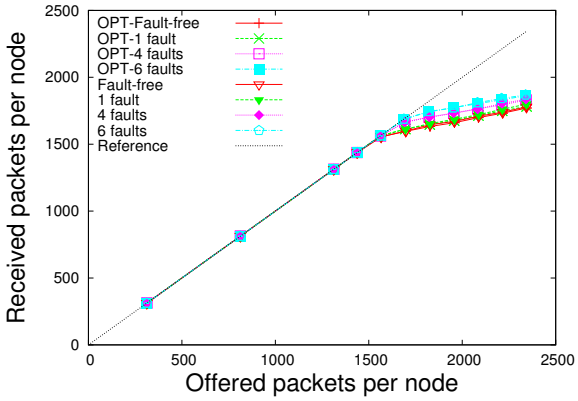
Figure 4.7(b) shows the results for the hotspot tra c pattern. As was also observed in Section 4.5.1, Thr_n increases as switch faults occur (due to alleviated congestion when fewer processing nodes generate tra c). Contrary to the observations for uniform tra c, the di erences in Thr_n between the original and optimized versions of LASH are negligible for hotspot tra c.

4.6 Related work and our contribution

The ASI specification [17] did not prescribe a particular routing algorithm. Thus, guidelines were needed in order to select an appropriate routing algorithm for this – at the time our study was conducted – new and promising interconnection network technology. We stated a number of criteria that a routing algorithm for ASI should fulfil. These are presented in Section 4.1, where we argue that a routing algorithm for ASI should support deterministic routing, be topology agnostic and deadlock-free; provide shortest paths and high e ciency for regular topologies; and not require transitions from one virtual layer to another. Based on these criteria, we evaluated a number of routing algorithms and finally recommended one algorithm – LASH, which fulfils all of the criteria – to be used with ASI. Section 4.1 gives reason for why each of the other algorithms fall short of the requirements that a routing algorithm for ASI should fulfil.



(a) Thr_n, uniform tra c.



(b) Thr_n, hotspot tra c.

Figure 4.7: A comparison of LASH with route optimization (prefixed *"OPT-"*) and without route optimization (no prefix) for an 8×8 mesh. 4 virtual layers are used.

We are not aware of any other studies than our own that have performed a systematic evaluation of routing algorithms in order to find an appropriate routing algorithm for ASI. In [236] it was stated that FRoots, a fault-tolerant routing mechanism based on Up*/Down* routing, is applicable for ASI. We do not disagree that FRoots can be used with ASI. However, as it is tied to Up*/Down* routing, FRoots does not meet the criteria stated in Section 4.1.3 that shortest path routing should be supported.

Some studies of congestion control (see e.g. [66, 78, 121]) and quality of service (see e.g. [149, 184]) for ASI are available in the literature. Although few studies have directly focused on e - cient routing in ASI, some of the papers that target ASI fabric management also consider routing. Management procedures for assimilating topological changes in ASI fabrics are the main topics of [188, 189]. These studies assume that Up*/Down* routing (which does not meet the criteria stated in Section 4.1.3) is used. In [188] all paths are calculated by a centralized entity (called *Fabric Manager* in ASI). In order to reduce path computation time, [189] proposes a distributed approach, where each processing node computes the set of paths for which it is the source. We notice that such a distributed path calculation is more suitable for Up*/Down* than for LASH. For Up*/Down*, a directed acyclic graph representing the topology and turn-restrictions can be disseminated to all the processing nodes (a new data structure must be defined for ASI, however). For LASH, on the other hand, a distributed approach would complicate the assignment of paths to virtual layers.

The performance of LASH in comparison to the performance of alternative routing algorithms has been assessed by previous studies. Moreover, some general properties of LASH, such as its computational complexity and need for layers, have also been investigated. In addition, realization of LASH in Ethernet has been discussed. The deterministic LASH algorithm was first proposed in [218], where the need for virtual layers was evaluated. LASH was further elaborated on in [144, 234], where di erent variants of LASH (also supporting multiple paths and adaptivity) were presented and their performance compared to existing routing algorithms such as Up*/Down*, MRoots and Escape Routing [213]. Regarding the complexity of LASH, the most explicit assessment and discussion are found in [144]. LASH has been proposed as a routing algorithm for the Gigabit Ethernet technology [206]. In [183], a modification of Ethernet's flow control mecha-

nism is proposed in order to support the use of LASH. Furthermore, the performance of LASH is compared to (and found significantly better than) the performance of Ethernet's spanning tree protocol [207] and the Tree Based Turn Prohibition algorithm [174].

Previous studies [144, 218, 234] mainly focused on assessing LASH's requirements of virtual layers for irregular topologies. Our study, as well as [132][7], considered LASH's requirements of virtual layers for regular topologies such as fat-trees, meshes and tori. (For the latter two, faulty network components were also taken into account.)

Our study discusses main principles regarding realization of LASH in ASI, without touching upon implementation details. Virtual layers are relatively straightforwardly realized by utilizing the TC to VC mapping supported by ASI. However, when more than one virtual layer is needed by LASH for deadlock avoidance, a service di erentiation mechanism depending on TCs cannot be used as originally intended. For such cases, we propose a novel layer to priority-class mapping which, given a su cient number of available VCs, supports service di erentiation. In [149], a framework for quality of service provisioning in ASI is proposed, covering such issues as tra c classification (according to bandwidth, latency and jitter), scheduling and admission control. However, in [149] ASI's original TC to VC mapping is assumed.[8]

We study the performance of LASH as a static fault tolerance mechanism, a quality of LASH that has not previously been thoroughly investigated. We show that LASH supports a graceful degradation of performance as an increasing number of switches fail. The focus of [132], which followed our studies, was on the e ect of communication channel failures on LASH's requirements of virtual layers in a two-dimensional torus topology.

In a mesh topology, for uniformly distributed tra c in particular, the original version of LASH could not compete with DOR with respect to performance. We proposed an optimization which allows LASH to select the same paths, and thus achieve the same performance, as DOR in a fault-free mesh topology. (A single virtual layer is then su cient for LASH). Moreover, with the optimization, LASH's abilities as a static fault tolerance mechanism are maintained. Our findings regarding the static fault tolerance abilities and the route optimization possibilities of LASH are also relevant for usage of LASH outside ASI.

The route optimization of LASH for mesh topologies was suggested by us in [224]. Later, the same principle was applied to quadratic two-dimensional torus topologies in [132]. For such k-ary 2-cube topologies, [132] shows that only 3 virtual layers are needed for any $k > 3$. Recall from Section 2.5.1 that the use of DOR in torus topologies depends on an additional mechanism, such as VCs (see [55]) or Bubble flow control (see [35]), for the avoidance of deadlock. For interconnection network technologies that allow transitions from one virtual layer to another, DOR would only need 2 VCs in order to ensure deadlock-freedom for k-ary 2-cubes where $k > 3$. However, as ASI does not support such transitions, 3 VCs are the minimum requirement for DOR as well as for LASH.

Other topology agnostic routing algorithms can also calculate paths that correspond to the paths calculated by DOR in a mesh topology. We remember from Section 2.8.1 that Flexible Routing (FX) [197] improves the Up*/Down* algorithm by permitting a cycle of channel dependencies to be broken in di erent places for the clockwise and counterclockwise directions. In [197], this allowed FX to identify the DOR (XY) paths in a two-dimensional mesh. However, FX cannot

[7] A Master's thesis for which the author of this PhD thesis was one of the supervisors.

[8] Tra c classes are called service classes in [149].

guarantee shortest path routing in arbitrary topologies, and does therefore not meet the requirement of a routing algorithm for ASI stated in Section 4.1.3. Like LASH, Descending Layers [119], Transition-Oriented Routing (TOR) [200] and LASH-TOR [217] can also calculate any set of paths, including a set of paths corresponding to DOR for a mesh topology. However, Descending Layers and TOR generally implement VC-transitions also for the XY or YX paths in a mesh (depending on their underlying directed acyclic graphs). As previously explained, since these algorithms depend on VC-transitions for deadlock avoidance, they do not comply with the requirement of a routing algorithm for ASI stated in Section 4.1.4.

High-level discussions on the suitability of LASH as a routing algorithm for InfiniBand [105] were included in [144, 218, 234]. (In addition, [144] briefly mentioned a potential use of LASH in ASI.) After our studies of routing in ASI were performed, our considerations were found also to be highly relevant for routing in InfiniBand. The requirements stated for a routing algorithm to be used with ASI also provide directions for selecting a routing algorithm to be used with InfiniBand (see Section 4.3.1). Furthermore, various issues related to realization of LASH in ASI are also relevant for realization of LASH in InfiniBand (see Section 4.3.2). In addition, as explained in Section 4.4, the results of our evaluation of the performance of LASH in ASI are also representative of the performance of LASH in InfiniBand. Our considerations were taken into account when LASH was included into the OpenFabrics Enterprise Distribution [163] software stack for InfiniBand. Practical experience has confirmed our conclusion that LASH is also a suitable routing algorithm for InfiniBand.

A recently proposed topology agnostic routing algorithm called Deadlock-free single-source-shortest-path (DFSSSP) [62] is also suitable for InfiniBand. DFSSSP utilizes the principles of LASH on the purpose to realize deadlock-freedom for the previously proposed Single-source-shortest-path (SSSP) routing algorithm [99] which aims to optimize load balancing. It is proven in [62] that for an arbitrary network topology the problem of deciding the minimum number of virtual layers that ensures deadlock-freedom is NP-complete [79]. DFSSSP introduces new heuristics to distribute a set of shortest paths across a set of virtual layers in such a way that no virtual layer includes a cyclic channel dependency. The virtual layers are processed in turn, and every cyclic channel dependency is broken by selecting one of the channel dependencies that forms the cycle for removal. (All the paths that give rise to the selected channel dependency are relocated to the next higher virtual layer.) Results presented in [62] show that, when compared to LASH, DFSSSP reduces virtual layer requirements as well as computation time. DFSSSP increases the demand for memory space, however.

Another topology agnostic routing algorithm that builds upon the principles of LASH is InFIR [208] which handles a single fault dynamically and supports shortest paths. InFIR is partly based on an algorithm that targets the Internet (where packet deadlock is generally not an issue), and the principles of LASH were applied to ensure deadlock-freedom in an interconnection network environment.

4.7 Critique

When our studies were initiated in 2005, ASI was a promising interconnection network technology with support from a significant part of the high performance computing industry. Contrary to expectations, ASI did not become widespread, and the specification process was discontinued.

Nevertheless, the ASI technology is now incorporated in Dolphin Express, and is therefore still of interest. Furthermore, as previously explained, our considerations regarding routing in ASI also turned out to be relevant for routing in InfiniBand. LASH is now available in the OpenFabrics Enterprise Distribution software stack for InfiniBand, where it has proved to be a useful routing algorithm.

Apart from a relatively high computational complexity, the main drawback of LASH is, in our opinion, that its requirements of virtual layers are not fully predictable. According to [218,234], the theoretical upper bound on the number of virtual layers required by LASH is as high as $\frac{N}{2}$ for an interconnection network of N switches, although experiments indicate that a moderate number of virtual layers are normally su cient [144, 218, 234]. The number of virtual layers required mainly depends on the topology, size and connectivity of an interconnection network. The requirement of virtual layers also depends on which path is selected (from a set of possible paths), and on the sequence that the selected paths are arranged into the virtual layers. In addition, for faulty regular topologies, the number of virtual layers needed by LASH depends on the location of the failed switches and communication channels. The DFSSSP algorithm (which was recently presented in [62]) builds upon main principles of LASH, but introduces new heuristics that reduce virtual layer requirements as well as computation time.

4.8 Future work

Although a number of studies have focused on LASH (see e.g. [144, 183, 218]), some issues are still open. The perhaps most important remaining challenge concerns LASH's usage of virtual layers. Improved predictability is desirable – both with respect to the number of virtual layers required, and with respect to the particular virtual layer that a path between a source s and a destination d is placed in. The latter may be an issue of particular importance in the case of a network reconfiguration. A reconfiguration could cause a change of virtual layer for a path between s and d. When using the OpenFabrics Enterprise Distribution for InfiniBand, for example, such a change of virtual layer in general necessitates a restart of an application that runs on a set of processing nodes which includes s and d. (Unless the mechanism recently proposed in [90] – and made more e cient in [89] – is used. This mechanism, which is implemented as a research prototype, supports a dynamic update of the data structures in the processing nodes – including the data structures holding SL values – and is transparent to running applications.)

An important objective for future research is to provide some kind of guarantee on LASH's usage of virtual layers, both for regular topologies such as tori and – even more challenging – for arbitrary topologies. Recall that [62] proves that for an arbitrary topology the problem of deciding the minimum number of virtual layers that ensures deadlock-freedom is NP-complete [79]. Also recall that the DFSSSP routing algorithm [62], which builds upon main principles of LASH, introduces new heuristics to distribute a set of shortest paths across a set of virtual layers in such a way that no virtual layer includes a cyclic channel dependency. When compared to LASH, the new heuristics introduced by DFSSSP presuppose storage of more information (concerning which paths that give rise to which channel dependencies). The approach taken by DFSSSP should be further studied in order to assess its usefulness as a means to improve the predictability of LASH's usage of virtual layers.

Chapter 5

Reconfiguration

Replacement of an interconnection network's routing function sometimes becomes necessary, for instance in a case where the routing function has been disconnected due to a topology change. Whether such a topology change is planned or unplanned is unimportant in this thesis. Recall from Section 2.6 that a number of network management functions (fault detection, topology discovery, path computation and reconfiguration) are involved in order to restore a connected routing function. This chapter focuses on the final step of the restoration process – the *reconfiguration*.

A reconfiguration comprises the change-over from one routing function to another, an operation which, as explained in Section 2.6, is prone to deadlock. Some reconfiguration strategies include a rigid deadlock avoidance mechanism which causes performance degradation, such as increased latency and decreased throughput, during the routing change-over. For the provisioning of a predictable network service, an e cient reconfiguration strategy is crucial. Such a reconfiguration strategy minimizes the performance-penalties imposed on a running application during the transition from one routing function to another. The research presented in this chapter aims at improving the availability of e cient reconfiguration strategies.

In Section 5.1, we expand the area of application for one of the most versatile and e cient reconfiguration strategies available in the literature – Overlapping Reconfiguration (OR) [139,140]. Previously, OR was only applicable for distributed routing systems. We propose an adaptation of OR in order to enable using it also in source routing systems. For source routing systems, we also propose an optimization of the OR algorithm in order to improve the network service o ered to running applications during a reconfiguration. Furthermore, we study how di erent degrees of synchronization of the start of a reconfiguration process a ect the performance of OR. Section 5.1 is based on [221].

In Section 5.2, we present a new reconfiguration strategy, RecTOR, which ensures deadlock-freedom during a reconfiguration process without causing performance degradation. RecTOR is applicable for both source and distributed routing systems, is based on a simple concept, is easy to implement, and assumes Transition-Oriented Routing (TOR) [200] which supports excellent performance. Section 5.2 is based on [222].

Both of the reconfiguration strategies focused in this chapter allow application tra c into the network during the reconfiguration, and thus fall within the category of dynamic reconfiguration.[1] Throughout this chapter, we refer to the routing functions in use before and after a reconfiguration

[1] Although in [139], OR was categorized as a static reconfiguration scheme with overlapping phases.

process as R_{old} and R_{new}, respectively. Each of the routing functions is assumed to be deadlock-free. Furthermore, we refer to packets that exclusively follow either R_{old} or R_{new} as packets$_{\text{old}}$ or packets$_{\text{new}}$, respectively.

Following the presentation of our research, Section 5.3 discusses related work and our contribution, Section 5.4 includes critical comments regarding our work, and Section 5.5 presents issues for further study.

5.1 OR for source routing environments

OR [139,140] is probably the most versatile of the reconfiguration strategies found in the literature. It ensures in-order packet delivery, does not require virtual channels (VCs), nor is it designed for a particular topology, technology or pair of routing algorithms. OR is the only reconfiguration scheme for lossless networks that possesses all of these qualities.

Previously, OR was only available for systems that apply distributed routing. The design of a general[2] reconfiguration scheme is usually more complicated for a source routing technology (such as Dolphin Express [122]) than for a distributed routing technology (such as InfiniBand [105]), as a distributed routing environment is more flexible with respect to packet routing. In a source routing system, the path of a packet is decided and included in the packet's header by the source processing node, and can thus not be changed by an intermediate switch. In a distributed routing system, on the other hand, each of the intermediate switches traversed from a source to destination processing node decides the next hop of a packet's path.

We propose an adaptation of the OR algorithm in order to expand its area of application to include source routing environments. Section 5.1.1 first describes the original OR algorithm as designed for distributed routing systems, whereas Section 5.1.2 presents the adaptations which enable use in source routing systems. Furthermore, targeting source routing systems, Section 5.1.3 introduces an optimization of the OR algorithm that aims to reduce the performance penalties experienced by running applications during a reconfiguration. The performance impact of this optimization is evaluated in Sections 5.1.5 and 5.1.6.

OR uses a special packet called a *token* to control that deadlock does not occur during the transition from one routing function to another. Section 5.1.4 discusses how the injection of tokens by di erent tra c sources may be more or less synchronized, and Sections 5.1.5 and 5.1.6 evaluate how performance is a ected by di erent degrees of synchronization.

5.1.1 The original OR algorithm

Before we propose adaptations of OR in order to enable using it in source routing systems, this section describes the essence of the original OR algorithm [139, 140] which was proposed for distributed routing systems.

OR strictly requires that a packet is routed the entire path from its source to its destination according to only one routing function. We already mentioned that both R_{old} and R_{new} are by themselves deadlock-free, and that OR uses a special packet called a *token* to prevent that deadlock occurs during the transition from R_{old} to R_{new}. The task of the token is to separate packets$_{\text{old}}$

[2] We refer to a reconfiguration scheme which is not developed for a particular interconnection network technology as a *general* reconfiguration scheme.

from packets$_{\text{new}}$ on each (physical or virtual) communication channel, such that each channel first transmits packets$_{\text{old}}$, then the token, and finally packets$_{\text{new}}$. This means that tokens must propagate through an interconnection network in accordance with the channel dependency graph of R_{old}. The core of the OR algorithm is the following set of rules which governs the propagation of tokens through the network. These rules control the behaviour of injection channels (communication channels from source processing nodes to switches – the points where packets are injected into the interconnection network); input channels of switches; and output channels of switches:

- Each *injection channel* inserts a token to indicate the boundary between the last packet$_{\text{old}}$ and the first packet$_{\text{new}}$ originating from this channel.
- A *switch input channel* routes packets according to R_{old} until the token is processed, and thereafter routes packets according to R_{new}. After having processed the token, the input channel may dispatch packets solely to output channels which have already transmitted their token. Packets bound for output channels that have not yet transmitted the token are temporarily held back.
- *A switch output channel*, c_o, must not transmit the token until all input channels, c_i, for which channel dependencies[3] exist according to R_{old} from c_i to c_o, have processed the token. If no such channel dependency exists for an output channel, it could transmit the token as soon as a token has been processed by any of the switch's input channels.

5.1.2 Adaptations for use in source routing systems

In this section, we propose adaptations of the OR algorithm in order to enable using it in systems that apply source routing. In [139, 140], proofs are presented that OR provides deadlock-free reconfiguration, and the central statements in [140] are included in Lemma 1, Lemma 2 and Corollary 1. With adaptation of OR to a source routing environment in mind, we extract two invariants as fundamental for ensuring deadlock-freedom during the reconfiguration process. Invariants A and B below correspond to Lemma 2 and Corollary 1, respectively, of [140]. Given that Invariants A and B are obeyed by the OR algorithm adapted for source routing environments, deadlock-freedom follows from Lemma 1[4], Lemma 2, and Corollary 1 of [140].

Invariant A A packet must be routed from its source to its destination according to only one of the routing functions (either R_{old} or R_{new}).

Invariant B Any communication channel must initially transmit packets$_{\text{old}}$, thereafter the token, and finally packets$_{\text{new}}$.

For the practical adaptation of OR to source routing environments, let us return to the set of rules summarized in the three bullet points of Section 5.1.1. These constitute the core of the OR algorithm, and regulate the behaviour of injection channels from tra c sources as well as

[3] Remember from Section 2.3 that if a packet may use a channel c_b immediately after a channel c_a there is a channel dependency from c_a to c_b.

[4] Lemma 1 of [140] states that – under the assumption that all packets follow only one of the routing functions (either R_{old} or R_{new}) – some deadlocked set of packets resulting from a reconfiguration from R_{old} to R_{new} must include packets$_{\text{old}}$ permanently blocked by packets$_{\text{new}}$.

the behaviour of input and output channels of switches. In the following we consider each of these bullet points, and, where necessary, recommend adaptations that enable use of OR in source routing environments. In order to guarantee deadlock-free reconfiguration processes, it is essential that the resulting algorithm complies with Invariants A and B above.

We immediately observe that, for a source routing system, Invariant A is trivially fulfilled. A source routed packet will follow either R_{old} or R_{new} as its entire path is included in the header. Thus, in the following the focus will be on compliance with Invariant B.

- Each *injection channel* inserts a token to indicate the boundary between the last $\text{packet}_{\text{old}}$ and the first $\text{packet}_{\text{new}}$ originating from this channel.
  - For compliance with Invariant B, which states that a token must separate $\text{packets}_{\text{old}}$ from $\text{packets}_{\text{new}}$ on each communication channel, this requirement must also be adopted by a source routing system. The realization is straightforward.
- A *switch input channel* routes packets according to R_{old} until the token is processed, and thereafter routes packets according to R_{new}.
  - For a source routing system this requirement is redundant. It relates to Invariant A, which is trivially fulfilled as explained above.

  After having processed the token, the input channel may dispatch packets solely to output channels which have already transmitted their token. Packets bound for output channels that have not yet transmitted the token are temporarily held back.
  - For compliance with Invariant B, this requirement must also be adopted by a source routing system. The realization is straightforward.
- *A switch output channel*, c_o, must not transmit the token until all input channels, c_i, for which channel dependencies exist according to R_{old} from c_i to c_o, have processed the token. If no such channel dependency exists for an output channel, it could transmit the token as soon as a token has been processed by any of the switch's input channels.
  - For compliance with Invariant B, this requirement must also be adopted by a source routing system. The realization of this requirement becomes the main challenge in the adaptation of OR to a source routing environment. In order to decide when the token may be transmitted on an output channel each switch must know the dependencies from its input channels to its output channels. For distributed routing switches such information could be extracted from the routing tables. Below we propose how source routing switches (which do not hold routing tables) can obtain channel dependency information.

Channel dependency aware switches

We suggest a new procedure that allows a source routing switch to acquire channel dependency information while forwarding data packets. This procedure enables utilization of OR in source routing systems. It is useful for source routing systems that depend on distributed route calculation, and also for centralized source routing systems where uploading channel dependency information

to the switches is not feasible (e.g. if an appropriate packet format for such communication with the switches has not been defined).

On the other hand, if a source routing system uses a central unit for route calculation, and if extraction and upload of channel dependency information to the switches is possible, the procedure may not be needed. Then, the central unit – which knows the entire routing function – could simply assemble the channel dependency information relevant for each individual switch and upload it.

The channel dependency acquisition procedure is as follows:

- When a data packet is forwarded from an input channel c_{ix} to an output channel c_{oy}, register the channel dependency from c_{ix} to c_{oy} (if not already registered). Registration of channel dependencies should commence at system startup. During reconfiguration an output channel should reset its records and restart registration as soon as the token has been transmitted. In each switch, a routing and arbitration unit (see Figure 2.2) could be responsible for registering and maintaining the channel dependencies (which typically will be held in a random access memory).

- At the start of a reconfiguration process a switch may not be aware of all its channel dependencies. Therefore, in order to prevent deadlock during the reconfiguration, caution must be exercised when data packets are to be forwarded. Assume that a packet bound for an output channel c_{o2} arrives before the token on an input channel c_{i1}; that the dependency from c_{i1} to c_{o2} was previously unknown; and that c_{o2} has already transmitted the token. Then, the packet must be discarded since forwarding it might cause deadlock (as the packet is routed according to R_{old} and the output channel has already transmitted the token). If the system has been running for a reasonable amount of time (since startup or a previous reconfiguration) we believe that there is only a low probability of such packet discarding due to unknown channel dependencies. However, if any packet discarding is unacceptable and transmission of dummy information in an all-to-all fashion is acceptable, disclosure of all channel dependencies could be forced by letting each of the processing nodes transmit dummy packets to all other processing nodes (at system startup and, in case of a reconfiguration, immediately following the token).

Figures 5.1, 5.2 and 5.3 present a pseudocode, inspired by Java syntax, to illustrate channel dependency acquisition and reconfiguration using OR in a source routing environment. For simplicity, we assume that only one reconfiguration process is ongoing at a time. Figure 5.1 shows an example data structure for state information to be gathered and maintained by a switch. In addition, Figure 5.1 includes a method for dispatching packets that arrive at the head of the bu er of an input channel. We assume that packets are input to this method according to a suitable scheduling scheme. The arrival of a token packet results in a call to the method shown in Figure 5.2, whereas the arrival of a data packet results in a call to the method shown in Figure 5.3. Some of the methods called in Figures 5.1, 5.2 and 5.3, such as `GetType`, `TransmitToken` and `ForwardDataPacket`, are not detailed, but their purposes should be obvious from their names and from the accompanying comments. The pseudocode examples are included to illustrate main principles, not to present a minimal data structure or an e cient implementation of OR in a source routing environment.

```
boolean Reconf      // Is a reconfiguration in progress?

const int NUM_IN    // Number of in-channels
const int NUM_OUT   // Number of out-channels

int InID            // Current in-channel, [0..NUM_IN-1]
int OutID           // Current out-channel, [0..NUM_OUT-1]

int NumProc         // Number of in-channels with processed token
int NumTrans        // Number of out-channels with transmitted token

boolean[NUM_IN] Proc            // Has InID processed token?
boolean[NUM_OUT] Trans          // Has OutID transmitted token?

boolean[NUM_IN][NUM_OUT] ChDep // Channel dependency InID->OutID?



void DispatchPacket (Packet, InID) {
  // Dispatch the packet at the head of InID's buffer
  // according to its type (token or data)
  if (GetType(Packet) == TOKEN_PACKET) {
    ProcessToken(InID);
  }
  else if (GetType(Packet) == DATA_PACKET) {
    ProcessData(InID, Packet);
  }
}
```

Figure 5.1: Reconfiguration and channel dependency acquisition in a source routing switch: Example data structure and method for dispatching an incoming packet.

5.1.3 A performance optimization

We know from [139, 140] that the deadlock avoidance mechanism of OR causes performance penalties in terms of increased latency and decreased throughput during a reconfiguration, due to a number of $\text{packets}_{\text{new}}$ being held back awaiting transmission of tokens on output channels. In order to alleviate such performance penalties for source routing environments, we propose an optimization of OR that takes advantage of the fact that during a reconfiguration a number of $\text{packets}_{\text{new}}$ may follow routes that are also legal routes according to R_{old}. We will refer to such packets as $\text{packets}_{\text{oldCompatible}}$, whereas $\text{packets}_{\text{new}}$ that do not follow legal routes according to R_{old} will be referred to as $\text{packets}_{\text{oldIncompatible}}$.

As an example, let us consider a 4×4 mesh topology, and assume that R_{old} is based on the Up*/Down* [204] graph shown in Figure 5.4(a), and that node 4 is suddenly removed from the topology. Remember that Up*/Down* assigns up and down directions to all the communication channels in a network to form a directed acyclic graph rooted in one of the nodes, and that deadlock is avoided by prohibiting the turn from a down-link to an up-link. (In Figure 5.4 an arrow indicates the direction of an up-link.) The removal of node 4 disconnects R_{old} since nodes 8 and 12 can no longer communicate with all other nodes. Let us further assume that R_{new} is based on the modified Up*/Down* graph shown in Figure 5.4(b), where the directions have been changed for

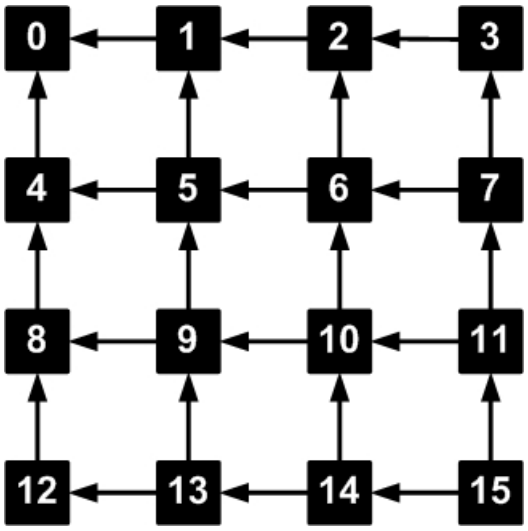
```
void ProcessToken (InID) {
  // Process the token at the head of InID's buffer
  Proc[InID] = true;
  NumProc++;

  // If first input channel to process token,
  // mark reconfiguration in progress
  if (!Reconf) {
    Reconf = true;
  }

  // Can token now be transmitted on any output channel?
  for (int outCh = 0; outCh < NUM_OUT; outCh++) {
    if (Trans[outCh]) {
      // outCh has already transmitted token
      continue;
    }

    // Have all inCh where dependency inCh->outCh processed token?
    boolean allProc = true;
    for (int inCh = 0; inCh < NUM_IN; inCh++) {
      if (ChDep[inCh][outCh] && !Proc[inCh]) {
        // inCh has not yet processed token
        allProc = false;
        break;
      }
    }

    if (allProc) {
      // All inCh where dependency inCh->outCh have processed token
      // Transmit token on outCh
      TransmitToken(outCh);
      NumTrans++;

      // Reset all dependencies towards outCh to prepare for
      // registration of dependencies for new routing function
      SetArray(ChDep[][outCh], false);
    }
  }

  if (NumProc == NUM_IN && NumTrans == NUM_OUT) {
    // Reconfiguration over, prepare for subsequent reconfiguration
    Reconf = false;
    NumProc = 0;
    NumTrans = 0;
    SetArray(Proc, false);
    SetArray(Trans, false);
  }
}
```

Figure 5.2: Reconfiguration and channel dependency acquisition in a source routing switch: Example of processing an incoming token.

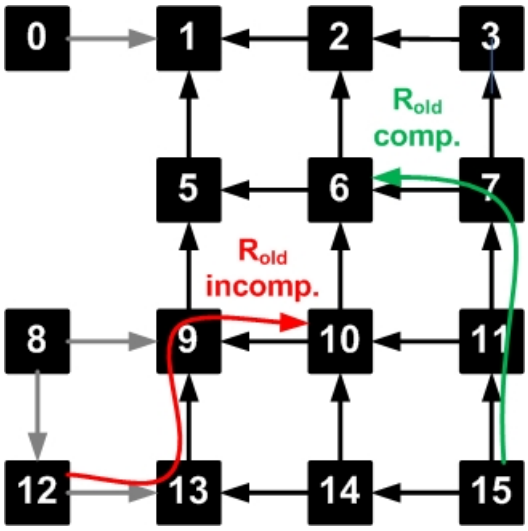
```
void ProcessData (InID, Packet) {
  // Process the data packet at the head of InID's buffer

  // Extract the output channel from the packet
  OutID = GetOutputChannel(Packet);

  if (Reconf) {
    // Reconfiguration ongoing

    if (!Proc[InID]) {
      // InID has not processed token

      if (!Trans[OutID]) {
        // OutID has not transmitted token
        // Packet can be forwarded safely
        ForwardDataPacket(Packet);

        // Register dependency despite ongoing reconfiguration
        ChDep[InID][OutID] = true;
      }
      else {
        // OutID has transmitted token.
        // Dependency InID->OutID previously unknown
        // Packet can never be forwarded, and is discarded
        DiscardDataPacket(Packet);
      }
    }
    else if (Trans[OutID]){
      // InID has processed and OutID has transmitted token
      // Forward packet
      ForwardDataPacket(Packet);

      // Register dependency (for new routing function)
      ChDep[InID][OutID] = true;
    }
    else {
      // InID has processed but OutID has not yet transmitted token
      // Packet must be held back until OutID has transmitted token
      // It will be re-entered into method DispatchPacket
      // No action
      ;
    }
  }
  else {
    // Reconfiguration not ongoing
    // Forward packet
    ForwardDataPacket(Packet);

    // Register dependency
    ChDep[InID][OutID] = true;
  }
}
```

Figure 5.3: Reconfiguration and channel dependency acquisition in a source routing switch: Example of processing an incoming data packet.

all four (gray coloured) communication channels connected with the nodes in the leftmost column of the mesh. Then, for R_{new}, the route indicated from node 15 to node 6 is an example of a route compatible with R_{old}, whereas the route indicated from node 12 to node 10 is an example of a route incompatible with R_{old}.



(a) Up*/Down* graph for R_{old}.



(b) Up*/Down* graph for R_{new}. One of the marked routes is compatible with R_{old} whereas the other is incompatible.

Figure 5.4: Examples of R_{new}-routes compatible and incompatible with R_{old}.

We propose the following optimization of the OR algorithm. Assume that a reconfiguration is in progress, and that a $\text{packet}_{\text{oldCompatible}}$ is transmitted from a source s to a destination d. Then, in every switch traversed from s to d, this packet may be forwarded to an output channel regardless of whether the token has already been transmitted on the output channel. This relaxation of the OR algorithm will not cause deadlock since the packet follows a route that is compatible with R_{old} (see Lemma 5.1 and its proof).

For an elaboration on how the optimization can be realized within a switch, assume that the switch has channel dependencies from input channels c_{i0}, c_{i1}, and c_{i2} to output channel c_{o2}; that c_{i0} and c_{i2} have not yet processed the token; and that a $\text{packet}_{\text{oldCompatible}}$ bound for c_{o2} arrives after the token on c_{i1}. Then, the optimization allows immediate forwarding of the packet to c_{o2}, even if c_{o2} has not yet transmitted the token. Thus, the packet escapes any delay caused by the token forwarding regime.

Remember that the OR algorithm consists of a set of rules controlling the behaviour of packet injection channels, input channels of switches, and output channels of switches. Our optimization of OR only a ects the behaviour of switch input channels, not injection channels and switch output channels. For a source routing system to apply the optimized OR algorithm, the rule that governs the behaviour of a switch input channel (the second bullet point discussed in Section 5.1.2) becomes as follows:

- After having processed the token, a *switch input channel* may dispatch $\text{packets}_{\text{oldIncompatible}}$ solely to output channels which have already transmitted their token. $\text{Packets}_{\text{oldIncompatible}}$ bound for output channels that have not yet transmitted the token are temporarily held

back. $\text{Packets}_{\text{oldCompatible}}$ may be forwarded regardless of whether an output channel have transmitted its token.

Lemma 5.1. *The optimized OR algorithm provides deadlock-free reconfiguration in a source routing environment.*

Proof. The proof is by contradiction. Assume that a deadlocked set of packets, S_d, is a set of packets where none of the packets can advance before another packet in the set advances. Assume also that a reconfiguration from R_{old} to R_{new}, where the optimized OR algorithm is applied, results in some non-empty S_d.

As both R_{old} and R_{new} are by themselves deadlock-free, S_d must include a mix of $\text{packets}_{\text{old}}$ and $\text{packets}_{\text{new}}$ which are mutually waiting for each other to advance. We must consider the three different situations that could lead to a non-empty S_d consisting of a mix of $\text{packets}_{\text{old}}$ and $\text{packets}_{\text{new}}$:

1. Assume that at least one $\text{packet}_{\text{old}}$ is transmitted after the token on a communication channel. This contradicts the optimized (as well as the non-optimized) OR algorithm.

2. Assume that at least one $\text{packet}_{\text{new}}$ is transmitted before the token on a communication channel, and that the packet follows a route that is incompatible with R_{old}. This contradicts the optimized OR algorithm which states that $\text{packets}_{\text{oldIncompatible}}$ bound for output channels that have not yet transmitted the token are temporarily held back.

3. Assume that at least one $\text{packet}_{\text{new}}$ is transmitted before the token on a communication channel, and that the packet follows a route that is compatible with R_{old}. The optimized OR algorithm allows $\text{packets}_{\text{oldCompatible}}$ to be transmitted before the token on a communication channel. However, any claim that a set of packets consisting of $\text{packets}_{\text{oldCompatible}}$ and $\text{packets}_{\text{old}}$ may be deadlocked contradicts the assumption that R_{old} is deadlock-free.

□

Assuming that both R_{old} and R_{new} are deterministic routing functions, [140] shows how the original OR algorithm ensures in-order packet delivery by enforcing the rules of token propagation also on the final hop of any path – the (internal or external) communication channel connecting a switch to a destination processing node. However, if the final hop has multiple (virtual or physical) communication channels, a token aware mechanism is also needed on the receiving end of the final hop to ensure in-order packet delivery. Assume that the final hop has k communication channels. Then, in-order packet delivery could be realized by treating the receiving end of the final hop as a switch with k input channels and a single output channel, where a channel dependency exists from each of the k input channels to the single output channel, and where the non-optimized OR algorithm applies.

As described above, the optimization of OR allows immediate forwarding of a $\text{packet}_{\text{new}}$ that follows a route which is *compatible* with R_{old}. However, such a route is not necessarily *identical* to the corresponding route of R_{old}. Although the version of the optimization described above ensures deadlock avoidance during a reconfiguration, it cannot ensure in-order delivery of packets.

The following example illustrates the risk of out-of-order packet delivery. Assume that s and d are a source and destination processing node, respectively; that d is connected to a switch D; that r_o and r_n are the routes from s to d according to R_{old} and R_{new}, respectively; and that r_n

is compatible with R_{old} although $r_n \neq r_o$. Then, a risk exists that D may forward some packet following r_n to d before having forwarded all packets following r_o to d.

A more restrictive version of the optimization could allow immediate forwarding only of those $\text{packets}_{\text{new}}$ that follow a route which is identical to the corresponding route of R_{old}, and thereby support in-order packet delivery. Assume that $S_{identical}$ refers to the set of routes in R_{new} for which identical routes exist in R_{old}, and that $S_{compatible}$ refers to the set of routes in R_{new} which are compatible with R_{old}. Then, $S_{identical}$ is a subset of $S_{compatible}$. Thus, Lemma 5.1 and its proof are equally valid for the more restrictive version of the optimization.

The optimized OR algorithm requires that each $\text{packet}_{\text{new}}$ carries information stating whether or not its route is also a legal (either compatible or identical) route of R_{old}. The implementation of the optimization thus requires a new status bit in each packet, and additional logic to set this status bit. Furthermore, additional logic is also needed in the switches to check whether or not a packet can be forwarded independently of the token forwarding status.

Given that a central entity is responsible for the route calculation, it could also set and distribute the status bit for each of the routes in R_{new}. For checking whether a route that belongs to R_{new} is *compatible* with R_{old}, the use of such a central entity is perhaps the most straightforward solution. However, such a check for compatibility is also conceivable if distributed route calculation is used – see e.g. [189] where Up*/Down* routing is assumed and the entire directed acyclic graph is distributed to the processing nodes.

When using OR for reconfiguration of a source routing system, a continuous injection of tra c generally presupposes that the processing nodes keep the routing tables for R_{old} (and do not inject tokens) until their routing tables for R_{new} are fully available. Thus, as each processing node must have storage space for the routing tables of both R_{old} and R_{new}, the check of whether a route that belongs to R_{new} is *identical* with the corresponding route in R_{old} could be performed in the following alternative manner. Assume that a $\text{packet}_{\text{new}}$, pkt, is to be sent from a source s to a destination d; that r_n is returned as the result of a route-lookup in R_{new}; and that r_o is returned as the result of a route-lookup in R_{old}. Then, the status bit in pkt should be set if r_n and r_o are equal. This check should be performed for each $\text{packet}_{\text{new}}$ to be injected for the duration of the reconfiguration process.

For a source routing switch, Figure 5.5 illustrates how a simple modification of the empty else-clause in the `ProcessData` method from Figure 5.3 could realize the optimization of OR. For simplicity, the registration of channel dependencies for R_{new} is assumed not to start until the token has been transmitted on the output channel.

The optimization of OR is evaluated in Sections 5.1.5 and 5.1.6, where its least restrictive version is assumed.

5.1.4 Token injection synchronization

When using OR, all tra c sources inject a token to indicate that the last packet that follows R_{old} has been sent on an injection channel, and that subsequently injected packets will be routed according to R_{new}. Token injection could be fully synchronized if all tra c sources injected their tokens simultaneously.

Due to such factors as clock skew or reception of routing or control information at di erent times such synchronization is hard to achieve. A tra c source may inject the token and change

```
void ProcessData (InID, Packet) {
  // Process the data packet at the head of InID's buffer

    .
    .
    .

    else {
      // InID has processed but OutID has not yet transmitted token
      // Packet must be held back until OutID has transmitted token
      // It will be re-entered into method DispatchPacket
      // No action
      ;
    }

    .
    .
    .
}



void OptimizedProcessData (InID, Packet) {
  // Process the data packet at the head of InID's buffer
  // Optimization included

    .
    .
    .

    else {
      // InID has processed but OutID has not yet transmitted token

      if (PacketOldCompatible(packet)) {
        // Packet compatible with old routing function: Forward
        ForwardDataPacket(Packet);
      }
      else {
        // Packet not compatible with old routing function:
        // Hold back until OutID has transmitted token
        // Packet will be re-entered into method DispatchPacket
        // No action
        ;
      }
    }

    .
    .
    .
}
```

Figure 5.5: Else-clause (from Figure 5.3) a ected by the optimization: Comparison of original and optimized OR algorithm.

routing function as soon as it has received or calculated the new routing information. This approach may lead to highly unsynchronized token injection as the tra c sources may have their routing information ready (and thus inject their tokens) over a significant period of time. For a technology that utilizes a central control, improved synchronization may be achieved. Then, each tra c source could – after having received updated routing information – defer injection of the token until a following control packet that enables token injection is received. In this case the level of synchronization depends on such issues as varying distance between the central control unit and the individual tra c sources, and on whether or not the technology supports packet broadcast. If, on the other hand, token injection is triggered at a predetermined time, the injection of tokens will be as synchronized as clock skew allows. A drawback of the latter alternative is the need to postpone reconfiguration for a period of time to ensure that all tra c sources possess the updated routing information. This could cause considerable packet loss, for instance in the case of a network component failure.

The following example, which assumes that the optimization suggested in Section 5.1.3 is not used, illustrates why token injection synchronization is an issue when using OR. Assume that a switch has channel dependencies from input channels c_{i0}, c_{i1} and c_{i2} to output channel c_{o2}; that c_{i0} and c_{i2} have not yet processed the token; and that a $\text{packet}_{\text{new}}$ bound for c_{o2} arrives after the token on c_{i1}. This packet cannot be forwarded to c_{o2} until c_{o2} has transmitted the token, which depends on prior token processing by both c_{i0} and c_{i2}. If the token arrival at c_{i0} or c_{i2} depends on processing nodes that inject tokens at a late stage, the packet may experience a significant delay.

The performance impact of token injection synchronization for a source routing environment is evaluated in Sections 5.1.5 and 5.1.6.

5.1.5 Experiment setup

Our evaluation of the performance of OR in source routing environments focuses on the impact of token injection synchronization as well as on the impact of the new optimization of OR proposed in Section 5.1.3.

The results presented in [139] show that OR performs significantly better than traditional static reconfiguration and has similar performance to the Double Scheme [179], one of the most e cient general dynamic reconfiguration methods available in the literature. The adaptations we suggested to utilize OR in a source routing environment should not significantly a ect the performance of the method itself (given that all channel dependencies are known at the start of a reconfiguration). Thus, as another comparison of OR with alternative reconfiguration schemes is not expected to provide new information, it is not performed here.

We consider both mesh and torus topologies of size 8×8 and 16×16. A reconfiguration is triggered after 8 000 cycles, either by a switch fault (for evaluation of the proposed optimization of OR) or by a change of routing function (for evaluation of token injection synchronization). Packets are removed from the network upon reaching a faulty switch. At the start of a reconfiguration process all switches are assumed to know their channel dependencies. Thus, no packets are discarded due to unknown channel dependencies.

In the case of a switch fault, notification messages are distributed to all una ected processing nodes, whereas the processing nodes are notified immediately if the reconfiguration is caused by a change of routing function. When all processing nodes have been notified to initiate reconfigu-

ration, each processing node draws a token injection time t_{token} from a normal distribution with a mean of 500 cycles and where the standard deviation is a simulation parameter ($token_{std}$) that assumes the values 0, 5, 10, 20, 30, 40, 50 and 100 to model different levels of token injection synchronization. A $token_{std} = 0$ represents fully synchronized token injection, and the higher the $token_{std}$ is, the more unsynchronized the token injection becomes. Each processing node starts a timer according to t_{token} and continues injecting $\text{packets}_{\text{old}}$ until the timer expires, then injects the token, and thereafter injects $\text{packets}_{\text{new}}$.

Data are collected for a period of 20 000 cycles, and several metrics are considered. Thr_t and Lat_t result from the division of the data collection period into 200 time intervals, each with a duration of 100 cycles. For a time interval int, Thr_t is the number of packets that are generated by any processing node in int and that subsequently reach their destination processing node. Lat_t for int is the average latency of all packets that are generated by any processing node in int and that subsequently reach their destination processing node. The latency for a single packet is the time that elapses from when the packet is generated and injected into the transmission queue of the source processing node until the packet header is received by the destination processing node.

For each int the values for Thr_t and Lat_t are plotted in the middle of the interval, whereas in the same plots the start and end times of the reconfiguration period are plotted without regard to interval borders. The reconfiguration period is the time that elapses from when the first token is injected by a processing node (start of reconfiguration) until the last token is received by a processing node (end of reconfiguration). We also monitor the number of packets rejected by source processing nodes during reconfiguration, which indicates the severity of the service interruption experienced by running applications. Each experiment is repeated 16 times (with a different seed), and the mean values of these repetitions are presented together with their 95% confidence intervals.

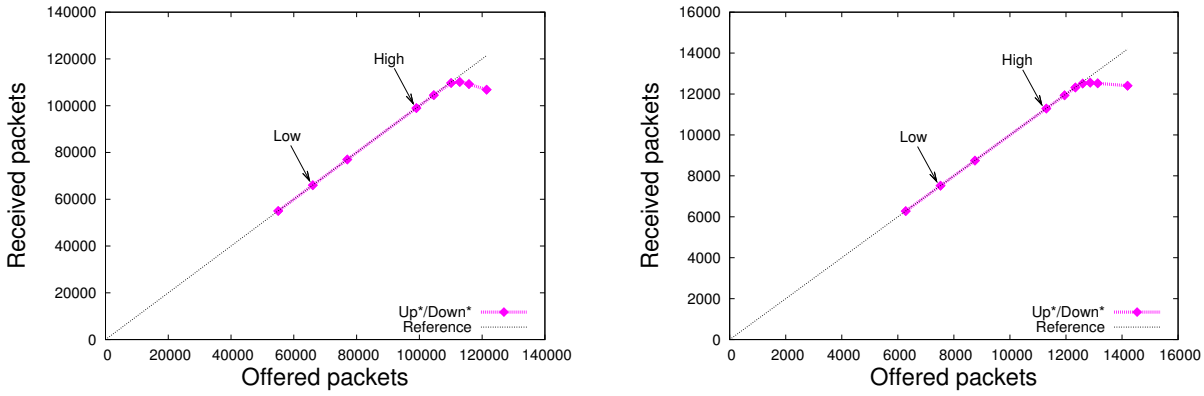
We study two different traffic patterns – a uniform destination address distribution, and a hotspot traffic pattern where 80% of the packets are destined for a hotspot node located in the middle[5] of the topology whereas the remaining 20% of the packets are uniformly distributed.

A transmission queue in a processing node has space for 6 packets, and overflows when the network cannot deliver packets at the rate they are injected. Both an ingress and egress buffer of a switch port can hold 6 packets. OR does not require VCs for deadlock avoidance, and we do not assume availability of VCs in these experiments.

Our focus is on the performance of OR – which can be used for any routing algorithm – not on the performance of specific routing algorithms. The topology agnostic Up*/Down* routing algorithm was chosen for these experiments as it is conceptually simple, well known, and does not depend on VCs for deadlock avoidance. We use the original method proposed in [204] since it allows the Up*/Down* root to be set by a simulation parameter (in [196, 199] new heuristics were applied, e.g. for identifying the best root).

An initial set of experiments – simulating neither reconfiguration nor component faults – was performed in order to ensure relevant load levels for the subsequent experiments. For every combination of topology and traffic pattern we first identified the network saturation point – the load level where traffic sources start to discard packets due to transmission queue overflow. Figures 5.6(a) and 5.6(b) show the network saturation point (corresponding to the knee-point of the Up*/Down* curve) for a 16×16 torus topology under uniform and hotspot traffic, respectively.

[5] In order to simplify the discussion, we visualize a torus topology laid out in a plane as a mesh topology with wraparound channels.



(a) Uniform tra c.

(b) Hotspot tra c.

Figure 5.6: The high and low load levels (90% and 60% of the network saturation point, respectively) used in the experiments for the 16×16 torus.

For uniform tra c the saturation point occurs when approximately 110 000 packets are injected into the network over the simulation period of 20 000 cycles, whereas for hotspot tra c saturation occurs at the injection of approximately 12 500 packets. After having identified the network saturation point, two load levels were selected for use in the following experiments. The lowest and highest load levels correspond to 60% and 90%, respectively, of the saturation point. These two load levels are indicated in Figures 5.6(a) and 5.6(b).

For all of our experiments, R_{old} is based on an Up*/Down* graph with the root in the upper left corner node of the topology. Remember that the root of an Up*/Down* graph is the one node with only incoming up-links, and that a leaf is a node with only outgoing up-links.

The first of our main sets of experiments evaluates how di erent degrees of synchronization of the token injection a ect the performance of OR for source routing systems. A reconfiguration from R_{old} to R_{new} is assumed to be triggered by a need – not caused by a topology change – to change the routing function. R_{new} is based on an Up*/Down* graph with the root in the upper right corner node of the topology. The optimization of the OR algorithm is not applied.

The second set of experiments evaluates the performance impact of the optimization of OR. The least restrictive version of the optimization is applied, and we study two di erent scenarios. The first scenario represents a case where a significant part of the routes of R_{new} are incompatible with R_{old}. We assume that a reconfiguration process is triggered by a disconnection of R_{old} caused by a fault in the root node of the Up*/Down* graph, and let R_{new} be based on a new Up*/Down* graph with the root in the upper right corner node of the topology. The second scenario represents a case where all the routes of R_{new} are compatible with R_{old}. In order to represent such a best-case scenario for the optimization of OR, a fault induced in the leaf node of the Up*/Down* graph of R_{old} triggers a reconfiguration process (although, strictly speaking, a reconfiguration is not necessary when a leaf node fails), and R_{new} is based on the resulting Up*/Down* graph.

5.1.6 Results

In the following, we present the results of our evaluations of two di erent matters that may a ect the performance of the OR algorithm in source routing environments. We first evaluate the impact on performance of di erent levels of token injection synchronization. Recall from Section 5.1.4

that traffic sources may transmit their tokens in an unsynchronized manner due to such factors as clock skew or reception of routing or control information at different times. Secondly, we evaluate the optimization of the OR algorithm introduced in Section 5.1.3 which allows $\text{packets}_{\text{new}}$ with routes compatible to R_{old} to be forwarded independently of the token propagation status. We have selected and included a set of plots which is representative of the results of our experiments.

Token injection synchronization

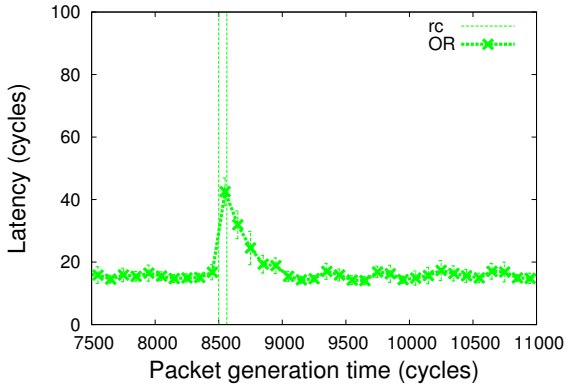
Recall that these experiments are based on the following assumptions: Reconfiguration is triggered by a change of routing function (not caused by topology change); the reconfiguration moves the root of the Up*/Down* graph from the upper left to the upper right corner of the topology; and the new optimization of the OR algorithm is not applied. Also recall that the range of $token_{std}$ values represents different levels of token injection synchronization.

Figures 5.7 and 5.8 show Lat_t and Thr_t, respectively, for a 16×16 mesh under a high traffic load for $token_{std}$ values of 0, 40, and 100. The vertical lines indicate the start and end times of a reconfiguration process. Figures 5.7(a), 5.7(c), and 5.7(e) show Lat_t for hotspot traffic for a selected period of time that includes the reconfiguration period. Corresponding results for uniform traffic are shown in Figures 5.7(b), 5.7(d), and 5.7(f). Figures 5.7(a) and 5.7(b) show the cases of fully synchronized token injection ($token_{std}$ is 0) and thus represent baselines for the consideration of Lat_t for hotspot and uniform traffic, respectively. Our experiments show crests in the Lat_t curves during reconfiguration even in the cases where token injection is fully synchronized. Tokens flow through the network according to the channel dependency graph of R_{old}, and – even if all processing nodes inject tokens simultaneously – the tokens that must be collected before an output channel of a switch can transmit its token may reach their input channels at different times. $\text{Packets}_{\text{new}}$ bound for an output channel of a switch must be held back until the token has been transmitted on the output channel. The withheld packets experience increased latency that results in the observed crests of the Lat_t curves. As token injection becomes more unsynchronized, both the height and width of the crests of the Lat_t curves increase. Figure 5.7 illustrates how the ratios of the peaks of the Lat_t curves to the latency values before reconfiguration starts are 2.6, 6.7, and 14.1 (hotspot traffic) and 2.5, 6.2, and 12.0 (uniform traffic) for $token_{std}$ values of 0, 40, and 100, respectively.

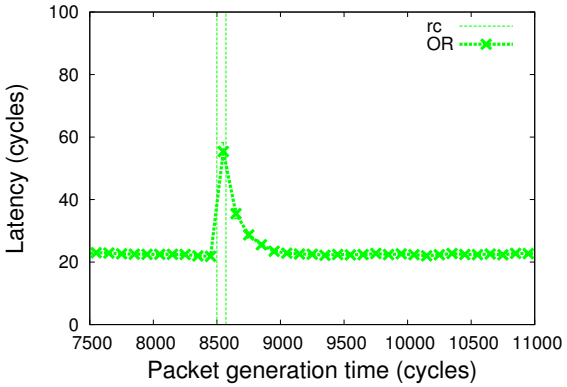
Figures 5.8(a), 5.8(c), and 5.8(e) show Thr_t for hotspot traffic. For hotspot traffic the packet throughput is limited due to 80% of the traffic being directed towards one particular node, and, for the values of $token_{std}$ that we have studied, no reconfiguration-related troughs are observed in the Thr_t curves.

Figures 5.8(b), 5.8(d), and 5.8(f) show Thr_t for uniform traffic. Figure 5.8(b) shows the case of fully synchronized token injection and thus represents the baseline for the consideration of Thr_t for uniform traffic. The plots illustrate how increasingly unsynchronized token injection results in a more severe fall in Thr_t during a prolonged reconfiguration period. For $token_{std}$ values of 40 and 100 the lowest points of the Thr_t curves are, respectively, 0.98 and 0.40 times the throughput values before reconfiguration starts.

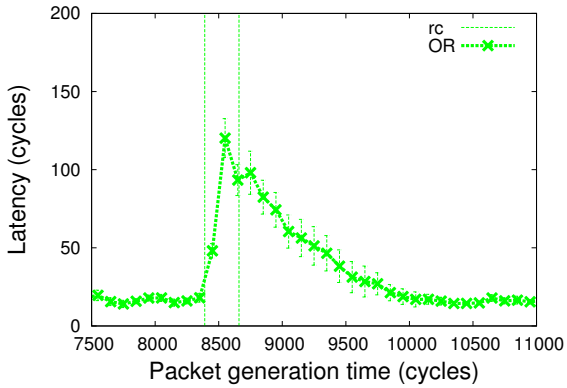
Congestion effects following a reconfiguration were in many cases observed for the combination of a high traffic load and highly unsynchronized token injection. The second crests of the Lat_t curves of Figures 5.7(e), and 5.7(f) (and also, less expressed, of Figure 5.7(c)) and the second



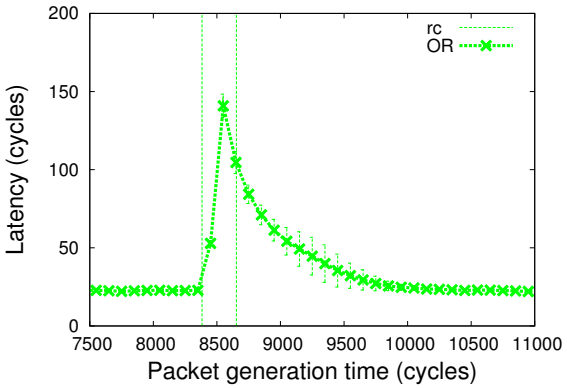
(a) Hotspot, $token_{std} = 0$.



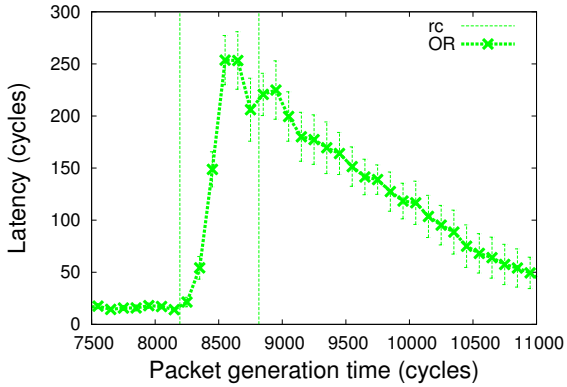
(b) Uniform, $token_{std} = 0$.



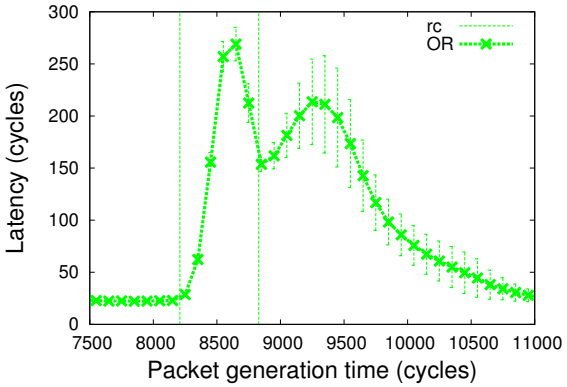
(c) Hotspot, $token_{std} = 40$.



(d) Uniform, $token_{std} = 40$.

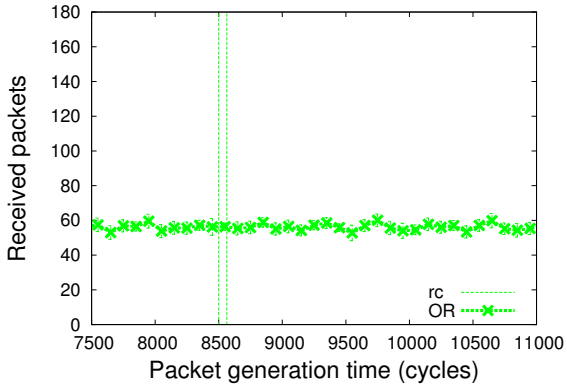


(e) Hotspot, $token_{std} = 100$.

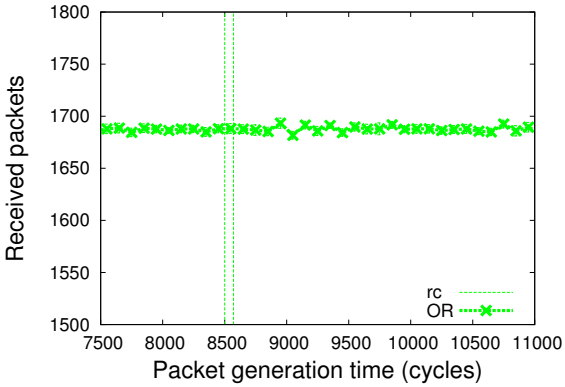


(f) Uniform, $token_{std} = 100$.

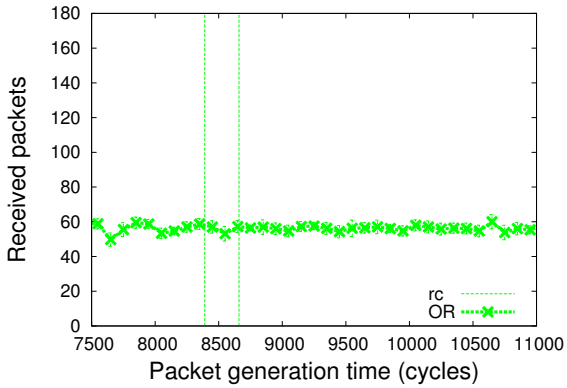
Figure 5.7: Lat_t and start/end times of reconfiguration (*rc*) for a 16×16 mesh under a high tra c load.



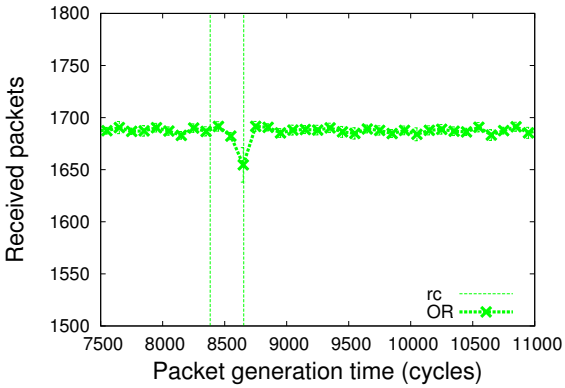
(a) Hotspot, $token_{std} = 0$.



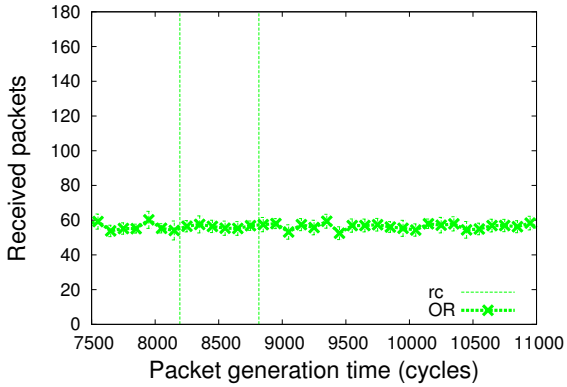
(b) Uniform, $token_{std} = 0$.



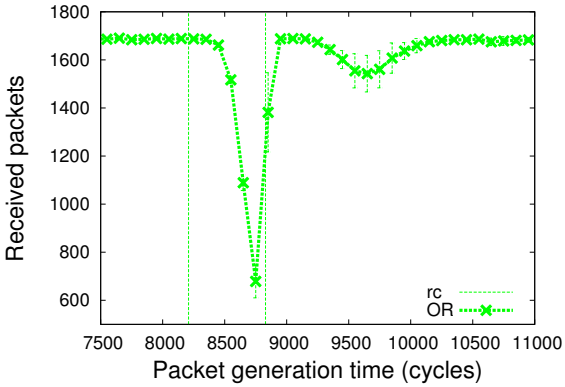
(c) Hotspot, $token_{std} = 40$.



(d) Uniform, $token_{std} = 40$.

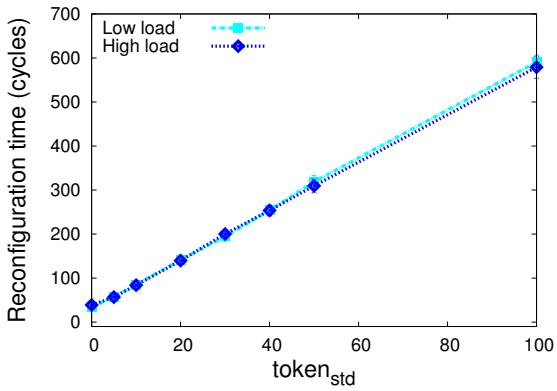


(e) Hotspot, $token_{std} = 100$.

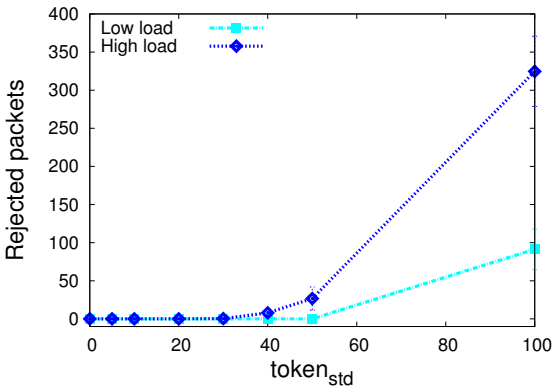


(f) Uniform, $token_{std} = 100$.

Figure 5.8: Thr_t and start/end times of reconfiguration (*rc*) for a 16×16 mesh under a high tra c load.



(a) Reconfiguration time, 16×16 torus.



(b) Packet rejection by tra c sources during reconfiguration, 8×8 torus.

Figure 5.9: Reconfiguration time and packet rejection (uniform tra c).

trough of the Thr_t curve of Figure 5.8(f) are manifestations of such congestion e ects. For these cases, we observe how the normalization of latency or throughput after reconfiguration is delayed. We believe that such congestion e ects are due to the release of a high number of packets that were held back during reconfiguration while awaiting token forwarding.

We observe from Figures 5.7 and 5.8 that the reconfiguration period is significantly lengthened for increasingly unsynchronized token injection. Figure 5.9(a) shows reconfiguration time as a function of $token_{std}$ for a 16×16 torus under uniform tra c. Our results demonstrate a close to linear relation between the $token_{std}$ and the duration of the reconfiguration period. We also note that the duration of the reconfiguration period is similar for high and low tra c load. The reason is that a considerable part of the data tra c is held back while awaiting token forwarding, and thus does not significantly delay the tokens.

The two load levels under study represent load levels below network saturation. Thus, the number of packets rejected by source processing nodes during a reconfiguration process gives an indication of the service interruption – caused by the reconfiguration – experienced by running applications. Whereas for hotspot tra c no such packet rejection occurred for any of our experiments, most of our experiments for uniform tra c show that a number of packets are rejected for the highest value(s) of $token_{std}$. Figure 5.9(b) shows packet rejection by source processing nodes during a reconfiguration (as a function of $token_{std}$) for an 8×8 torus under uniform tra c.

Algorithm optimization

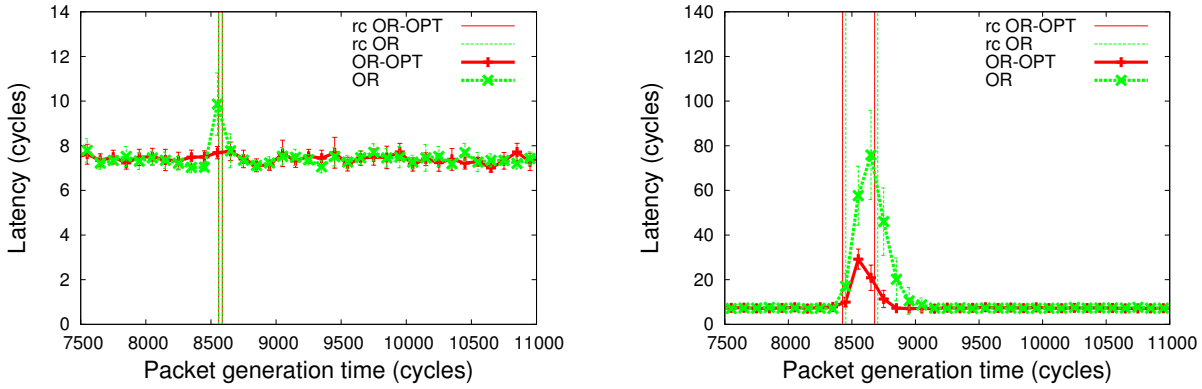
Two scenarios are presented in this section. In the first scenario we study the performance impact of the proposed optimization of OR for a case where a significant number of the routes of R_{new} are incompatible with R_{old}. We let a fault occur in the root node of the Up*/Down* graph, and during the reconfiguration process the root of the graph is moved from the upper left to the upper right corner of the topology.

In our plots, the prefix "OPT-" or su x "-OPT" indicates that the optimization of OR is applied, whereas the lack of such a prefix or su x indicates that the optimization is not applied.

Figure 5.10 shows Lat_t for an 8×8 mesh under a low tra c load with a hotspot pattern for $token_{std}$ values of 0 and 50. We observe how the optimization significantly reduces the crests

of the Lat_t curves that result from reconfiguration. For hotspot traffic we did not observe any troughs in the Thr_t curves caused by reconfiguration. This leaves no improvement potential for the optimization with respect to the Thr_t metric, and none of these plots are included.

For uniform traffic, Figure 5.11 shows Lat_t and Thr_t for an 8×8 mesh under a low traffic load for $token_{std}$ values of 0 and 50. Figures 5.11(a) and 5.11(c) show Lat_t, whereas Figures 5.11(b) and 5.11(d) show Thr_t. The reduction of Thr_t that starts at time 8 000 is due to packet loss in the faulty root node. In general, our results show that the effect of the optimization is less pronounced for uniform traffic than for hotspot traffic. Nevertheless, for uniform traffic the optimization does, in most cases, reduce the troughs of the Thr_t curves and the crests of the Lat_t curves that result from a reconfiguration.



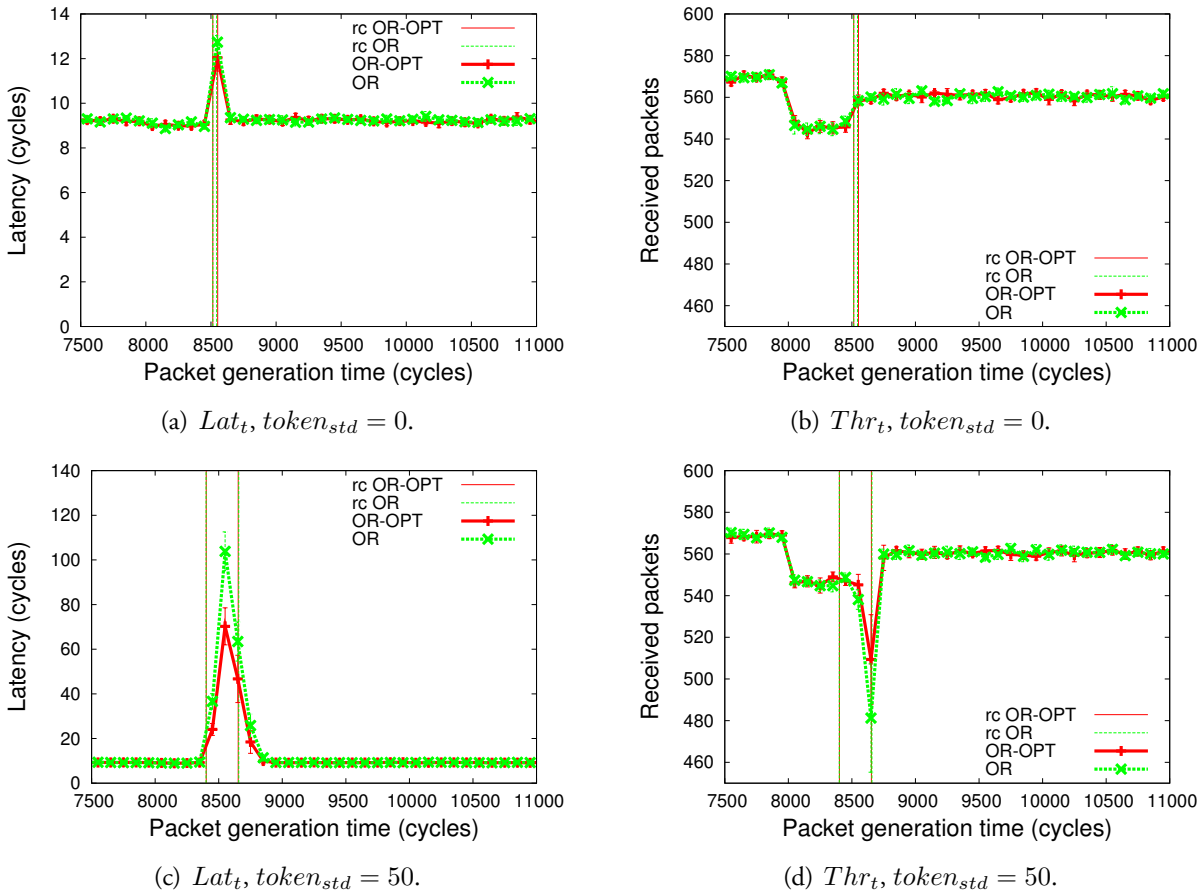
(a) $token_{std} = 0$. (b) $token_{std} = 50$.

Figure 5.10: Lat_t and start/end times of reconfiguration (*rc*) for an 8×8 mesh under a low traffic load with a hotspot pattern. 33% of packets$_{\text{new}}$ were found to follow routes incompatible with R_{old} (see Table 5.1).

The performance improvement potential of the optimization partly depends on two metrics that we will refer to as L_{frac} and L_{length}. L_{frac} is the fraction of packets$_{\text{new}}$ that are sent during a reconfiguration that also have legal (compatible) routes according to R_{old}. L_{length} is the lengths of the routes of those packets in proportion to the lengths of the routes of all packets$_{\text{new}}$ that are sent during the reconfiguration. The larger L_{frac} is, the more packets can be forwarded by the switches without regard to token forwarding status, and thus, the more packets avoid additional delays due to the reconfiguration. The larger L_{length} is, the more significant the reductions in latency become.

For the current scenario, Table 5.1 presents L_{frac} and L_{length} for the topologies and traffic patterns included in our study. These numbers, which are the average over our eight $token_{std}$ values and two load levels, show that L_{frac} is 58–74%, whereas L_{length} is 72–93%. For the 8×8 mesh we found that L_{frac} is similar for hotspot and uniform traffic (66.9% and 66.6%, respectively), whereas L_{length} is somewhat larger for hotspot traffic (85.6%) than for uniform traffic (81.8%). However, we do not believe that the difference in L_{length} alone explains why the benefit of the optimization is significant for hotspot traffic but more limited for uniform traffic (as demonstrated in Figures 5.10 and 5.11).

Virtual output queuing is not used in these experiments, and we believe that head-of-line blocking explains the limited effect of the optimization for uniform traffic when compared to hotspot traffic. In any switch packets$_{\text{new}}$ that follow routes compatible with R_{old} may be blocked by packets$_{\text{new}}$ that are bound for a different output port and that follow routes incompatible with



(a) Lat_t, $token_{std} = 0$.

(b) Thr_t, $token_{std} = 0$.

(c) Lat_t, $token_{std} = 50$.

(d) Thr_t, $token_{std} = 50$.

Figure 5.11: Lat_t, Thr_t, and start/end times of reconfiguration (*rc*) for an 8×8 mesh under a low tra c load with a uniform pattern. 33% of $\text{packets}_{\text{new}}$ were found to follow routes incompatible with R_{old} (see Table 5.1).

R_{old}. For the uniform tra c pattern every processing node is an equally likely packet destination, whereas for our hotspot tra c pattern 80% of the packets are destined for the same processing node. Thus, uniform tra c is more vulnerable to such head-of-line blocking e ects than hotspot tra c is.

OR could also be used in a system that applies virtual output queuing. Then, a token that arrives on an input channel must be replicated and inserted into all virtual output queues. Assume that a switch has channel dependencies from input channels c_{i0}, c_{i1}, and c_{i2} to an output channel c_{o2}, and that an input channel c_{ix} has n virtual output queues, $[c_{ix,0}, c_{ix,1} \ldots c_{ix,n-1}]$, where n corresponds to the number of output channels. Output channel c_{o2} must then wait until tokens have been processed by virtual output queues $c_{i0,2}$, $c_{i1,2}$, and $c_{i2,2}$ before transmitting the token.

In the second scenario all the routes of R_{new} are compatible with R_{old}, which gives the maximum improvement potential for the optimization: $L_{frac} = 100\%$; $L_{length} = 100\%$; and no head-of-line blocking e ects are caused by the token forwarding regime. Recall that, in order to represent such a best case scenario, a fault is induced in a leaf node of the Up*/Down* graph, whereas the root node is not moved by the reconfiguration process.

For an 8×8 torus under a high tra c load, and for $token_{std}$ values of 0 and 100, Figure 5.12 shows Lat_t for hotspot tra c, whereas Figure 5.13 shows Lat_t and Thr_t for uniform tra c. We observe that, for the current scenario, the optimization eliminates the troughs of the Thr_t curves

Table 5.1: L_{frac} and L_{length} when a reconfiguration, caused by a failure of the Up*/Down* root node, moves the root node from the upper left to upper right corner of the topology.

(a) Hotspot tra c pattern.

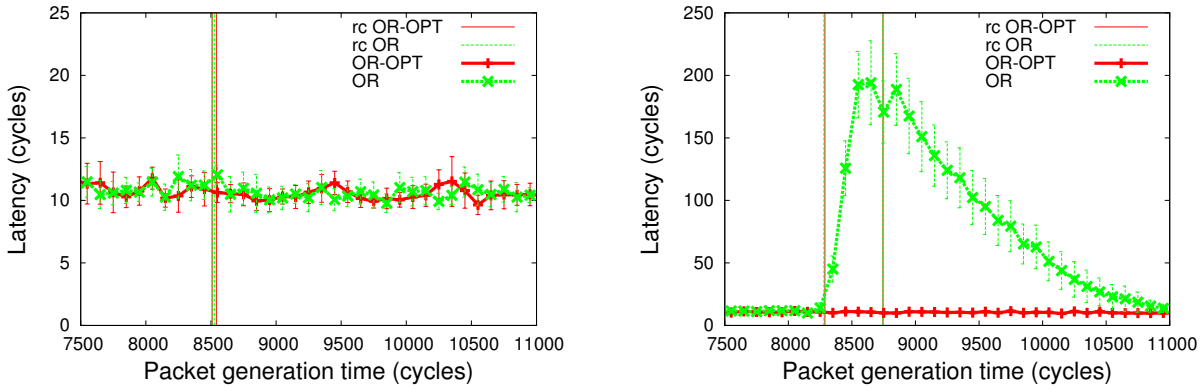
	8 × 8 Mesh	**16 × 16 Mesh**	**8 × 8 Torus**	**16 × 16 Torus**
$\mathbf{L_{frac}}$	66.9	60.1	74.3	72.1
$\mathbf{L_{length}}$	85.6	92.9	80.9	82.0

(b) Uniform tra c pattern.

	8 × 8 Mesh	**16 × 16 Mesh**	**8 × 8 Torus**	**16 × 16 Torus**
$\mathbf{L_{frac}}$	66.6	59.4	59.7	57.6
$\mathbf{L_{length}}$	81.8	92.1	72.3	72.7

as well as the crests of the Lat_t curves. Furthermore, Figure 5.14(a) demonstrates how the optimization eliminates congestion e ects caused by the release of a bulk of packets that were held back awaiting token forwarding. Finally, Figure 5.14(b) shows that, with the optimization, no packets were rejected by source processing nodes during the reconfiguration process. In the previous experiments, such packet rejection indicated service interruption for running applications.

Figure 5.14(a) demonstrates that the optimization of OR may in some cases prolong the reconfiguration period (since increased data tra c may delay token forwarding). However, when the negative e ects of a reconfiguration have been eliminated, a prolonged reconfiguration period becomes an issue of little importance (except perhaps with respect to an increased risk of out-of-order packet delivery for the least restrictive version of the optimization). Furthermore, for this absolute best case scenario with respect to optimization potential, token injection synchronization becomes another issue of little importance.



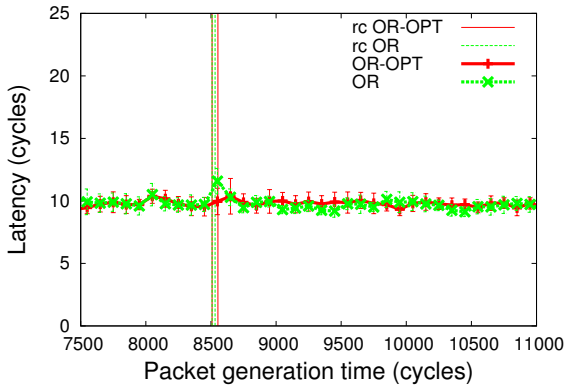
(a) $token_{std} = 0$.

(b) $token_{std} = 100$.

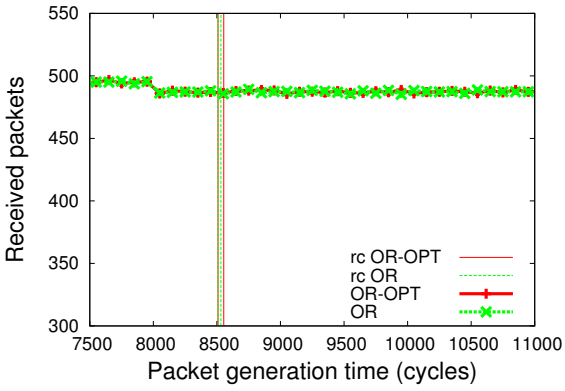
Figure 5.12: Lat_t and start/end times of reconfiguration (*rc*) for an 8×8 torus under a high tra c load with a hotspot pattern. A best case scenario for the optimization: All the routes of R_{new} are compatible with R_{old}.

5.2 Reconfiguration without performance penalties

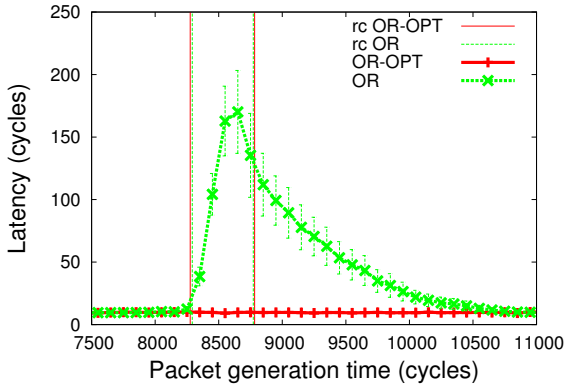
A reconfiguration process is prone to deadlock, and available reconfiguration strategies commonly include deadlock avoidance mechanisms that cause performance degradation, such as increased



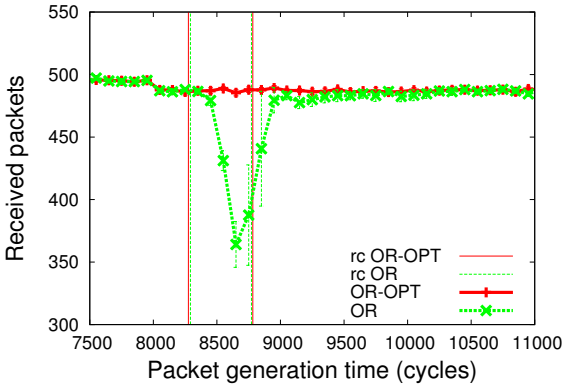
(a) Lat_t, $token_{std} = 0$.



(b) Thr_t, $token_{std} = 0$.

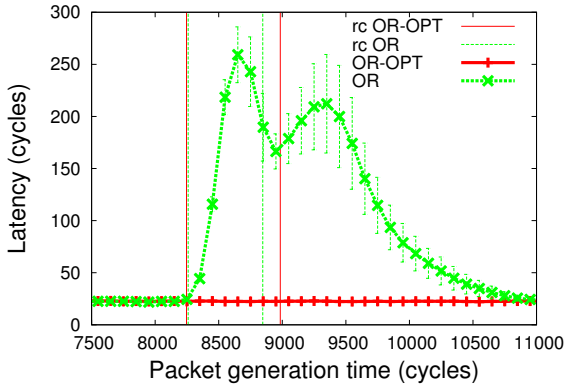


(c) Lat_t, $token_{std} = 100$.

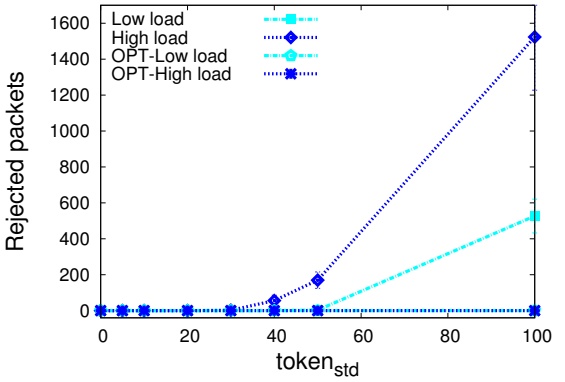


(d) Thr_t, $token_{std} = 100$.

Figure 5.13: Lat_t, Thr_t, and start/end times of reconfiguration (*rc*) for an 8×8 torus under a high tra c load with a uniform pattern. A best case scenario for the optimization: All the routes of R_{new} are compatible with R_{old}.



(a) Eliminating congestion e ects after reconfiguration. High tra c load, $token_{std} = 100$.



(b) Eliminating packet rejection by tra c sources during reconfiguration.

Figure 5.14: A best case scenario for the optimization for a 16×16 mesh under uniform tra c: Eliminating negative e ects of a reconfiguration when all the routes of R_{new} are compatible with R_{old}.

latency and decreased throughput, during the transition from R_{old} to R_{new}. We propose a new dynamic reconfiguration strategy – RecTOR – which does not impose such performance penalties during a reconfiguration.

RecTOR is based on a simple principle which ensures deadlock-freedom during the reconfiguration while packets that follow either R_{old}, R_{new} or both can coexist in the network without restrictions. It assumes the topology agnostic Transition-Oriented Routing (TOR) algorithm [200] which supports excellent performance. For instance, given sound path selection, TOR matches the performance of the topology specific Dimension-Order Routing (DOR) in meshes and tori. In addition to the advantages related to performance, RecTOR also has other merits. It does not require complex network switches; does not need more VCs than a routing function needs; and is useful for both source and distributed routing systems.

Section 5.2.1 presents RecTOR, whereas its performance is evaluated in Sections 5.2.2 and 5.2.3.

5.2.1 RecTOR

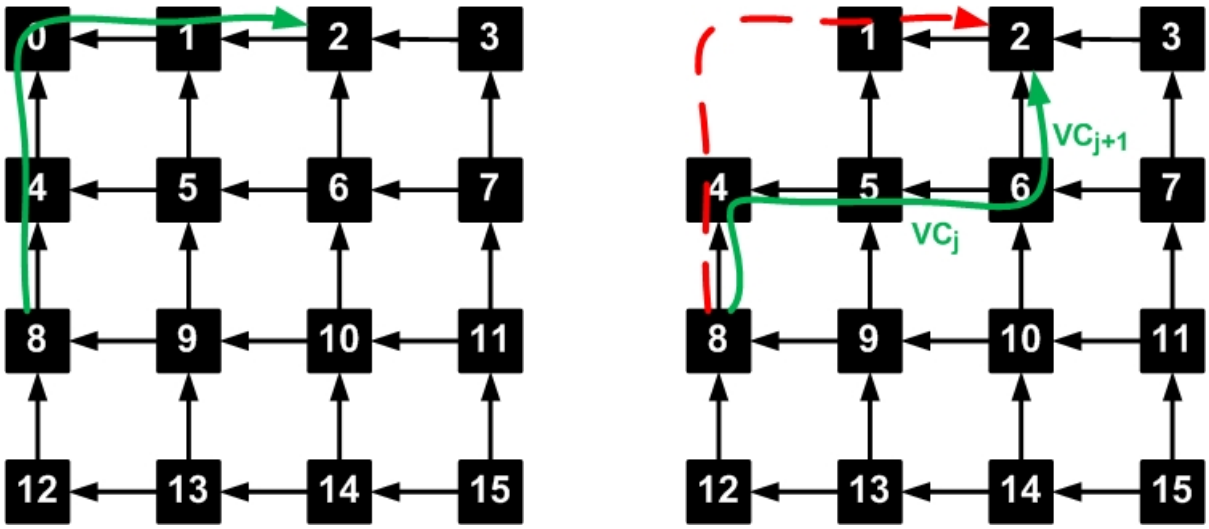
RecTOR assumes that an interconnection network applies the TOR routing algorithm. TOR was briefly presented in Section 2.8.1, and, as its manner of operation is essential for RecTOR, it is also recapitulated here.

TOR allows selection of any available path between a source and destination processing node, and depends on VC-transitions for deadlock avoidance. As was also assumed in [200], we assume in the following that such VC-transitions are based on a directed acyclic graph (DAG) corresponding to an Up*/Down* [204] graph. Communication channels and switches are represented in the DAG by edges and vertexes, respectively. TOR selects paths without regard to the edge directions of the underlying DAG. (Thus, in this respect, TOR considers the DAG simply as an undirected graph representing the topology.) For TOR, the main purpose of the DAG is to identify the *breakpoints* – the turns where cycles of channel dependencies must be broken. The breakpoints are the down-link to up-link turns (as with the Up*/Down* routing algorithm). TOR prevents deadlock by requiring that when a packet crosses a breakpoint (traverses from a down-link to an up-link), it makes a transition to the next higher VC. Thus, TOR supports a flexible shortest path routing.

RecTOR is based on the following observations concerning TOR. During the operation of a network, changes in the topology can be reflected in the DAG. TOR selects paths independently of the edge directions of the underlying DAG (which main purpose is to define the breakpoints that decide VC-transitions). Thus, after a topology change, a set of new paths that restores connectivity can always be found, provided that the topology is still physically connected and that a su cient number of VCs are available. Note in particular that, for computation of a connected routing function, TOR does not require the DAG to be a correct Up*/Down* graph.

Assume that G_{old} and G_{new} are the DAGs that apply before and after a topology change, respectively; that R_{old} is a deadlock-free routing function which includes VC-transitions according to G_{old}; and that R_{new} is a deadlock-free routing function which includes VC-transitions according to G_{new}. For a 4×4 mesh topology, Figure 5.15 shows an example of a G_{old} and G_{new} as the root node of the DAG (node 0) is removed. Figure 5.15(a) shows that the TOR route from node 8 to node 2 according to R_{old} crosses the root node of the DAG. When this node disappears TOR

must calculate a new route from node 8 to node 2, and one of the alternative choices is depicted in Figure 5.15(b). We observe how crossing node 6 represents a turn from a down-link to an up-link. Thus, TOR orders a transition to the next higher VC to guarantee deadlock-freedom. Notice, as an aside, that G_{new} is not a correct Up*/Down* graph as it has more than one root node.



(a) G_{old}. (b) G_{new}.

Figure 5.15: Example of DAG evolution and update of a route when a switch (the root of the DAG) is removed. An arrow indicates the direction of an up-link.

In order to ensure that packets routed according to R_{old}, R_{new} or both can coexist in the network without causing deadlock, RecTOR makes only one assumption on the evolution of the DAG: *No breakpoint must be moved* during the transition from G_{old} to G_{new}. Thus, as a general rule, an edge that persists from G_{old} to G_{new} must keep its (up or down) direction. However, breakpoints (and turns in general) can be removed or added as vertexes and edges are removed or added.

Assume that TOR is used, and that R_{old} (which is deadlock-free and includes VC-transitions according to G_{old}) applies when a topology change occurs for an interconnection network. Then, RecTOR prescribes the following procedure to reconfigure the network:

1. Update the underlying DAG to reflect the change in topology. If a communication channel or switch was removed, simply remove the corresponding edge or vertex (including its connecting edges). If a communication channel or switch was added, add an edge or vertex (including its connecting edges). Avoid introduction of cycles when assigning directions to newly added edges (see the Up*/Down* method).

2. Let TOR calculate a new deadlock-free routing function, R_{new} as follows: First, select a new set of paths which restore connectivity. Then, for each path, insert a transition to the next higher VC wherever the path crosses a breakpoint in G_{new}.

3. R_{new} can be applied instantly. Packets routed according to R_{old} and R_{new} – or possibly both in the case of distributed routing – can coexist in the network without causing deadlock.

With regard to step 3 above, there is a risk of a packet looping if the packet is routed according to R_{old} in some of the switches and R_{new} in others (which could happen if the system uses distributed routing). Such looping could cause packet loss, as a packet that has reached the highest available VC and still needs to make another VC-transition must be rejected. A simple approach that prevents this problem implies that each switch holds routing tables for both R_{old} and R_{new} during the reconfiguration, and that each packet is tagged to indicate which of the routing functions should be used. However, a better solution could be adopted from the Internet research community, where several studies (e.g. [76]) have focused on preventing packets from looping during the update of routing tables.

Like e.g. the Double Scheme [179], RecTOR cannot guarantee in-order packet delivery during a reconfiguration process.

As deadlock avoidance is an inherent challenge in dynamic reconfiguration, we include and prove Lemma 5.2.

Lemma 5.2. *RecTOR provides deadlock-free reconfiguration.*

Proof. The proof is by contradiction. Assume that a deadlocked set of packets, S_d, is a set of packets where none of the packets can advance before another packet in the set advances. Assume also that a reconfiguration from R_{old} to R_{new}, where RecTOR is applied, results in some non-empty S_d.

Both R_{old} and R_{new} are, by themselves, deadlock-free. Thus, S_d must include at least one packet that is taking a turn which is present in both G_{old} and G_{new}, and which is either a breakpoint in G_{old} and not a breakpoint in G_{new}, or a breakpoint in G_{new} and not a breakpoint in G_{old}. In either case some breakpoint must have been moved during the transition from G_{old} to G_{new}, which contradicts the premise of RecTOR.

□

5.2.2 Experiment setup

For the evaluation of the performance of RecTOR, we include a comparison with Overlapping Reconfiguration (OR) [139, 140]. OR is one of the most e cient and versatile reconfiguration strategies available in the literature, and was shown in [139] to have similar performance to the Double Scheme [179]. An adaptation of OR for use in source routing environments was a main topic of Section 5.1.

We consider both mesh and torus topologies of size 8×8 and 16×16. In order to compare the performance of RecTOR with the performance of OR, we conducted a number of experiments where a communication channel fault and, subsequently, a switch fault were introduced and handled by reconfiguration. The communication channel (port) fault occurs after 6 000 cycles and the switch fault occurs after 28 000 cycles. In each case a reconfiguration process is triggered.

As previously explained, RecTOR assumes that TOR is the routing algorithm used by an interconnection network. OR can be used between any pair of routing algorithms, and we consider two di erent alternatives. In the first case (referred to as OR_{TOR}), TOR is used. In the second case (referred to as $OR_{DOR/UD}$), DOR is used initially (for the fault-free topology) whereas Up*/Down* is used after the first network component has failed. As both DOR and Up*/Down* are simple and well established routing algorithms, this is a likely arrangement.

Whereas TOR and Up*/Down* are topology agnostic routing algorithms, DOR only works for fault-free meshes and tori. Recall from Section 2.5.1 that, for a two-dimensional mesh, DOR

avoids deadlock by first routing a packet in the X-dimension until the o set in this dimension is zero. Thereafter the packet is routed in the Y-dimension until it reaches its destination. For a torus, in addition, DOR needs two VCs for deadlock avoidance [55].

We use the original version of the Up*/Down* routing algorithm [204], and let the switch in the upper left corner[6] of the topology be the root of the Up*/Down* graph. (Unless this switch or one of its communication channels has failed, in which case a neighbour switch with healthy communication channels is selected as the root.)

TOR can calculate shortest path routes in a number of di erent ways. In these experiments an out-port in the X-dimension is preferred over an out-port in the Y-dimension in every intermediate switch. For a fault-free mesh or torus, this gives similar paths as DOR, although the VC-use is di erent for many of the paths. In these experiments, the switch in the upper left corner of the topology is the root of the underlying Up*/Down* graph.

The number of VCs available is 4. Each routing algorithm, TOR, DOR or Up*/Down*, evenly distributes the paths among the available VCs in order to achieve a balanced load.

We assume a source routing environment, and compare RecTOR and OR for di erent degrees of synchronization of the start of a reconfiguration process. For a source routing system, the change-over from R_{old} to R_{new} could be fully synchronized if all tra c sources performed the change simultaneously. As explained in Section 5.1.4, due to such factors as clock skew or reception of routing or control information at di erent times by the di erent tra c sources, such synchronization is hard to achieve. In these experiments, the processing nodes perform the change-over from R_{old} to R_{new} as follows. When all processing nodes have been notified to initiate reconfiguration, each processing node draws its own change-over time, t_{change}, from a normal distribution with a mean of 500 cycles and where the standard deviation is a simulation parameter, $change_{std}$.[7] Each processing node starts a timer according to t_{change} and continues injecting $\text{packets}_{\text{old}}$ until the timer expires, then, if OR is used, injects the token, and thereafter injects $\text{packets}_{\text{new}}$. The higher $change_{std}$ is, the more unsynchronized the change of routing function becomes. In these experiments, $change_{std}$ assumes the values 0 and 100, where the former value represents the fully synchronized case.

Although a source routing system is assumed in these experiments, RecTOR is also applicable for a distributed routing system. As it allows packets routed according to R_{old} and R_{new} – or possibly both in a distributed routing system – to coexist in the network without restrictions, we expect the behaviour of RecTOR to be relatively similar for source and distributed routing systems. The optimization of the OR algorithm, which was proposed in Section 5.1.3, is only applicable for source routing systems and is not applied in these experiments.

We study two di erent tra c patterns – a uniform destination address distribution, and a hotspot tra c pattern where 80% of the packets are destined for an appointed hotspot node whereas the remaining 20% of the packets are uniformly distributed.

A transmission queue in a processing node has space for 12 packets per VC, and overflows when the network cannot deliver packets at the rate they are injected. Both an ingress and egress bu er of a switch port can hold 6 packets per VC.

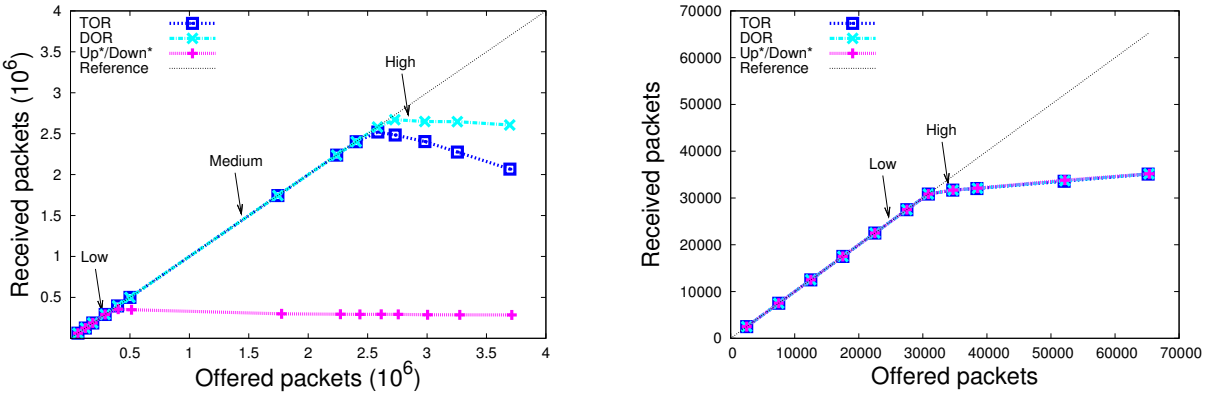
[6] As in Section 5.1, in order to simplify the discussion, we visualize a torus topology laid out in a plane as a mesh topology with wraparound channels.

[7] For OR, t_{change} and $change_{std}$ correspond to t_{token} and $token_{std}$, respectively, which were introduced in Section 5.1.5.

Data are collected over 100 000 cycles. We consider the metrics Thr_t and Lat_t which result from the division of the data collection period into 500 time intervals, each with a duration of 200 cycles. For a time interval int, Thr_t is the number of packets that are generated by any processing node in int and that subsequently reach their destination processing node. Lat_t for int is the average latency of all packets that are generated by any processing node in int and that subsequently reach their destination processing node. The latency for a single packet is the time that elapses from when the packet is generated and injected into a transmission queue in the source processing node until the packet header is received by the destination processing node.

For each int the values for Thr_t and Lat_t are plotted in the middle of the interval, whereas in the same plots the start and end times of the reconfiguration period are plotted without regard to interval borders. For OR the reconfiguration starts when the first token is injected and ends when the last token is received by a processing node. For RecTOR the reconfiguration starts when the first processing node takes R_{new} into use and ends when the last packet that follows R_{old} is removed from the network. The mean values that result from 30 repetitions of each experiment are presented in our plots. Each repetition is initialized by a di erent seed.

A communication channel (port) fault, switch fault and hotspot node (in the case of hotspot tra c) are selected randomly, but under the restriction that unique component faults and a unique hotspot node are used for each repetition of an experiment.



(a) 16×16 torus, uniform tra c.

(b) 8×8 mesh, hotspot tra c.

Figure 5.16: The load levels used in the experiments.

In order to ensure relevant load levels for the main experiments described above, an initial set of experiments was performed where neither network component faults nor reconfiguration occurred. For each of the three routing algorithms – TOR, DOR and Up*/Down* – the load level where the transmission queues of the processing nodes start to overflow was identified. Such saturation points are shown in Figure 5.16(a) for a 16×16 torus under uniform tra c and in Figure 5.16(b) for an 8×8 mesh under hotspot tra c. For uniform tra c, Up*/Down* has the lowest saturation point (Sat_{min}) of the three routing algorithms, whereas the highest saturation point (Sat_{max}) is achieved for TOR and DOR for the mesh topology, and for DOR for the torus topology. Figure 5.16(a) indicates the three load levels selected for use in the main experiments. The *low*, *medium* and *high* load levels correspond to 90% of Sat_{min}, the center between Sat_{min} and Sat_{max}, and 110% of Sat_{max}, respectively. For hotspot tra c, the saturation point (Sat) is the same for all three routing algorithms (as the saturation point is mainly decided by the congestion

that results from 80% of the traffic being directed towards one of the processing nodes). The two load levels selected for use in the main experiments are indicated in Figure 5.16(b), where the *low* and *high* load levels correspond to 80% and 110%, respectively, of Sat.

5.2.3 Results

In the following, we present the results of our comparison of RecTOR with OR. First, the results of the experiments for the uniform traffic pattern are discussed. Second, we discuss the results of the experiments for the hotspot traffic pattern. A representative set of plots was selected and included.

Uniform traffic

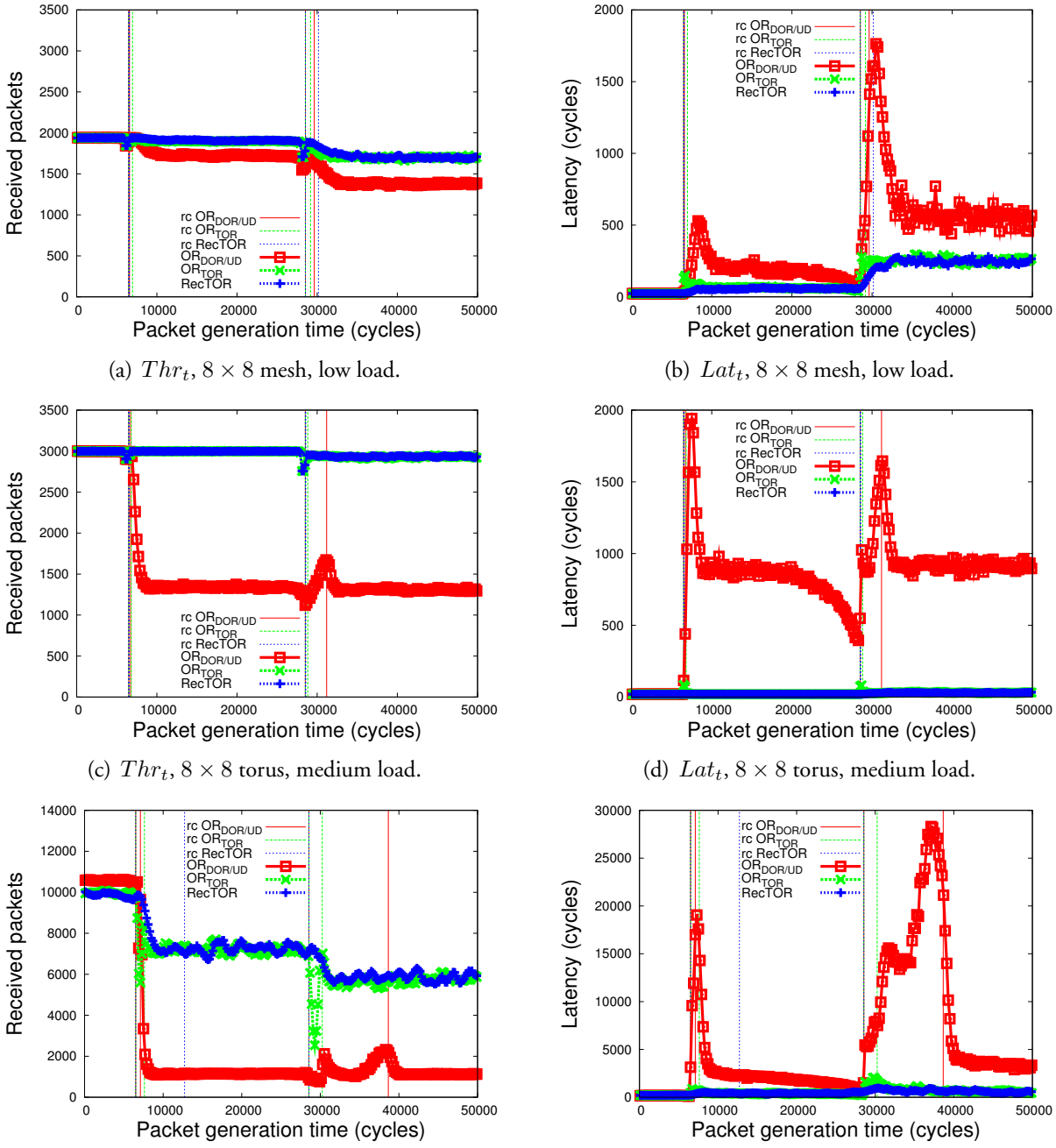
For the uniform traffic pattern, Figure 5.17 compares RecTOR with OR_{TOR} and $\text{OR}_{\text{DOR/UD}}$ for an 8×8 mesh under a low traffic load; an 8×8 torus under a medium traffic load; and a 16×16 torus under a high traffic load. The vertical lines indicate the start and end times of reconfiguration for each of the three cases (some of the lines are plotted on top of each other). The reduction of Thr_t seen at times 6 000 and 28 000 is due to packet loss in the faulty communication channel and switch, respectively (and is more pronounced for the low and medium load levels than for the highest load level where the network operates above its saturation point).

Figures 5.17(a), 5.17(c) and 5.17(e) show that, after the first reconfiguration, RecTOR and OR_{TOR} achieve a significantly higher Thr_t than $\text{OR}_{\text{DOR/UD}}$ does. Likewise, Figures 5.17(b), 5.17(d) and 5.17(f) show that, after the first reconfiguration, RecTOR and OR_{TOR} achieve a significantly lower Lat_t than $\text{OR}_{\text{DOR/UD}}$ does. The low load applied in Figures 5.17(a) and 5.17(b) is below the saturation point for the Up*/Down* routing algorithm in the fault-free case. Nevertheless, the fault of only a single communication channel causes a significant performance degradation for Up*/Down* when compared to TOR.

Using DOR for a fault-free mesh or torus – and relying on Up*/Down* to calculate new routes if a network component fails – is a probable arrangement (remember that DOR only works for fault-free meshes and tori). However, at least for a traffic pattern that resembles uniform, Figure 5.17 clearly demonstrates the drawbacks of such an approach. RecTOR assumes TOR, which not only achieves better performance than Up*/Down* after a fault has occurred, but also matches the performance of DOR in the fault-free case. Furthermore, using only one routing algorithm could simplify the implementation of a fault-tolerant interconnection network.

Recall that the deadlock avoidance mechanism of OR in many cases causes a decreased throughput and increased latency during a reconfiguration process (since a number of $\text{packets}_{\text{new}}$ are temporarily held back in the switches awaiting token propagation). For OR_{TOR}, Figure 5.17(e) provides examples of the characteristic troughs of the Thr_t curves, whereas the crests of the Lat_t curves are barely noticeable in Figure 5.17 due to the scale of these plots. Figure 5.18, on the other hand, clearly shows the advantages of RecTOR over OR_{TOR} in the case of uniform traffic. In order to better compare RecTOR with OR_{TOR}, $\text{OR}_{\text{DOR/UD}}$ is not included in these plots. (We have already concluded from Figure 5.17 that the performance of $\text{OR}_{\text{DOR/UD}}$ is inferior to the performance of RecTOR and OR_{TOR}.)

Figures 5.18(a) and 5.18(b) show Thr_t and Lat_t, respectively, for a 16×16 mesh under a low traffic load; Figures 5.18(c) and 5.18(d) show Thr_t and Lat_t, respectively, for a 16×16 torus under a medium traffic load; whereas Figures 5.18(e) and 5.18(f) show Thr_t and Lat_t, respectively, for



(a) Thr_t, 8×8 mesh, low load.

(b) Lat_t, 8×8 mesh, low load.

(c) Thr_t, 8×8 torus, medium load.

(d) Lat_t, 8×8 torus, medium load.

(e) Thr_t, 16×16 torus, high load.

(f) Lat_t, 16×16 torus, high load.

Figure 5.17: Thr_t, Lat_t, and start/end times of reconfiguration (*rc*) for RecTOR, OR_{TOR} and $\text{OR}_{\text{DOR/UD}}$ (uniform tra c, synchronized change-over to R_{new}).

an 8×8 torus under a high traffic load. These plots demonstrate that, in most cases, the Thr_t and Lat_t curves for OR_{TOR} have significant troughs and crests, respectively. For RecTOR, on the other hand, there are no troughs in the Thr_t curves nor crests in the Lat_t curves, as no restrictions are placed on the forwarding of packets during a reconfiguration process. The decreased throughput and increased latency observed for RecTOR after the occurrence of a component fault are merely due to the reduced capacity of the network when packets can no longer be forwarded across the faulty communication channel or switch. The same effect is naturally also observed for OR_{TOR}.

RecTOR performs equally well in the case of a less synchronized change-over ($change_{std} = 100$) from R_{old} to R_{new} as in the case of a fully synchronized change-over ($change_{std} = 0$). Therefore, Figure 5.18 includes only the less synchronized case for RecTOR, whereas both the fully and less synchronized cases are included for OR_{TOR}. Figures 5.18(e) and 5.18(f) show that, for a traffic load well above saturation, the performance of OR_{TOR} seems independent of how synchronized the change-over from R_{old} to R_{new} is. For low and medium traffic load, on the other hand, Figures 5.18(a) – 5.18(d) demonstrate that, for OR_{TOR}, the troughs of the Thr_t curves grow deeper and the crests of the Lat_t curves grow higher as the change-over from R_{old} to R_{new} gets less synchronized.

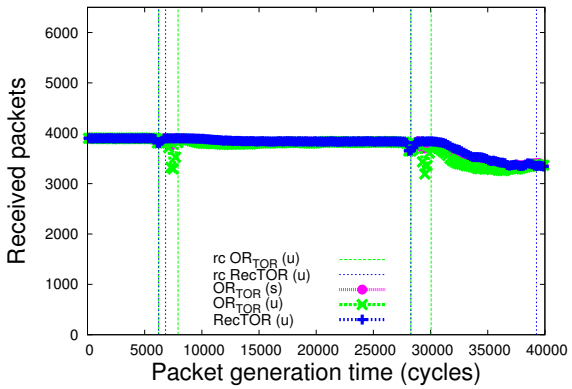
Our results show that during a reconfiguration process RecTOR provides a better network service than OR provides to a running application with a uniform communication pattern. Moreover, the advantages of using RecTOR become even more apparent if the change-over from R_{old} to R_{new} is not fully synchronized.

Hotspot traffic

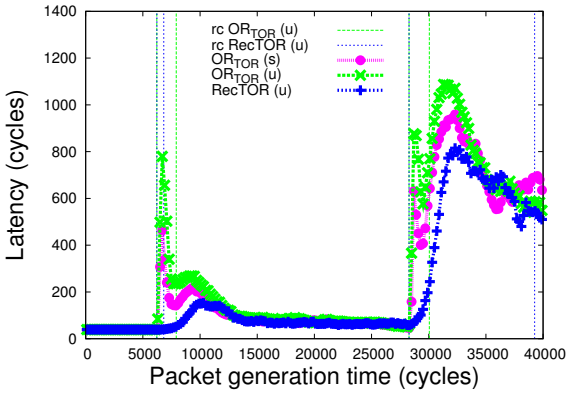
Figure 5.19 compares RecTOR with OR_{TOR} and $\text{OR}_{\text{DOR/UD}}$ for the hotspot traffic pattern. For a 16×16 torus under a low traffic load, Lat_t resulting from a fully synchronized and less synchronized change-over from R_{old} to R_{new} is shown in Figures 5.19(a) and 5.19(b), respectively. For a synchronized change-over from R_{old} to R_{new}, Figures 5.19(c) and 5.19(d) show Thr_t and Lat_t, respectively, for a 16×16 mesh under a low traffic load, whereas Figures 5.19(e) and 5.19(f) show Thr_t and Lat_t, respectively, for an 8×8 mesh under a high traffic load.

RecTOR guarantees that packets routed according to R_{old} and R_{new} can safely coexist in the network without causing deadlock. No restrictions are needed on packet forwarding during the transition between the two routing functions. Thus, for a low traffic load, the Lat_t curves for RecTOR are smooth, without any crests (see Figures 5.19(a), 5.19(b) and 5.19(d)). For OR_{TOR} and $\text{OR}_{\text{DOR/UD}}$, on the other hand, the crests of the Lat_t curves are in most cases pronounced. Furthermore, a comparison of Figure 5.19(a) with Figure 5.19(b) demonstrates that, for both OR_{TOR} and $\text{OR}_{\text{DOR/UD}}$, the heights of the crests of the Lat_t curves increase significantly as the change-over from R_{old} to R_{new} gets less synchronized. The performance of RecTOR is independent of how synchronized the transition from R_{old} to R_{new} is. Thus, for hotspot traffic – as for uniform traffic – the advantages of RecTOR over OR become even more apparent when the change-over from R_{old} to R_{new} is not fully synchronized.

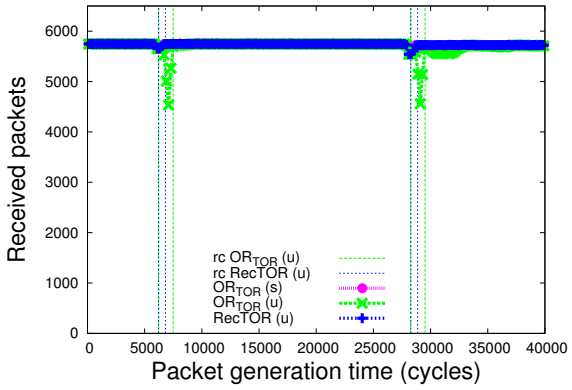
For a torus topology, Figures 5.19(a) and 5.19(b) indicate that, after the first reconfiguration, Lat_t for $\text{OR}_{\text{DOR/UD}}$ is higher than for RecTOR and OR_{TOR}. This is due to inferior performance of Up*/Down* when compared to TOR (which was also demonstrated for uniform traffic in Figure 5.17 – both for mesh and torus topologies). For a mesh topology, the stable latency levels



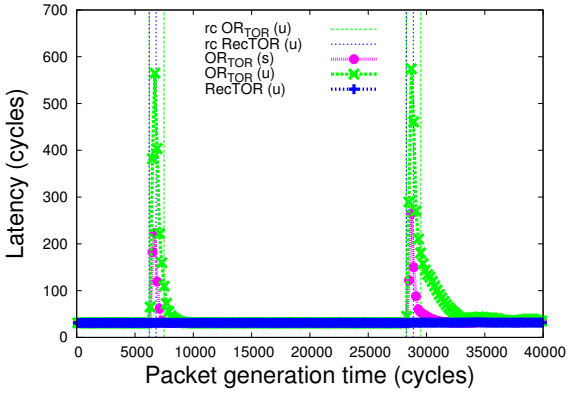
(a) Thr_t, 16 × 16 mesh, low load.



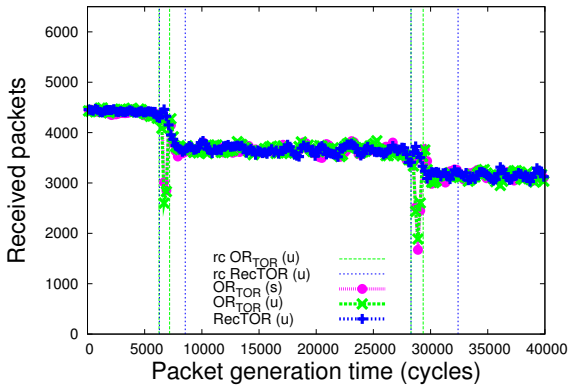
(b) Lat_t, 16 × 16 mesh, low load.



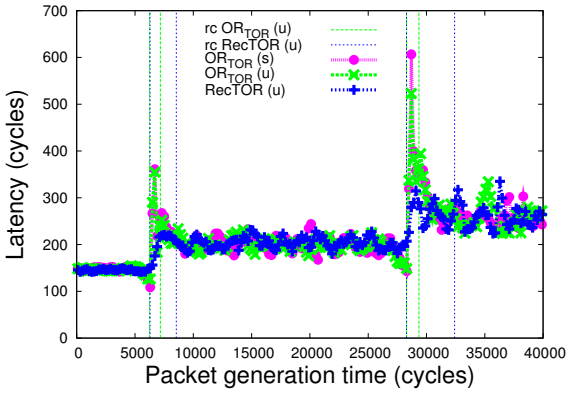
(c) Thr_t, 16 × 16 torus, medium load.



(d) Lat_t, 16 × 16 torus, medium load.



(e) Thr_t, 8 × 8 torus, high load.



(f) Lat_t, 8 × 8 torus, high load.

Figure 5.18: Thr_t, Lat_t, and start/end times of reconfiguration (*rc*) for RecTOR and OR$_{\text{TOR}}$ under synchronized (*s*) and unsynchronized (*u*) change-over to R_{new} (uniform tra c).

observed outside reconfiguration periods for the hotspot traffic pattern are similar for Up*/Down* and TOR (see Figure 5.19(d)). When based on a breadth-first search, the spanning tree of the Up*/Down* algorithm will typically be shallower and wider for a torus than for a mesh. Thus, for the torus, the area around the root switch to a larger extent becomes a traffic bottleneck. This could explain the differences observed for the mesh and torus topologies.

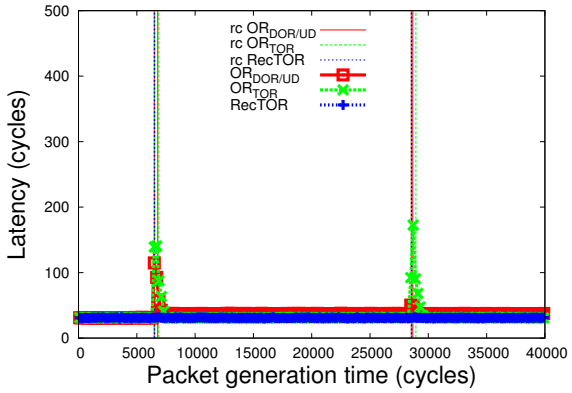
For hotspot traffic, the packet throughput is limited due to 80% of the traffic being directed towards one particular node. This explains the results for Thr_t presented in Figures 5.19(c) and 5.19(e). For a low traffic load, the Thr_t curves simply resemble horizontal lines (except that, for some plots not included here, falls related to packet loss in a faulty communication channel or switch are visible). For a high traffic load, the characteristic troughs in the Thr_t curves for OR are barely visible. Furthermore, we believe that the small increase in Thr_t immediately after the start of each reconfiguration in Figure 5.19(e) is due to more packets being accepted into the network when the new routes around a faulty communication channel or switch are taken into use. Figure 5.19(f) shows that such an increase in Thr_t corresponds to an increase in Lat_t for RecTOR as well as for $\text{OR}_{\text{DOR/UD}}$ and OR_{TOR}.

Figure 5.19(f) demonstrates that, for a load level well above saturation, RecTOR experiences fluctuations in Lat_t to a less extent than OR does. During a reconfiguration with OR, a number of $\text{packets}_{\text{new}}$ are held back in the switches awaiting token propagation. This explains the decrease in Lat_t for OR_{TOR} and $\text{OR}_{\text{DOR/UD}}$ observed for packets injected over a period of time before a reconfiguration. As a number of $\text{packets}_{\text{new}}$ are temporarily held back in the switches, a number of $\text{packets}_{\text{old}}$ experience lower latency due to reduced network load. However, this should not be interpreted as an advantage of OR over RecTOR. As $\text{packets}_{\text{new}}$ may be considerably delayed as a result of the token propagation procedure, some significant crests in the Lat_t curves are observed for OR_{TOR} and $\text{OR}_{\text{DOR/UD}}$. Figures 5.19(e) and 5.19(f) represent an extreme scenario – a heavy hotspot traffic pattern in combination with a workload well above saturation. However, even in this case, we conclude that RecTOR supports a more stable network service than OR does.

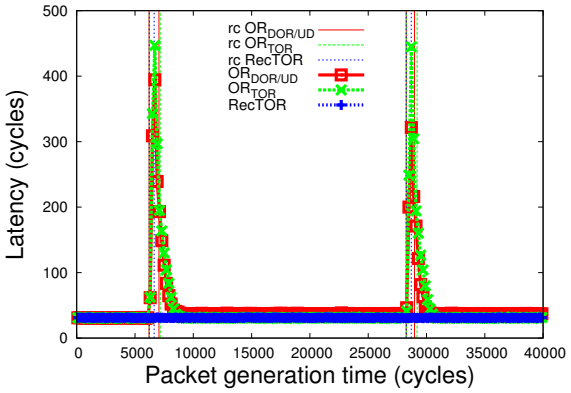
5.3 Related work and our contribution

A number of mechanisms are available in the literature for performing reconfiguration of an interconnection network, and an overview of the most relevant approaches was given in Section 2.8.2. Our contribution is to the area of general dynamic reconfiguration methods. Such methods allow application traffic into the network during a change-over from one routing function to another, and are not tied to any particular interconnection network technology. For our focus area the most relevant of the existing reconfiguration methods are Partial Progressive Reconfiguration (PPR) [36], Double Scheme [179], NetRec [19], Agent NetReconf [6,7], LORE [235], and Close Graph-based Reconfiguration (CGR) [185, 187]. Their manners of operation, including their main strengths and shortcomings, were discussed in Section 2.8.2.

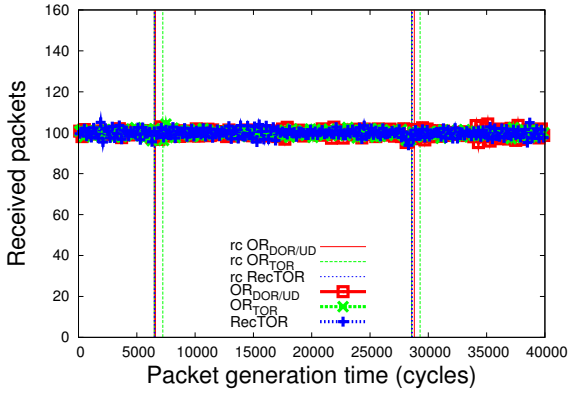
Recall that PPR, NetRec, Agent NetReconf, and LORE are all designed for distributed routing systems. A distributed routing system is more flexible with respect to packet routing than a source routing system is, since in the former intermediate switches can affect, and thus change, the path of a packet. This kind of flexibility is usually advantageous for the development of a general dynamic reconfiguration method, and could explain why more such methods are currently available for distributed routing systems than for source routing systems.



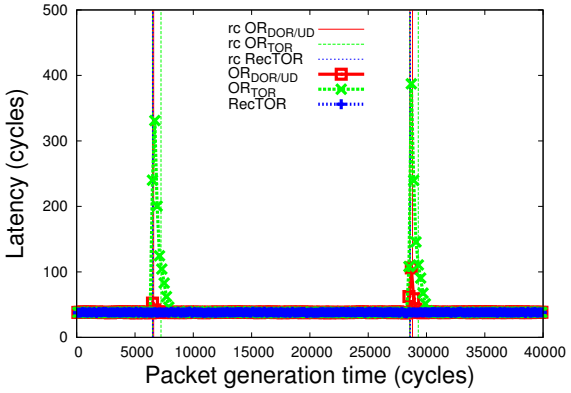
(a) Lat_t, 16×16 torus, low load, synchronized change-over to R_{new}.



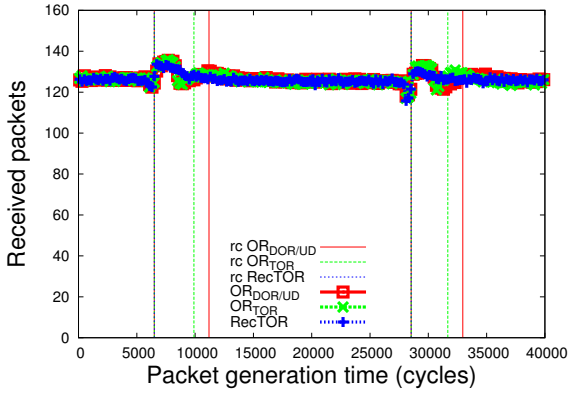
(b) Lat_t, 16×16 torus, low load, unsynchronized change-over to R_{new}.



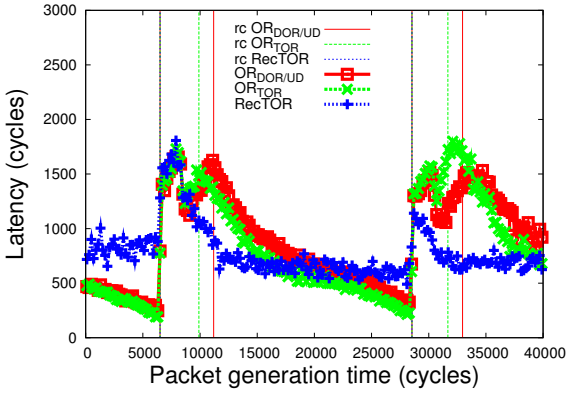
(c) Thr_t, 16×16 mesh, low load, synchronized change-over to R_{new}.



(d) Lat_t, 16×16 mesh, low load, synchronized change-over to R_{new}.



(e) Thr_t, 8×8 mesh, high load, synchronized change-over to R_{new}.



(f) Lat_t, 8×8 mesh, high load, synchronized change-over to R_{new}.

Figure 5.19: Thr_t, Lat_t, and start/end times of reconfiguration (*rc*) for RecTOR, OR$_{TOR}$ and OR$_{DOR/UD}$ (hotspot tra c).

CGR and Double Scheme are applicable for source routing systems as well as for distributed routing systems. (In [185] CGR was presented as a reconfiguration strategy for both source and distributed routing networks, whereas in [187] it was presented as a reconfiguration strategy for source routing networks.) CGR does not require VCs for deadlock avoidance. It depends on a relatively complicated procedure to create an R_{new} such that packets routed according to R_{old} and R_{new} can coexist in the network without causing deadlocks. Thus, CGR supports an unaffected network service during a reconfiguration process. However, CGR is tied to the Up*/Down* routing algorithm, which (as explained in Section 2.8.1) has performance issues, mainly related to the congestion that tends to form close to the root switch. Double Scheme does not assume a particular routing algorithm (or pair of routing algorithms). However, it depends on VCs for deadlock avoidance and does not support in-order delivery of packets during a reconfiguration process without heavy synchronization.

Up to now OR has been available only for distributed routing systems. However, OR possesses a number of qualities which are also attractive for source routing systems. Like Double Scheme, OR can be used between any pair of routing algorithms (and in [139] OR was shown to provide similar performance as Double Scheme). Furthermore, as opposed to Double Scheme, OR supports in-order packet delivery and does not require VCs. In Section 5.1, we proposed an adaptation that expands OR's area of application to include source routing systems. Moreover, we introduced an optimization of OR for source routing systems, and demonstrated how a relaxed token propagation procedure reduces performance penalties during a reconfiguration process.

As explained in Section 5.1.4, a complete synchronization of token injection – meaning that all traffic sources inject their tokens simultaneously – is hard to achieve. Our simulation results show that the degree of token injection synchronization has a significant impact on the increased latency, decreased throughput and congestion effects related to a reconfiguration process managed by OR. The less synchronized the injection of tokens is, the more significant such performance penalties become.

Our investigation of the effects of token injection synchronization is relevant for source routing systems where routing tables are kept in the processing nodes rather than in the switches. In [139] two different realizations of OR were evaluated for a distributed routing environment, and here we refer to these two implementations as OR_{v1} and OR_{v2}. For OR_{v1} the control packets that trigger token injection and the new routing tables are distributed in parallel from a central management entity. For OR_{v2}, on the other hand, the new routing tables are fully downloaded to the switches before the control packets that trigger token injection are distributed by the central management entity. When using OR_{v1}, traffic sources may inject their tokens before all switches have received complete new routing tables. This is a highly relevant synchronization issue for distributed routing systems using OR. As an aside, plots of [139] (showing traffic injected by processing nodes) indicate that the degree of synchronization of the token injection is similar for OR_{v1} and OR_{v2}.

In Section 5.2 we proposed RecTOR, a new general dynamic reconfiguration method which is applicable both for source and distributed routing systems. Like CGR, RecTOR allows the packets routed according to R_{old} and R_{new} to coexist in the network without restrictions, and does therefore not cause performance penalties during a reconfiguration process. However, whereas CGR depends on a relatively complicated procedure to establish R_{new}, RecTOR is based on the simple principle explained in Section 5.2.1. Furthermore, CGR assumes the Up*/Down* routing algorithm which achieves only limited performance. RecTOR, on the other hand, is based on the flexible TOR

routing algorithm which, given sound path selection, achieves excellent performance. In particular, as opposed to Up*/Down*, TOR supports shortest path routing and does not invite hotspots.

Like Double Scheme, RecTOR requires VCs and does not support in-order delivery of packets. Assume that $|VC_{old}|$ and $|VC_{new}|$ are the number of VCs needed by R_{old} and R_{new}, respectively, and that `int Max(int a, int b)` is a function that takes two integers as input and returns the one with the highest value. Then, whereas Double Scheme generally requires $|VC_{old}|+|VC_{new}|$ VCs, RecTOR does not require more VCs than `Max(`$|VC_{old}|$`,` $|VC_{new}|$`)`.

A comparison of the two reconfiguration methods focused in this chapter – OR and RecTOR – shows that there are advantages and disadvantages related to either of the methods. Whereas RecTOR cannot guarantee in-order delivery of packets during a reconfiguration process, OR supports in-order packet delivery (unless the least restrictive version of the optimization is used). Furthermore, OR can be used in systems where VCs are not available for the routing function, whereas RecTOR is based on TOR and depends on VCs for deadlock avoidance.

Our results show that, as RecTOR ensures deadlock-freedom without causing temporarily degraded throughput and latency, it supports a better network service during a reconfiguration process than OR does. Only in a best case scenario (where all the routes of R_{new} are also legal routes according to R_{old}), and only for a source routing system, can the optimized OR support an uninterrupted network service during a reconfiguration process. Furthermore, as opposed to the performance of OR, the performance of RecTOR is independent of how synchronized the change-over from R_{old} to R_{new} is.

OR can be used between any pair of routing algorithms, and is in this respect more flexible than RecTOR is. The choice of routing algorithm(s) is important for the performance of an interconnection network. Our results demonstrate how the performance degrades when – after having used the topology specific DOR algorithm for a fault-free mesh or torus topology – the topology agnostic Up*/Down* algorithm are taken into use for a non-regular topology that results from the fault of only a single communication channel. The topology agnostic TOR algorithm, on the other hand, could match the performance of DOR in a fault-free mesh and torus, and continue to support high performance in a non-regular topology that results from a fault. RecTOR is based on TOR, and – in addition to the advantages related to the performance of TOR – the use of only one routing algorithm could ease the implementation of a reconfiguration mechanism.

OR requires implementation of special functionality in the switches in order to govern the propagation of tokens throughout the network. RecTOR is simpler to realize as it does not require additional functionality in the network switches (except perhaps for preventing packets from looping in a distributed routing environment).

5.4 Critique

OR requires particular network switches as each switch must hold and process information regarding the reception and transmission of tokens. Our adaptation of OR for applicability in source routing systems assumes that each switch acquires the dependencies from its input to output channels while forwarding data packets. The acquisition of channel dependencies and registration of token propagation, as well as the proposed optimization of OR, demand additional logic in the switches. As source routing switches often are expected to be simple, the need for relatively complex switches may be a disadvantage for a reconfiguration scheme targeting a source routing envi-

ronment. Nevertheless, we believe in the usefulness of OR for such systems. OR has a number of desirable qualities, and the selection of general dynamic reconfiguration methods applicable for source routing systems is currently limited.

The optimization of OR proposed in Section 5.1.3 adds an overhead bit to each data packet, and requires additional logic in switches and processing nodes. In the case where reconfigurations are expected to occur infrequently, the implementation of the optimization might not be justified. However, our results demonstrate that congestion induced by a reconfiguration process may cause network instability for a significant period of time following the reconfiguration. Preventing or alleviating such congestion may motivate the use of the optimization even in cases where reconfigurations do not occur frequently. The least restrictive version of the optimization of OR cannot guarantee in-order packet delivery. However, this version of the optimization in general has the highest improvement potential. Thus, the decision on whether to apply the optimization – and, if so, which version to use – involves a tradeo between acceptable overhead, reconfiguration frequency, required performance, and need for in-order packet delivery.

As OR does not depend on VCs for deadlock avoidance, and as a routing algorithm capable of handling non-regular topologies was required, the well established Up*/Down* algorithm was chosen for the performance evaluation of OR in Section 5.1.6. However, Up*/Down* achieves relatively low performance, and our choice of routing algorithm could perhaps therefore be criticized. On the other hand, the results of the experiments in Section 5.2.3 – where OR was also used with the TOR algorithm – mainly confirm the behaviour of OR observed when using Up*/Down*.

RecTOR is in several respects less versatile than OR is. It is tied to a specific routing algorithm – TOR; depends on the availability of VCs; cannot ensure in-order packet delivery; and may require mechanisms to prevent packets from looping in a distributed routing environment. On the other hand, TOR supports excellent performance; the use of a single routing algorithm could ease the implementation of a fault-tolerant interconnection network; RecTOR does not require complex network switches; and – perhaps most importantly – RecTOR does not cause degraded performance during a reconfiguration process. Thus, provided that a system has VCs available for the routing function and that out-of-order packet delivery is acceptable, RecTOR may be an attractive general dynamic reconfiguration method.

5.5 Future work

Our results indicate that, for tra c patterns close to uniform, the benefit of the optimization of OR is in some cases limited by head-of-line blocking e ects that result from the token propagation procedure. A future study could aim at improving the gain of the optimization for such tra c patterns, e.g. by introducing virtual output queues.

In our experiments, time is represented by an abstract unit called cycles. For an interconnection network, the duration of a reconfiguration process might be an important parameter. This parameter is probably more important when using OR than when using RecTOR, however, as temporary performance penalties must be expected as a result of a reconfiguration process managed by OR. A future study could address the real-time behaviour of reconfiguration strategies, either by applying a more detailed simulator model or by implementing the reconfiguration strategies in a real system and performing measurements.

Like RecTOR, the fault-tolerant routing algorithm proposed in [154] – referred to as Transition-

Based Fault-Tolerant Routing (TFTR) – is based on the TOR routing algorithm. TFTR assumes the InfiniBand interconnection network technology [105] and uses TOR to compute a set of disjoint paths for each pair of source and destination processing nodes in a system. If, during the operation of the system, the primary path is disrupted by a fault, the Automatic Path Migration hardware mechanism of InfiniBand supports a transition to an alternate path. The minimum number of faults tolerated by TFTR corresponds to the lowest number of disjoint paths calculated for any pair of processing nodes. As indicated in [154], a reconfiguration is needed when, for some pair of processing nodes, all of the calculated paths have been disrupted by component faults. Since TOR depends on VC-transitions for deadlock avoidance, using TOR (as well as RecTOR or TFTR) for interconnection networks based on the InfiniBand technology is not unproblematic. Recall from Section 4.3 that in InfiniBand – where VCs are called virtual lanes (VLs) – each packet carries a particular service level (SL) which cannot be changed by intermediate switches. The number of SLs are limited in InfiniBand, and the VL to use on each communication channel is decided by an SL to VL mapping table in each switch. Due to these factors VL-transitions cannot be used unrestrictedly in InfiniBand. TFTR may require a high number of SLs, and in [153] an alternative approach – which demands a low number of SLs and VLs – was proposed for calculating a set of disjoint paths in torus networks based on InfiniBand. However, for interconnection network technologies that support more flexible mechanisms for VC-transitions than InfiniBand does (and also support migration of paths), a future study could aim at combining RecTOR with the main principles of TFTR in order to provide an even more powerful fault tolerance mechanism than RecTOR is alone.

Chapter 6

Processor allocation

Processor allocation is a form of virtualization where sets of physical processing nodes are aggregated into virtual processors and assigned to execute parallel jobs. A set of processing nodes – often referred to as a partition – typically includes only a subset of the processing nodes available in a multiprocessor system such as a computer cluster, data center or multicore chip. In order to achieve a reasonable utilization of the resources available in a multiprocessor system, an e cient processor allocation strategy is important. Furthermore, for the parallel jobs to be executed in a multiprocessor system, the processor allocation strategy has an influence on such factors as running time, queuing time and interference between concurrently running jobs.

Recall from Section 2.7 that *tra c-containment* is a quality of some contiguous processor allocation strategies that prevents concurrently running jobs from sharing communication channels. Isolation of intra-job tra c within each partition may be advantageous both for performance, anonymity and security reasons. Such isolation ensures that the communication overhead for a job is independent of concurrently running jobs, and that one job cannot analyze nor tamper with the tra c belonging to another job. Unfortunately, fragmentation – and thus a suboptimal utilization of a system's processing resources – is a consequence of the enforcement of tra c-containment. Many of the processor allocation strategies found in the literature, such as those that allocate submesh shaped partitions in mesh topologies (see Section 2.7.1), are rigid approaches which achieve tra c-containment only at the cost of a significantly reduced resource utilization. The research presented in this chapter aims at providing more flexible processor allocation strategies which are able to ensure tra c-containment as well as a high resource utilization. For development of such flexible allocation strategies, we assume a close collaboration between the sub-system responsible for processor allocation and the sub-system responsible for routing. Our new approaches to processor allocation depend on topology agnostic routing, and in some cases reconfiguration is also needed.

Apart from the collaboration with the routing sub-system, we assume a traditional model of a resource management sub-system where jobs arrive in a waiting queue, and each job has requirements on such aspects as number of processing nodes and running time (see Section 2.7 and Figure 2.5). A scheduler decides the sequence in which the queued jobs are selected, and an allocator attempts to locate and assign a set of free processing nodes that meets the requirements of the selected job. The set of processing nodes is exclusively assigned to a single parallel job, which runs uninterruptedly until completion.

In Section 6.1 we present a novel tra c-contained processor allocation algorithm, UDFlex, which is based on the topology agnostic Up*/Down* routing algorithm [204]. UDFlex may be

used on any topology, and the only restriction on an allocated partition is that the set of nodes must constitute a sub-graph of the overall Up*/Down* graph. This provides increased flexibility and potentially reduced fragmentation when compared to the algorithms that restrict the allocated partitions to specific shapes. Section 6.1 is based on [225].

In Section 6.2 we introduce a new framework which combines flexible processor allocation, topology agnostic routing and reconfiguration in order to achieve both a high resource utilization (low fragmentation) and tra c-containment. The framework is conceptually simple and does not prescribe a particular processor allocation strategy, routing algorithm or reconfiguration method. Section 6.2 is based on [223].

After the presentation of our research, Section 6.3 discusses related work and our contribution, whereas Section 6.4 presents a critical view of our own work. Section 6.5 considers possibilities for future research and is based on [142].

Throughout the chapter, in order to simplify the discussion, we assume that some entity referred to as the *routing module* is responsible for maintaining the routing function. We also assume that the assignment of processing resources is performed sequentially, meaning that an allocator only handles a single request for resources at a time. Furthermore, we assume that a *node* refers to a combined node which contains both processing and switching elements. A free node is a node which is currently not allocated to any job. A busy node, on the other hand, is a node which is currently allocated to a job.

6.1 Tra c-contained allocation in an Up*/Down* routed system

In this section we present a novel processor allocation strategy, UDFlex, which may alleviate the fragmentation problem inherent in contiguous processor allocation algorithms by being flexible with respect to the allowed shape of partitions. This means that UDFlex has the potential of achieving a high utilization of a system's processing resources. Moreover, UDFlex ensures tra c-containment within each partition, and thereby prevents interference between the tra c belonging to concurrently running jobs. Section 6.1.1 presents the basic UDFlex algorithm, whereas Section 6.1.2 discusses an example implementation. The performance of UDFlex is evaluated in Sections 6.1.3 and 6.1.4.

6.1.1 The UDFlex algorithm

UDFlex assumes a close collaboration between an allocator and a routing module, and it assumes that the interconnection network applies the Up*/Down* routing algorithm [204]. Recall that Up*/Down* assigns up and down directions to all the communication channels in the network to form a directed acyclic graph (DAG) rooted in one of the nodes, and that deadlock is avoided by prohibiting the turn from a down-link to an up-link.

We include definitions of a *correct* Up*/Down* graph, and a *valid* sub-graph of an Up*/Down* graph as both are important concepts for the UDFlex algorithm:

Definition 6.1. A correct Up*/Down* graph is a DAG with only one root node, where all other nodes can reach the root by following a sequence of one or more up-links, and the root can reach

all other nodes by following a sequence of one or more down-links.

Definition 6.2. A valid sub-graph of an Up*/Down* graph G consists of a subset of the nodes and communication channels of G, and is itself a correct Up*/Down* graph.

Assume that an Up*/Down* graph has been constructed for an interconnection network that connects a number of nodes. For simplicity, we also assume a stable topology during the period of interest, such that changes in the Up*/Down* graph are not needed. The main idea behind UDFlex is to allocate to each job a set of free nodes that form a valid Up*/Down* sub-graph. The assignment of a valid Up*/Down* sub-graph to a scheduled job presupposes that the allocator has received information about the Up*/Down* graph from the routing module.

UDFlex has several attractive qualities. The allocation of partitions that constitute valid Up*/Down* sub-graphs allows flexibility with regard to the shape of partitions, and may thereby cause low fragmentation. It also supports containment of tra c within each partition. However, in order to ensure that no packet is routed outside its partition, the routing module needs information from the allocator about a new partition (or at least about the boundaries of a new partition). UDFlex can recognize a free Up*/Down* sub-graph given that one is available. (A free sub-graph is a valid sub-graph of an Up*/Down* graph where none of the nodes are currently busy.) Any number of nodes can constitute a partition to be allocated to a scheduled job, which means that internal fragmentation is completely avoided. Moreover, UDFlex is a topology agnostic algorithm, and is thus also useful for a non-regular topology that may result from faults in a regular topology.

The UDFlex allocation algorithm is presented next. Assume that the allocator and the routing module cooperate in running UDFlex; that the routing module has calculated an Up*/Down* graph G for the topology; that the allocator knows G; and that the routing module knows all possible paths between every pair of source and destination nodes in G and can activate or deactivate paths as required. Further assume that a parallel job just selected by the scheduler requests a number of nodes $|R|$.

1. The number of free nodes, $|N_{free}|$, of G is checked. The request can only be granted if $|N_{free}| \geq |R|$. If $|N_{free}| < |R|$, the scheduled job is returned to the waiting queue, pending termination of a running job.

2. A valid sub-graph P of G consisting of exactly $|R|$ free nodes is identified and reserved for the scheduled job.

3. The allocator notifies the routing module of the new partition P.

4. For every pair of nodes inside P, the routing module must ensure that any path in G that traverses an intermediate node outside P is deactivated, and that at least one path in G that does not traverse an intermediate node outside P is activated.

5. The routing module notifies the allocator that the routing function is ready.

6. The allocator assigns P to the scheduled job.

The status of the nodes that are part of P are typically changed from free to busy during step 2 or step 6 of the UDFlex allocation algorithm. However, in order to ensure tra c-containment and

connectivity, the scheduled parallel job is not started on P until step 6 has completed. Note that if a scheduled job is returned to the waiting queue (which may happen during step 1 or step 2), the remaining algorithmic steps are skipped, and the job must be rescheduled.

In order to establish that UDFlex guarantees isolation of the tra c belonging to di erent parallel jobs, we include and prove Lemma 6.1.

Lemma 6.1. *UDFlex ensures tra c-containment within each partition.*

Proof. The proof is by contradiction. Assume that an arbitrary node s and another arbitrary node d are both part of a partition P, and that underway from its source s to its destination d a packet crosses an intermediate node i which is not part of P. This contradicts step 4 of the UDFlex algorithm which prescribes that, for every pair of nodes inside P, any path in G that traverses an i outside P is deactivated. Thus, for a job running on P, no packet from s to d can leave P. □

Lemma 6.1 verifies that, when using UDFlex, packets passed between a set of nodes assigned to run a parallel job cannot be routed through any node that is not a member of the set. Next, in order to establish that UDFlex enables communication between all the members of the set of nodes assigned to run a parallel job, we include and prove Lemma 6.2.

Lemma 6.2. *UDFlex provides a connected routing function within each partition.*

Proof. The proof is by contradiction. Assume that an arbitrary node s and another arbitrary node d are both part of a partition P, and that no path exists within P from s to d. Let a *legal path* denote a path that obeys the turn-restrictions of G. We then have to consider the following two alternative situations:

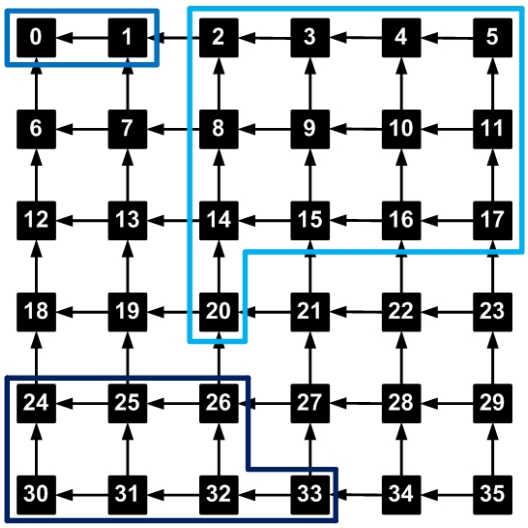
1. No legal path exists within P from s to d. This contradicts step 2 of the UDFlex algorithm which demands that P is a valid sub-graph of G. According to Definition 6.2, a valid sub-graph of G is itself a correct Up*/Down* graph. Thus, according to Definition 6.1, at least one path that crosses zero or more up-links before crossing zero or more down-links must exist within P from s to d.

2. One or more legal paths exist within P from s to d but are deactivated. This contradicts step 4 of the UDFlex algorithm which prescribes that, for every pair of nodes inside P, at least one path in G that does not traverse an intermediate node outside P is activated.

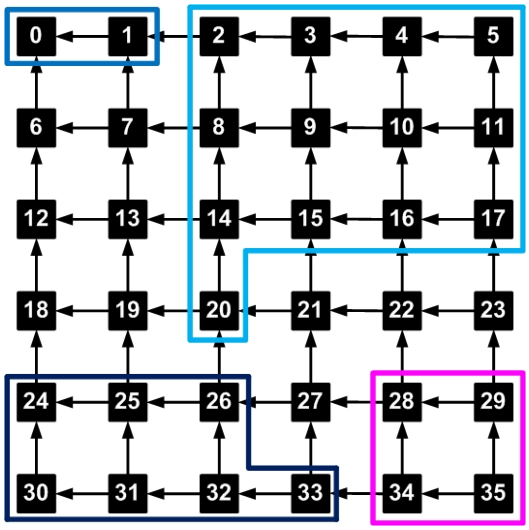
□

When a parallel job has finished running on a partition P, the UDFlex deallocation algorithm simply changes the status of each of the nodes which are part of P from busy to free. In the following, "UDFlex" or "the UDFlex algorithm" refers to the allocation algorithm, rather than the deallocation algorithm, unless otherwise stated.

Figure 6.1 illustrates UDFlex allocation in a 6×6 mesh multiprocessor system. A configuration where three parallel jobs are running on partitions of sizes 2, 7 and 13 nodes is displayed in Figure 6.1(a). Immediately afterwards, a partition of 4 nodes is assigned to a new job, and the new configuration is displayed in Figure 6.1(b). Notice how each of the partitions is a valid sub-graph of the overall Up*/Down* graph. Also notice how the latest assignment is one of several possible



(a) Snapshot of running jobs immediately before granting a request for 4 nodes.



(b) Snapshot of running jobs immediately after granting a request for 4 nodes.

Figure 6.1: Example of UDFlex allocation in a 6×6 mesh multiprocessor system. (An arrow indicates the direction of an up-link.)

placements of a 4-node job. Assuming minimal routing, for three of the partitions deactivation of paths is not required in order to ensure tra c-containment. However, for the 7-node partition, packets originating from node 33 and destined for node 24, 25 or 26 may be routed through node 27 (which is not part of the partition), unless the routing module has deactivated this path. (The corresponding return path must also be deactivated.)

6.1.2 An example implementation

In this section, we include and discuss a particular implementation of UDFlex that is also used in the performance evaluation presented in Sections 6.1.3 and 6.1.4. As previously explained, UDFlex presupposes a cooperation between an allocator and a routing module. However, the main focus of this chapter is on the operation of the allocator, rather than on the operation of the routing module. This section mainly aims to provide an example of a realization of step 2 of the UDFlex algorithm which determines that the partition reserved for a scheduled job must constitute a valid sub-graph of an Up*/Down* graph. However, towards the end of this section we will also include comments pertaining to step 4 of the UDFlex algorithm which provides directions for the activation and deactivation of paths necessary to ensure connectivity and tra c-containment within a partition. We will not consider further details related to the communication between the allocator and the routing module.

Recall that a free sub-graph is a valid Up*/Down* sub-graph where none of the nodes are currently busy. Assume that the allocator and the routing module cooperate in running UDFlex; that the routing module has calculated an Up*/Down* graph G for the topology; that the allocator knows G; that the routing module knows all possible paths between every pair of source and destination nodes in G and can activate or deactivate paths as required; that the distance (number of hops) from the root of G to every other node in G is known; and that the number of nodes in some free sub-graph x of G is denoted $|x|$. Further assume that a parallel job just selected by the

scheduler requests a number of nodes $|R|$. Then remember that step 2 of the UDFlex algorithm is as follows: *A valid sub-graph P of G consisting of exactly $|R|$ free nodes is identified and reserved for the scheduled job*.

In many cases, several candidates for P may exist. (Recall from the example illustrated in Figure 6.1 that the latest partition allocated was one of several alternative placements of a 4-node job.) This gives room for a variety of implementations, ranging from simple "first-fit" approaches (where the first free sub-graph encountered is selected) to more sophisticated "best-fit" approaches (where measures are typically taken to increase the probability of successful accommodation of subsequent jobs). Our example implementation leans towards the latter category of approaches.

The primary aim of our implementation is to place P in the smallest possible free sub-graph. (This free sub-graph may itself be part of a larger free sub-graph.) The motivation is to encourage defragmentation of the set of nodes that remains free after the allocation, in order to increase the chance of success for later allocation attempts. In order to minimize dispersion within an allocated set of nodes, our implementation first searches for a *complete* free sub-graph. For a complete free sub-graph a rooted in a node a_r, no other free sub-graph a' rooted in a_r exists where a is a subset of a'. In Figure 6.1(a) the 5-node free sub-graph consisting of nodes $27, 28, 29, 34, 35$ and rooted in node 27 is complete. On the other hand, the 3-node free sub-graph consisting of nodes $27, 28, 34$, which is also rooted in node 27, is not complete.

The secondary aim – which applies for instance if several equally small free sub-graphs exist – is to place P in the candidate sub-graph that has its root node farthest away from the root node of G. The motivation is to keep the area around the root node of G as free and defragmented as possible in order to promote a successful allocation of any subsequent job requesting a high number of nodes.

The allocation of the 4-node partition illustrated in Figure 6.1 is in accordance with our example implementation. In Figure 6.1(a), two complete free 4-node sub-graphs satisfy our primary aim. One of the sub-graphs, $sg1$, consists of the nodes $12, 13, 18, 19$ and the other, $sg2$, consists of the nodes $28, 29, 34, 35$. In order to satisfy our secondary aim, $sg2$ was chosen and assigned to the scheduled job. An alternative implementation could have prioritized di erent aims. Assume that H refers to a free sub-graph in G that is not itself part of a larger free sub-graph in G. Then, for instance, an alternative secondary aim could have been to place P in the candidate sub-graph inside the smaller H. Returning to our example from Figure 6.1, we observe that $sg1$ is part of a 6-node H, whereas $sg2$ is part of an 8-node H. Thus, with an alternative secondary aim, $sg1$ could have been chosen and assigned to the scheduled job.

Our example implementation of step 2 of the UDFlex algorithm is presented in the following.

1. Let an H_r be the root node of an H. An H_r is then any free node that does not have an up-link towards some free node in G. Identify every H_r and store the resulting set of nodes in S_{H_r}.

2. Search every H rooted in an $H_r \in S_{H_r}$ and identify the smallest complete free sub-graph g of any H where $|g| \geq |R|$. See the primary aim above.

    - If no g was found, the scheduled job must be returned to the waiting queue, pending termination of a running job. (The remaining steps of the UDFlex algorithm are

skipped, and the job must be rescheduled.) This is an example of external fragmentation (step 1 of the UDFlex algorithm already checked that a su cient number of nodes in G were free).

- Else if more than one candidate was found for g, let g be the candidate that has its root farthest away from the root of G (if there are still several candidates, a random selection is performed). See the secondary aim above.

3. For the scheduled job, reserve g or a valid Up*/Down* sub-graph of g that contains exactly $|R|$ nodes.

   - If $|g| = |R|$ then reserve g for the job.
   - Else if $|g| > |R|$ then the redundant number of nodes, $|g| - |R|$, must be excluded from the set of nodes reserved for the job (in order to avoid internal fragmentation). Identify the smallest valid Up*/Down* sub-graph g' of g where $|g'| \geq (|g| - |R|)$. Then g' contains the $|g| - |R|$ nodes to be excluded from the reservation. If more than one candidate was found for g', let g' be the candidate that has its root node nearest to the root node of G. (Recall that we attempt to keep the area close to the root of G as free and defragmented as possible.)
     - If $|g'| = (|g| - |R|)$ then reserve $g - g'$ for the job. The purpose of the exclusion of a valid Up*/Down* sub-graph, g', from the reservation (rather than any arbitrary, possibly fragmented, $|g| - |R|$ nodes) is to increase the probability of a successful accommodation of a subsequent job.
     - Else if $|g'| > (|g| - |R|)$ then apply a breadth-first search from the root of g' in order to identify a valid Up*/Down* sub-graph, g'', where $|g''| = |g'| - (|g| - |R|)$, and reserve $g - g' + g''$ for the job. This approach is chosen for simplicity, and may scatter the redundant $|g| - |R|$ nodes.

Assume that the Up*/Down* graph calculated by the routing module can be expressed as $G = (V, E)$, where V is the set of vertexes representing the nodes and E is the set of edges representing the communication channels of the interconnection network. The numbers of vertexes and edges are $|V|$ and $|E|$, respectively. In the following, the complexity of our example implementation of step 2 of the UDFlex algorithm is roughly estimated.

Step 1 of the example implementation involves identification of every H_r in the system. The complexity of this step is $O(|V| \times |p|)$ where $|p|$ is the number of ports per node.

Step 2 of the example implementation, which identifies the complete free sub-graph g where the new partition is to be located, involves a depth-first search from every H_r. The complexity of a depth-first search on G is $O(|E| + |V|)$ [248], and the complexity of Step 2 becomes $O(|V| \times (|E| + |V|))$.

Step 3 of the example implementation involves selection and reservation of exactly $|R|$ nodes for the scheduled job. Depending on the size and shape of g, alternative procedures will be executed. A breadth-first search constitutes the most complex part of the most expensive procedure. Thus, the complexity of Step 3 becomes $O(|E| + |V|)$ [248].

Let us consider two-dimensional mesh and torus topologies as examples since these topologies are used in the performance evaluation presented in Sections 6.1.3 and 6.1.4. For a two-dimensional torus $|p| = 4$ and $|E| = 2 \times |V|$, whereas for a two-dimensional mesh $|p| \leq 4$ and

$|E| < 2 \times |V|$. Thus, for such topologies, the complexity of steps 1, 2 and 3 of the example implementation can roughly be expressed as $O(|V|)$, $O(|V|^2)$ and $O(|V|)$, respectively, which may collectively be expressed as $O(|V|^2)$.

The routing module could precalculate all alternative paths between every pair of source and destination nodes in a system. Step 4 of the UDFlex algorithm states that the routing module must activate and deactivate paths in order to ensure tra c-containment within a partition as well as connectivity between every pair of nodes in the partition. The complexity of such activation and deactivation of paths will typically be at least $O(|V|^2)$ as it must be performed for every pair of nodes in a partition. For the sake of tra c-containment and connectivity a scheduled job cannot be started until the updated routing function is ready. However, we notice that starting the job earlier would not have caused deadlock as the routing functions in use before and after the update are based on the same Up*/Down* graph and are therefore compatible. In this study we have not explicitly simulated packet-exchange between nodes, and we will not further discuss implementation details concerning the routing module's activation and deactivation of paths.

Finally, let us consider the complexity of the UDFlex deallocation algorithm. As we recall, UDFlex deallocation simply implies a traversal through the nodes assigned to a finishing job to change their status from busy to free. Thus, the complexity of UDFlex deallocation becomes $O(|V|)$.

6.1.3 Experiment setup

In order to evaluate the performance of UDFlex, traditional contiguous allocation strategies are used as our main points of reference. We do, however, also include the Random [137] allocation strategy in our experiments. Recall from Section 2.8.3 that Random is an allocation strategy that fails only when the next job to be served requires a higher number of nodes than the number of currently free nodes. Thus, fragmentation is not an issue when using Random. The reason for including Random is to visualize the upper performance benchmark with respect to our metrics (as long as communication overhead is not considered). For practical purposes, communication overhead may reduce the attractiveness of Random.

We consider both mesh and torus topologies of size 16×16, 32×16, and 32×32. In these experiments, UDFlex assumes an Up*/Down* graph based on a tree identified by a breadth-first search (as proposed in [204]). The root of the Up*/Down* graph is in the upper left corner[1] of the topology.

For meshes, we compare UDFlex with the contiguous allocation algorithms First Fit [262], Best Fit [262], Adaptive Scan [61] and Flexfold [91]. All of these allocation strategies are described in Section 2.7.1.

For tori, we compare UDFlex with a contiguous strategy that allocates sub-meshes (possibly across wraparound channels) and has complete sub-mesh recognition capability. This allocation strategy is based on Adaptive Scan and fitted to torus topologies. First, allocation of the originally requested sub-mesh $a \times b$ is attempted, and, if no such sub-mesh is available, the 90 degrees rotation of the original request ($b \times a$) is attempted. We believe that for the metrics considered this strategy is a reasonable representative of algorithms that allocate sub-meshes in two-dimensional tori. As with

[1] In order to simplify the discussion, we visualize a torus topology laid out in a plane as a mesh topology with wraparound channels.

most of the allocation strategies for k-ary n-cubes included in Section 2.7.1, this strategy is not tra c-contained when Dimension-Order Routing (DOR) [231] is used. Nevertheless, for the evaluation of UDFlex with respect to a metric such as system utilization, we believe that a comparison with this strategy is more interesting than a comparison with a strategy such as k-ary Partner [251]. Recall from Section 2.7.1 that the k-ary Partner strategy supports tra c-containment, but does not have complete sub-cube recognition capability and may also be a ected by internal fragmentation.

For every experiment, data collection spans the execution of 16 000 jobs. In order to evaluate the performance of the various allocation algorithms we consider the metrics *system utilization* and *queuing time*. Assume that a mesh or torus has width w and height h; that the position of any node can be expressed as a pair of coordinates (x, y) where $1 \leq x \leq w$ and $1 \leq y \leq h$; that the node in position (i, j) has been busy for the aggregated time $busy_{i,j}$; and that data have been collected for a period of time T. Then, as in [262], we define the system utilization to be $\frac{\sum_{1 \leq i \leq w, 1 \leq j \leq h} busy_{i,j}}{w \times h \times T}$, and the *system fragmentation* is then $1 -$ *system utilization*. The queuing time is the average time that a job is held in the waiting queue, from the time of the arrival of a job until its requested nodes have been allocated. Each experiment is repeated 16 times (with a di erent seed), and the mean values of these repetitions are presented together with their 95% confidence intervals.

For a fair comparison between algorithms that allocate sub-meshes and those that do not, the resource demand of jobs (the number of nodes requested) should be equal for either group. Each job requests $a \times b$ nodes, which is considered a product of numbers by UDFlex and Random and a defined sub-mesh by the remaining allocation strategies. Recall from Section 3.1.2 that the values of a and b are drawn from separate uniform distributions with minimum sizes a_{min} and b_{min} and maximum sizes a_{max} and b_{max}, respectively. We conducted experiments for jobs with a potential for high, medium, and low resource demand. For all of the experiments $a_{min} = 1$ and $b_{min} = 1$. For a potentially high resource demand a_{max} is set to w and b_{max} is set to h; for a potentially medium resource demand a_{max} is set to $\frac{w}{2}$ and b_{max} is set to $\frac{h}{2}$; and for a low resource demand a_{max} is set to $\frac{w}{4}$ and b_{max} is set to $\frac{h}{4}$.

Recall from Section 3.1.2 that for a mesh or torus of width w and height h the input *load* is $\frac{|R|_{mean} \times ST_{mean}}{w \times h \times IT_{mean}}$, where $|R|_{mean}$ is the mean number of nodes requested by the jobs; ST_{mean} is the mean service time (running time) of jobs; and IT_{mean} is the mean interarrival time of jobs.

Our first set of experiments aims to assess the ability of UDFlex to reduce fragmentation, when compared to strategies that allocate sub-meshes. We simulated input load levels ranging from 0.1 to 1.1, where a load higher than 1 indicates that the arrival rate of jobs exceeds the departure rate. The desired range of input load levels originates from variation of ST_{mean} or IT_{mean}, as for a particular group of experiments w, h and $|R|_{mean}$ are given. Two subsets of experiments were conducted. In one subset, IT_{mean} was varied while ST_{mean} was fixed at 1 000 cycles. In the other subset (which was only performed for jobs with a potentially high resource demand), ST_{mean} was varied while IT_{mean} was fixed at 500 cycles.

For several reasons, the allocation and communication overhead may be higher for UDFlex than for the strategies that allocate sub-meshes. In general, UDFlex has a higher complexity than the strategies that allocate sub-meshes have. For instance, for two-dimensional meshes, the complexity of UDFlex allocation is roughly $O(|V|^2)$ (perhaps even higher when considering path activation and deactivation), whereas both First Fit and Best Fit have complexity $\Theta(w \times h)$ [262], which can

also be written as $\Theta(|V|)$ or $O(|V|)$. Moreover, the Up*/Down* routing algorithm has performance issues related to congestion that tends to form close to the root node. Thus, when using UDFlex, rather than allocating sub-meshes and using DOR, a potentially higher packet latency may increase the service time of jobs.

Up*/Down* imposes such restrictions on packet forwarding that the legal path between a pair of nodes may be longer than the shortest physical path. In addition, UDFlex allows allocation of irregularly shaped partitions. Thus, when compared to sub-mesh allocation and the use of DOR, the use of UDFlex may alter the distance between nodes. If the packet latency grows, a likely result is an increased service time of jobs. However, for an interconnection network that uses cut-through switching [114] a small increase in the number of hops between nodes is expected to have a limited impact on packet latency.

Although alleviating fragmentation is the primary objective of this study, the e ect of the potential allocation and communication overhead of UDFlex should also be considered, and this is the focus of our second set of experiments. To anticipate the course of events, the results from our first set of experiments confirmed that, when compared to using strategies that allocate sub-meshes, using UDFlex alleviates fragmentation. The aim of the second set of experiments is to quantify the amount of allocation and communication overhead that can be tolerated before the advantage of UDFlex with respect to fragmentation is neutralized. We compare UDFlex with Adaptive Scan, Best Fit and First Fit, three tra c-contained allocation strategies which allocate sub-meshes without manipulating the shape of the requested sub-meshes.[2] For UDFlex, the service time of jobs is gradually extended in order to model a growing allocation and communication overhead, and the extension is relative to the resource demand of the jobs. For Adaptive Scan, Best Fit and First Fit, on the other hand, the service time of jobs is not extended. In these experiments, the potential allocation and communication overhead of UDFlex are modelled together as a total overhead. Whether the allocation overhead or the communication overhead contributes most to the total overhead may depend on the service time of the jobs. For jobs with long service times the communication overhead may be the main factor, whereas for jobs with shorter service times the allocation overhead may become more important. The second set of experiments is conducted for mesh topologies and for jobs with a potentially high resource demand. A load level of 0.9 (with an ST_{mean} of 1 000 cycles and an IT_{mean} of 314 cycles) is selected as the basis. Given that UDFlex is used and that a job requests a number of nodes $|R| > 1$, the service time ST (drawn from an exponential distribution) is extended according to the following formula: Extended service time $ST' = ST + \frac{ST \times d \times |R|}{100 \times w \times h}$, where d represents the desired increase (a certain percentage) in the job's service time, whereas w is the width and h is the height of the mesh. We simulated a set of levels for d ranging from 0 to 130%.

6.1.4 Results

In the following, we present the results of our evaluation of the UDFlex allocation strategy. Fragmentation is an inherent issue for contiguous allocation strategies. In order to study whether the use of UDFlex is advantageous with respect to fragmentation, we compared UDFlex with a number of strategies that allocate sub-meshes. The results of these experiments are presented in the first

[2] Flexfold, which may alter the shape of a requested sub-mesh and thereby change the original distance between nodes, is not included in these experiments.

subsection below. When compared to using a strategy that allocate sub-meshes, using UDFlex may entail a higher allocation and communication overhead. The second subsection below presents the results of the experiments conducted to assess the amount of allocation and communication overhead that can be tolerated for UDFlex before its advantages with respect to fragmentation are neutralized. A set of representative plots was selected and included.

Fragmentation

System utilization and queuing time versus input load are shown in Figures 6.2 and 6.3 for the 32×32 mesh and torus topologies, respectively. For jobs with a potentially medium and a low resource demand, Figures 6.2(c) and 6.2(d) show queuing time for the 32×32 mesh, whereas Figures 6.3(c), and 6.3(d) show queuing time for the 32×32 torus. As the input load increases, we observe how the queuing time for UDFlex stays significantly lower than the queuing time for the methods that allocate sub-meshes. For all of the allocation strategies the queuing time starts to increase steeply at some input load level. However, when compared to the sub-mesh allocating strategies, UDFlex sustains a higher input load before such a steep increase in queuing time occurs. These results indicate that using UDFlex, rather than using a strategy that allocates sub-meshes, reduces fragmentation. As expected, Random, which is a fragmentation-free and non-contiguous allocation strategy, achieves an even lower queuing time than UDFlex does.

For the 32×32 mesh, system utilization for jobs with a potentially high and a low resource demand are shown in Figures 6.2(a) and 6.2(b), respectively. We observe how the knee-point of the curves (the point where the maximum system utilization is reached) occurs for a higher input load when using UDFlex than when using any of the other contiguous allocation strategies. Thus, UDFlex achieves a significantly higher system utilization than the other contiguous allocation strategies do. As expected, Flexfold has the best performance of the algorithms that allocate sub-meshes. Recall from Section 2.7.1 that Flexfold allows more flexibility with respect to the shape of a sub-mesh than the alternative strategies do. Flexfold can fold the originally requested $a \times b$ sub-mesh into two other, di erently shaped, sub-meshes comprising the same number of nodes, and the three alternative sub-meshes may also be rotated by 90 degrees. For jobs with a potentially high resource demand, the maximum system utilization for Flexfold is 0.53; for UDFlex it is 0.68 (28.3% higher than Flexfold); whereas for Random, the upper bound performance indicator, it is 0.73 (not more than 7.4% higher than UDFlex). For jobs with a low resource demand, the maximum system utilization for Flexfold is 0.73; for UDFlex it is 0.86 (17.8% higher than Flexfold); whereas for Random it is 0.98 (14.0% higher than UDFlex). Random is a fragmentation-free allocation strategy, but does not achieve a maximum system utilization of 1 for any of our scenarios. The reason is that, in some situations, the first job in the waiting queue is larger than the number of free nodes in the system. The probability that such situations occur is higher for jobs with a potentially high resource demand than for jobs with a lower resource demand, and both for Random, UDFlex and Flexfold the maximum achieved system utilization is higher the lower the resource demand of the jobs is.

For the 32×32 torus, system utilization for jobs with a potentially high and a low resource demand is shown in Figures 6.3(a) and 6.3(b), respectively. As with the mesh topology, we observe how the knee-point of the curves occurs for a higher input load when using UDFlex than when allocating sub-meshes. When compared to the strategy that allocates sub-meshes, UDFlex signifi-

Table 6.1: In percentage increase of the maximum system utilization when using UDFlex rather than using the best sub-mesh allocating strategy. (Topology size versus resource demand of jobs.)

(a) Mesh

	High demand	**Medium demand**	**Low demand**
$\mathbf{16 \times 16}$	23.6 %	20.9 %	8.6 %
$\mathbf{32 \times 16}$	38.8 %	25.0 %	14.5 %
$\mathbf{32 \times 32}$	28.3 %	27.0 %	17.8 %

(b) Torus.

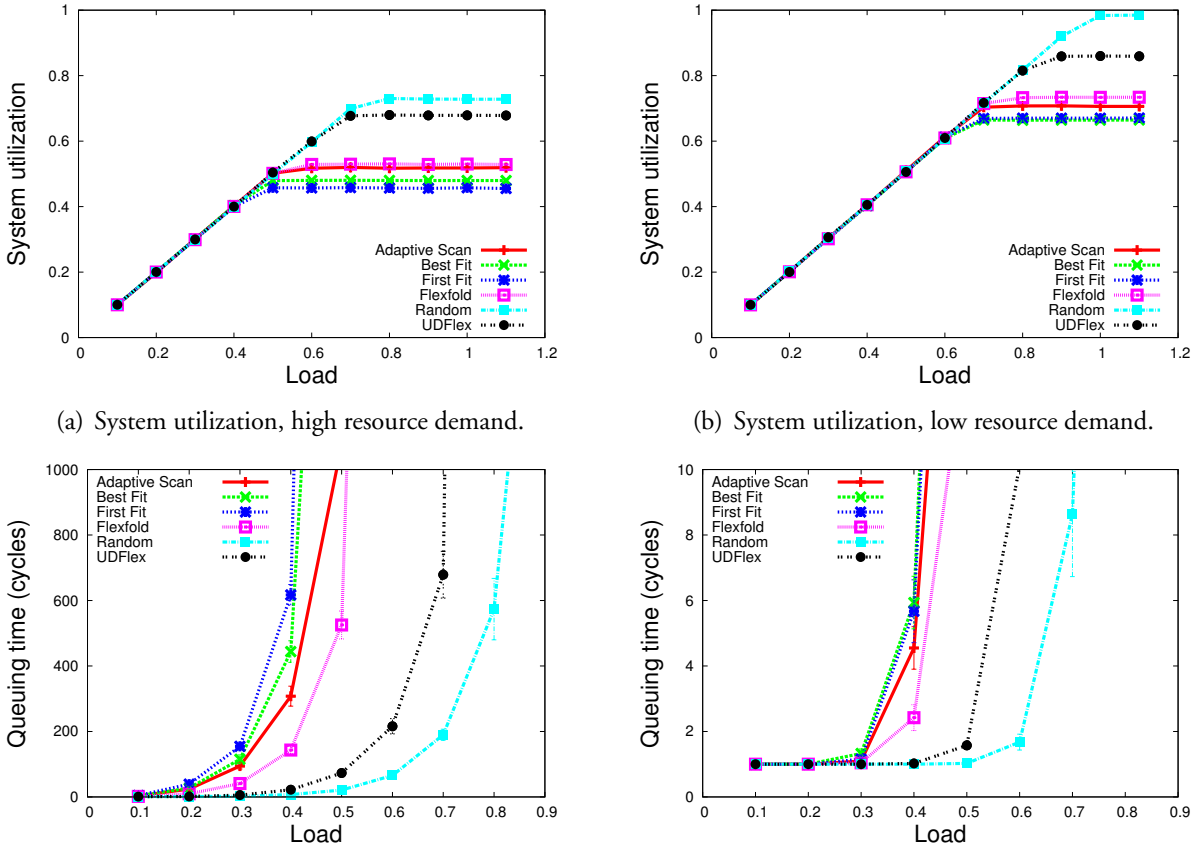
	High demand	**Medium demand**	**Low demand**
$\mathbf{16 \times 16}$	17.2 %	17.4 %	11.4 %
$\mathbf{32 \times 16}$	30.8 %	23.1 %	16.0 %
$\mathbf{32 \times 32}$	21.4 %	23.1 %	19.4 %

cantly improves the maximum achieved system utilization (with 21.4% and 19.4% for jobs with a potentially high and a low resource demand, respectively).

A torus topology may be considered a mesh topology with an additional set of communication channels – the wraparound channels. Random does not consider communication channels when allocating a set of nodes to a job. Thus, the results for meshes and tori are identical when using Random. Recall that, when based on a breadth-first search, the spanning tree of the Up*/Down* algorithm will typically be shallower and wider for a torus than for a mesh topology. Nevertheless, our results indicate that a similar maximum system utilization is achieved for the meshes and tori when using UDFlex.

Let k refer to the in percentage increase of the maximum system utilization observed when using UDFlex rather than using the best sub-mesh allocating strategy. For every combination of topology size and resource demand of jobs Tables 6.1(a) and 6.1(b) show k for the mesh and torus topologies, respectively. These results show that, in most cases, k is higher the larger the topology is. However, for jobs with a potentially high resource demand, an exception is observed for the 32×16 mesh and torus. In these cases, UDFlex achieves a more than 30% higher system utilization than the best sub-mesh allocating strategy achieves. The reason is that the traditionally best-performing sub-mesh allocating strategies perform relatively worse in the case of a non-quadratic mesh or torus – as a request for a sub-mesh $a \times b$ cannot be rotated by 90 degrees if $a > 16$. In addition, some of the folded alternative sub-meshes supported by Flexfold may not fit the 32×16 mesh topology. Figure 6.4 shows system utilization and queuing time for the 32×16 mesh and torus. The greater advantage of UDFlex – due to the relatively worse performance of the traditionally best-performing sub-mesh allocating strategies – is perhaps most easily spotted for the 32×16 mesh, where Adaptive Scan and Flexfold do not perform better than Best Fit (see Figures 6.4(a) and 6.4(c)).

For our first subset of experiments (to which the results presented above belong), the desired range of load levels resulted from a variation of IT_{mean} whereas the value of ST_{mean} was fixed. For our second subset of experiments, the value of IT_{mean} was fixed whereas the desired range of load levels resulted from a variation of ST_{mean}. A comparison of the results obtained for the two di erent subsets of experiments is presented in Figure 6.5: Figures 6.5(a) and 6.5(b) compare the system utilization obtained for a 16×16 torus, whereas Figures 6.5(c) and 6.5(d) compare the queuing time obtained for a 16×16 mesh. We observe that the system utilization is similar regardless of which of the two parameters (ST_{mean} or IT_{mean}) is varied. On the other hand, the



(a) System utilization, high resource demand.

(b) System utilization, low resource demand.

(c) Queuing time, medium resource demand.

(d) Queuing time, low resource demand.

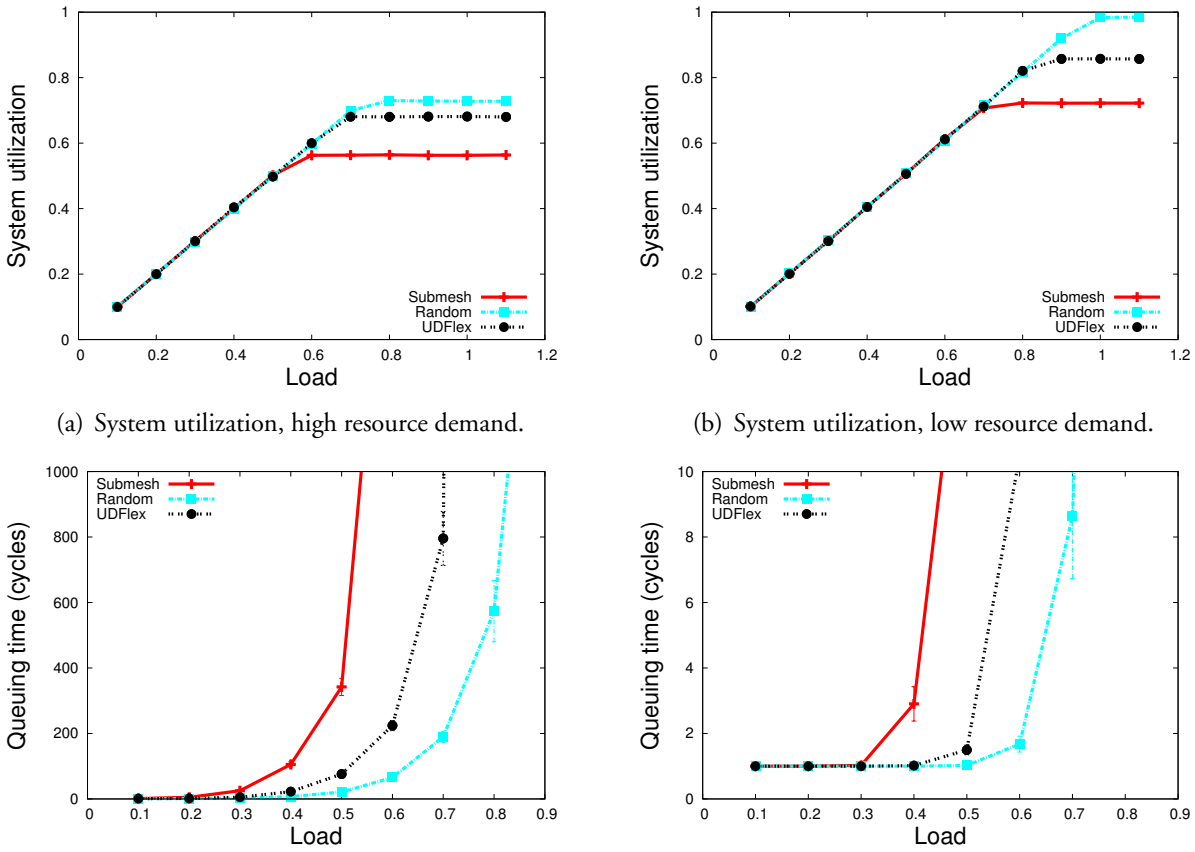
Figure 6.2: System utilization and queuing time for a 32×32 mesh.

queuing time obtained when ST_{mean} is varied di ers from the queuing time obtained when IT_{mean} is varied. In this study, the ranking of the allocation strategies is more important than the absolute numbers obtained for some metric, and, also with respect to queuing time, the ranking of the processor allocation strategies is the same regardless of which of the two parameters (ST_{mean} or IT_{mean}) is varied.

Allocation and communication overhead

The results presented in the previous subsection showed that, due to reduced fragmentation, UDFlex obtains a significantly higher system utilization and lower queuing time than the strategies that allocate sub-meshes do. However, as we recall, UDFlex may have a higher allocation and communication overhead than the sub-mesh allocating strategies have, and the previously presented experiments did not take such a potential overhead into consideration. As it could reduce the advantage of UDFlex over the strategies that allocate sub-meshes, the potential allocation and communication overhead of UDFlex should eventually be considered. This subsection presents the results of a set of experiments which aims to quantify the amount of overhead that could be tolerated before the advantage of UDFlex with respect to fragmentation is neutralized.

Assume that d refers to a given in percentage increase in the service time of jobs, representing a certain level of allocation and communication overhead when using UDFlex. Then, Figure 6.6 shows queuing time and system utilization as functions of d. Figure 6.6(d) shows system utilization



(a) System utilization, high resource demand.

(b) System utilization, low resource demand.

(c) Queuing time, medium resource demand.

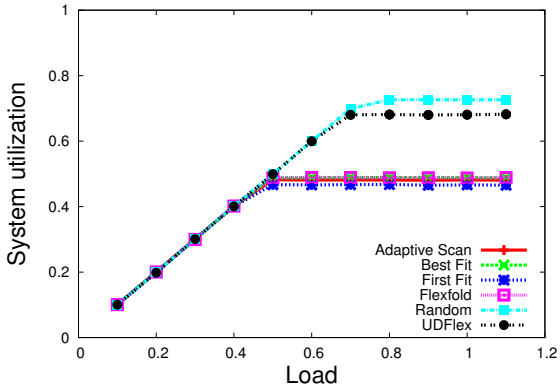
(d) Queuing time, low resource demand.

Figure 6.3: System utilization and queuing time for a 32×32 torus.

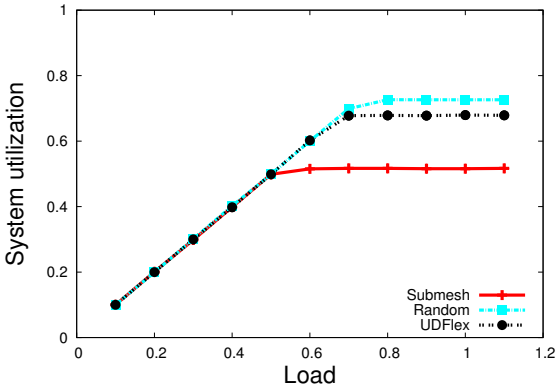
for the 32×32 mesh, and we observe how this metric is barely a ected as the overhead increases. The queuing time, on the other hand, is a ected. Figures 6.6(a), 6.6(b) and 6.6(c) show queuing time for the 16×16, 32×16 and 32×32 meshes, respectively. For the 16×16 mesh an overhead of around 60% can be tolerated for UDFlex before its queuing time crosses the queuing time of Adaptive Scan, whereas for the 32×32 mesh the crossing occurs at an overhead of around 70%. As discussed in the previous subsection, Adaptive Scan performs relatively worse for the 32×16 mesh for jobs with a potentially high resource demand (since a request for a sub-mesh $a \times b$ cannot be rotated by 90 degrees if $a > 16$). Our results show that for the 32×16 mesh UDFlex tolerates an overhead of around 90% before its queuing time crosses the queuing time of Best Fit and Adaptive Scan. First Fit has the highest queuing time of the three strategies that allocate sub-meshes, and an overhead of at least 100% can be tolerated for UDFlex before its queuing time crosses the queuing time of First Fit.

6.2 A framework for developing tra c-contained allocation strategies

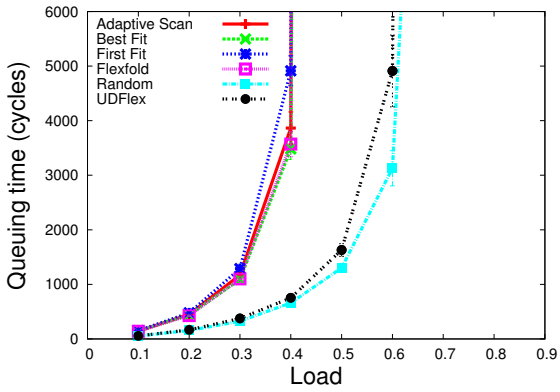
In this section we present a topology agnostic and conceptually simple framework for developing processor allocation strategies. The framework introduces the combination of a flexible processor allocation strategy, a topology agnostic routing algorithm and static reconfiguration as a means to



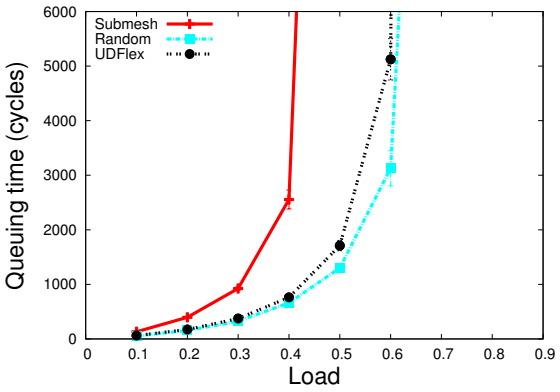
(a) System utilization, 32 × 16 mesh.



(b) System utilization, 32 × 16 torus.



(c) Queuing time, 32 × 16 mesh.



(d) Queuing time, 32 × 16 torus.

Figure 6.4: Non-quadratic topologies: System utilization and queuing time for jobs with a potentially high resource demand.

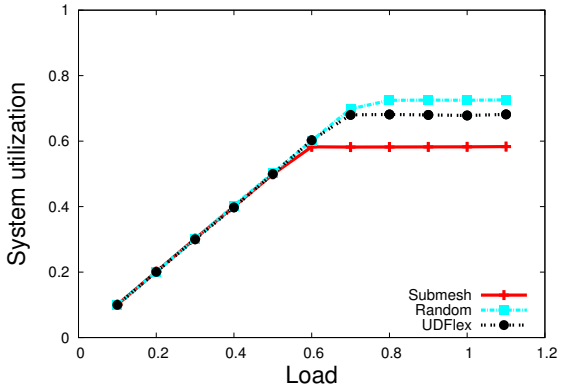
achieve low fragmentation and ensure tra c-containment within partitions.

Section 6.2.1 presents the framework, whereas Section 6.2.2 discusses an example configuration. A performance evaluation is included in Sections 6.2.3 and 6.2.4. Finally, Section 6.2.5 proposes and evaluates an alternative approach which aims to allocate partitions with an improved support for tra c-containment.

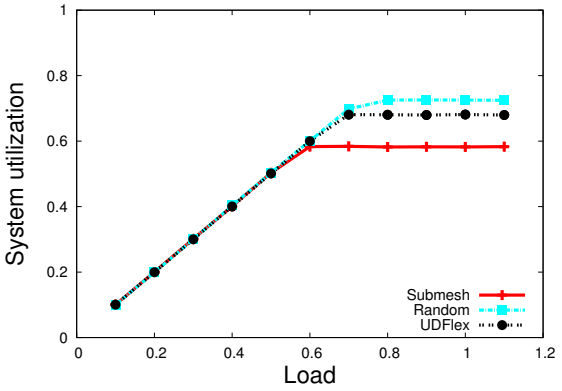
6.2.1 The framework

The main motivation behind our framework is to introduce a generic approach to designing tra c-contained allocation strategies that support a low fragmentation of the processing resources available in a system. The following two observations are at the basis of the framework: First, although fragmentation is an inherent issue for contiguous allocation algorithms, flexibility with respect to the shape of the allocated partitions may result in reduced fragmentation. Second, a routing algorithm that does not assume a particular topology could be utilized to ensure tra c-containment within contiguous partitions with non-regular shapes.

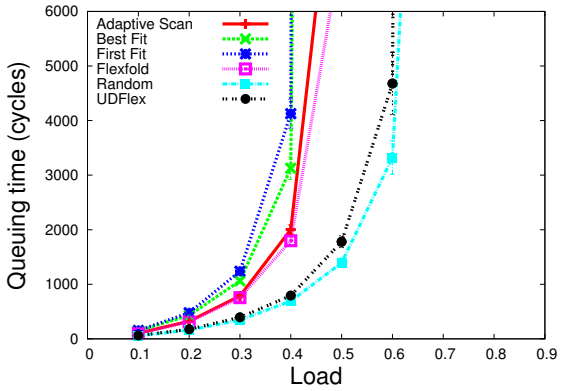
The framework assumes a close collaboration between an allocator and a routing module. It prescribes the assignment of an arbitrarily shaped contiguous partition to a parallel job; calculation of a new set of routes that ensures tra c-containment for each new partition; and local reconfiguration of the routing function for each job to be started. The requirement that each partition must



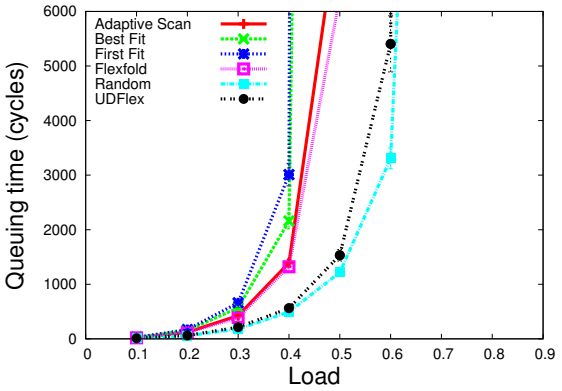
(a) System utilization, 16 × 16 torus, fixed ST_{mean}, varying IT_{mean}.



(b) System utilization, 16 × 16 torus, fixed IT_{mean}, varying ST_{mean}.



(c) Queuing time, 16 × 16 mesh, fixed ST_{mean}, varying IT_{mean}.

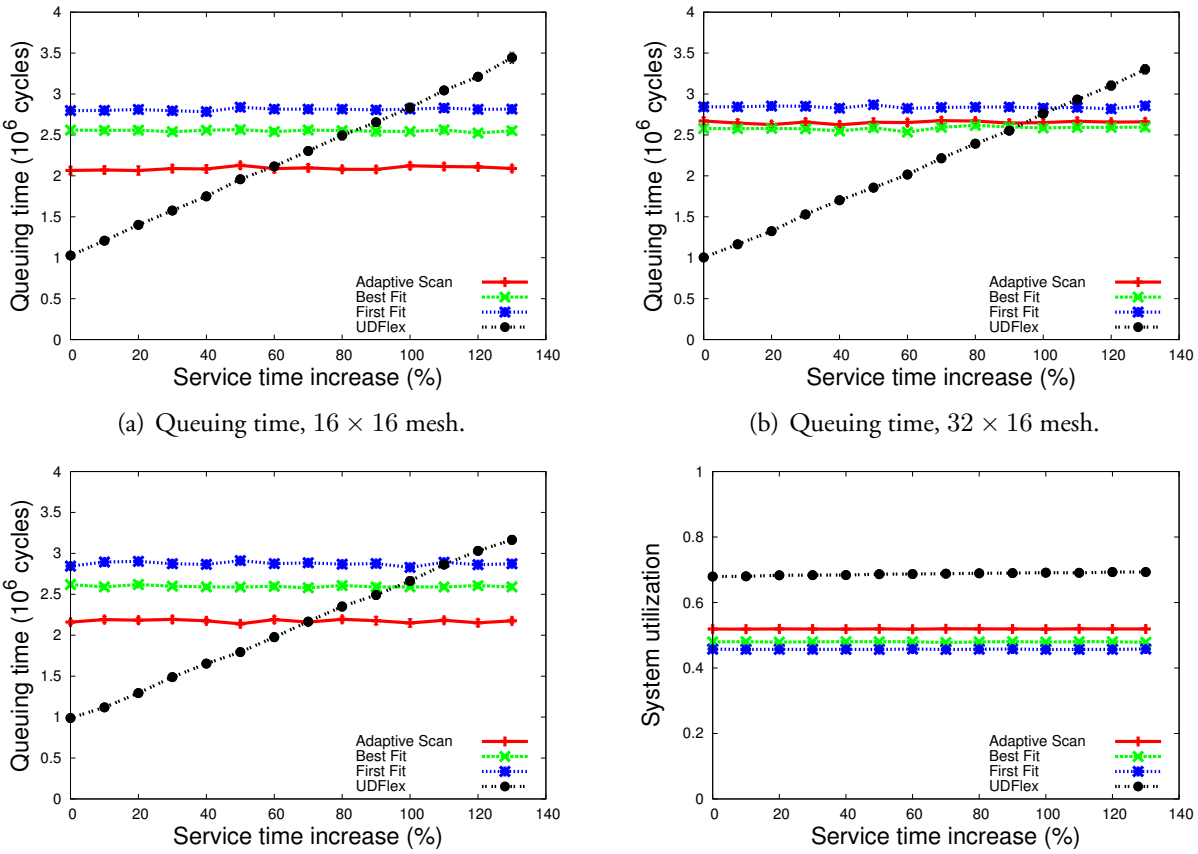


(d) Queuing time, 16 × 16 mesh, fixed IT_{mean}, varying ST_{mean}.

Figure 6.5: Sensitivity of the system utilization and queuing time metrics towards the parameters ST_{mean} and IT_{mean}.

consist of a contiguous region of nodes is due to the intention of supporting tra c-containment within each partition. We believe that by not imposing any restrictions on the shape of partitions, fragmentation may be significantly alleviated, resulting in a high utilization of a system's available processing resources. For such arbitrarily shaped partitions, a topology agnostic routing algorithm is needed in order to calculate routes that keep the tra c within the partitions. Furthermore, as partitions will be di erently sized and shaped, in order to ensure tra c-containment and connectivity, the routing function must in general be reconfigured locally for each new partition before a job is started. Such an update of the routing function is performed on a region of the interconnection network where no job is currently running. Thus, sophisticated dynamic reconfiguration strategies are in general not needed – simple static reconfiguration is su cient.

Our focus is on how the routing function can be updated with the purpose of ensuring tra c-containment and connectivity within partitions. However, in order to forward control tra c (traffic that is not part of the intra-partition tra c related to the execution of a parallel job), a globally connected routing function is generally needed. Such a routing function allows communication between pairs of nodes in an interconnection network that are not part of the same partition. In order to simplify the discussion, we assume that a globally connected routing function is implemented on a separate network for control tra c, and thus that interference between control tra c



(a) Queuing time, 16 × 16 mesh.

(b) Queuing time, 32 × 16 mesh.

(c) Queuing time, 32 × 32 mesh.

(d) System utilization, 32 × 32 mesh.

Figure 6.6: Tolerance for the potential allocation and communication overhead of UDFlex. (Such overhead is represented by an in percentage increase of the service time of jobs.)

and intra-partition tra c does not occur.

Next, our generic algorithm for developing tra c-contained and low-fragmentation allocation strategies is presented. We refer to this generic algorithm as a *framework* since no particular set of algorithms for contiguous processor allocation, topology agnostic routing and reconfiguration is assumed. Thus, a number of combinations of allocation, routing and reconfiguration algorithms are conceivable, and a particular example configuration is discussed in Section 6.2.2. However, as simple static reconfiguration algorithms are in general su cient, reconfiguration is not discussed in detail in the following. Assume that a system contains an interconnection network I which consists of a set of nodes and communication channels; that the routing module runs a topology agnostic routing algorithm; that the allocator and the routing module are able to communicate; that a routing function F applies for I that ensures tra c-containment and connectivity within those partitions where jobs are already running; and that a parallel job just selected by the scheduler requests a number of nodes $|R|$. Further assume that a *legal path* refers to a path that obeys any restrictions imposed by the system and by the routing algorithm. (Examples of such restrictions include the number of virtual channels (VCs) available and turn-restrictions.)

1. The number of free nodes, $|N_{free}|$, of I is checked. The request can only be granted if $|N_{free}| \geq |R|$. If $|N_{free}| < |R|$, the scheduled job is returned to the waiting queue, pending termination of a running job.

2. A new contiguous region P consisting of exactly $|R|$ free nodes is identified and tentatively reserved for the scheduled job.

3. The allocator notifies the routing module of the new partition P.

4. The routing module must calculate a set of deadlock-free paths for P and locally reconfigure F as follows: For every pair of nodes inside P, F must contain at least one legal path that does not traverse an intermediate node outside P, and F must contain no path that traverses an intermediate node outside P.

    - For some pair of nodes inside P a legal path that does not traverse an intermediate node outside P may not exist. In such a case, the tentative reservation of $|R|$ nodes is cancelled.
        - If there are more candidate partitions that could be tested, then the algorithm may return to step 2.
        - Else the scheduled job is returned to the waiting queue, pending termination of a running job.

5. The routing module notifies the allocator that the routing function is ready.

6. The allocator assigns P to the scheduled job.

The status of the nodes that are part of P is changed from free to busy during step 6 of the algorithm, and the scheduled parallel job is not started on P until step 6 has completed. Note that if a scheduled job is returned to the waiting queue, the remaining algorithmic steps are skipped, and the job must be rescheduled.

We need to prove that any allocation strategy developed using our framework ensures isolation of the tra c belonging to di erent jobs as well as communication between every pair of nodes that are part of the same partition. Thus, we include and prove Lemmas 6.3 and 6.4 below.

Lemma 6.3. *The framework ensures tra c-containment within each partition.*

Proof. The proof is by contradiction. Assume that an arbitrary node s and another arbitrary node d are both part of a partition P, and that underway from its source s to its destination d a packet crosses an intermediate node i which is not part of P. This contradicts step 4 of the framework which states that, for every pair of nodes inside P, F must contain no path that traverses an i outside P. Thus, for a job running on P, no packet from s to d can leave P. □

Lemma 6.4. *The framework provides a connected routing function within each partition.*

Proof. The proof is by contradiction. Assume that a job runs on a partition P; that an arbitrary node s and another arbitrary node d are both part of P; and that no legal path exists within P from s to d. This contradicts step 4 of the framework which directs that a scheduled job is not started on P unless at least one legal path (that does not traverse an intermediate node outside P) exists for every pair of nodes inside P. Thus, for a job running on P, at least one legal path must exist within P from s to d. □

When a parallel job has finished running on P, the nodes that constituted P must be deallocated. That is, their status must be changed from busy to free. We will not further discuss deallocation of nodes.

6.2.2 An example configuration

Our framework for developing tra c-contained and low-fragmentation allocation strategies supports the use of various pairs of processor allocation strategies and routing algorithms. This section gives an example of one of a number of possible combinations of algorithms that are usable with the framework. The chosen allocation strategy is described in the first subsection below, and the chosen topology agnostic routing algorithm is described in the second subsection below. The former is a slightly modified version of an existing allocation algorithm, whereas the latter is an existing routing algorithm. The algorithms below are also used in the performance evaluation presented in Sections 6.2.3 and 6.2.4.

A flexible allocation strategy

For the example configuration we have chosen slightly modified (contiguous) versions of the MC [146] and MC1x1 [34] allocation strategies. MC evaluates the costs of the allocation alternatives centered around every free node in a mesh or torus topology, and selects the alternative with the lowest cost. The cost of an allocation alternative is decided by using expanding *shells* centered around a free node, where a node in shell 0 has cost 0, a node in shell 1 has cost 1, and so on. In the following, a two-dimensional mesh or torus topology is assumed. For a sub-mesh request of size $a \times b$, both the orientation where shell 0 is $a \times b$ and the 90 degrees rotation where shell 0 is $b \times a$ are evaluated. The selection of the allocation alternative with the lowest cost means that a sub-mesh is allocated if one is available. Figure 6.7(a) illustrates an example where, for a request of 3×1 nodes, the cost of an allocation alternative centered around node 21 is 1 (nodes 20 and 21 are in shell 0 and node 14 is in shell 1).

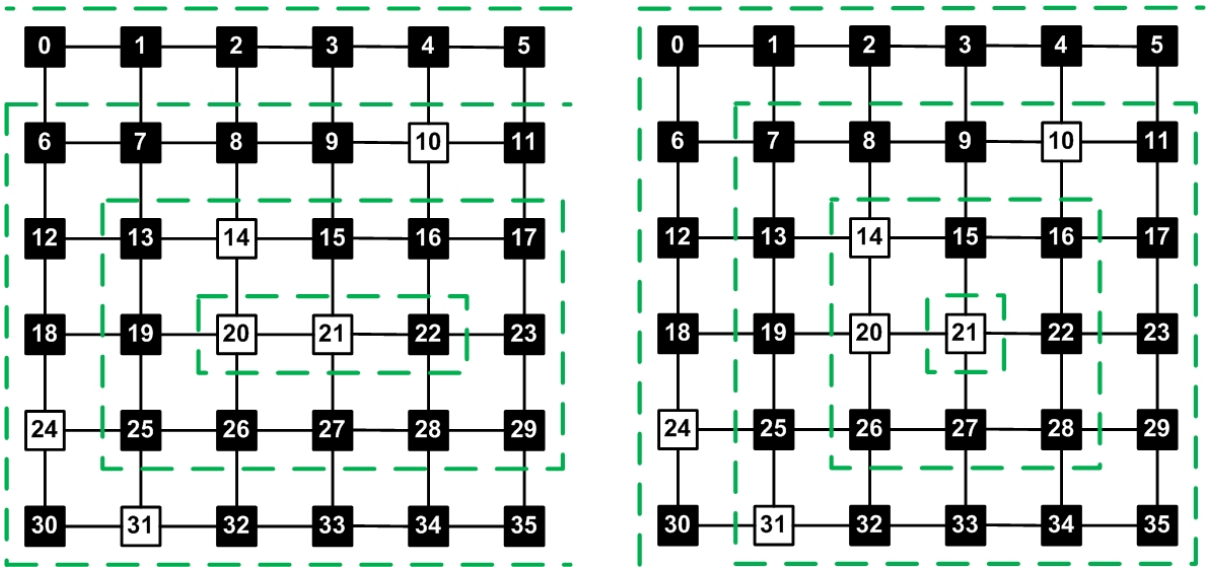
MC1x1 is an adaptation of MC for a system that requests a certain number of nodes rather than a specific sub-mesh. The main di erence from MC is that for MC1x1 shell 0 is a sub-mesh of size 1×1. For a request of three nodes, Figure 6.7(b) shows that for MC1x1 the cost of an allocation alternative centered around node 21 is 2 (node 21 is in shell 0 and nodes 14 and 20 are in shell 1).

MC and MC1x1 allow non-contiguous allocation. In this study we are interested in the e ect of tra c-containment and thus need contiguous allocation strategies. We restricted MC and MC1x1 to only consider the allocation alternatives that constitute contiguous regions, and refer to the resulting allocation strategies as Contiguous-MC (CMC) and Contiguous-MC1x1 (CMC1x1). For a request of 3×1 and 3 nodes, respectively, nodes 14, 20 and 21 in Figures 6.7(a) and 6.7(b) constitute a contiguous region of nodes and thus a valid partition according to CMC and CMC1x1.

In order to support a high utilization of a system's processing resources, our framework allows allocation of arbitrarily shaped partitions. CMC and CMC1x1 recognize any available and su - ciently large contiguous region of nodes. The strategies aim at minimizing the dispersal of the allocated nodes, and are conceptually simple. On the negative side, CMC and CMC1x1 have relatively high computational complexity and are not topology agnostic. Nevertheless, for our framework we consider CMC and CMC1x1 suitable examples of strategies that can be used with the allocator.

A topology agnostic routing algorithm

For the example configuration we have chosen the Transition-Oriented Routing (TOR) algorithm [200]. TOR's manner of operation was discussed in Sections 2.8.1 and 5.2, and is therefore



(a) CMC evaluates a request for 3 × 1 nodes.

(b) Shell 0 is always a sub-mesh of size 1 × 1 for CMC1x1.

Figure 6.7: The CMC and CMC1x1 strategies use *shells* around every free node in order to evaluate the costs of di erent allocation alternatives. The broken lines indicate the shells, and the white and black squares represent free and busy nodes, respectively. The illustrations are inspired by [146].

only briefly repeated here. TOR allows selection of any available path between a source and destination node, and depends on VC-transitions for deadlock avoidance. As was also assumed in [200], we assume that such VC-transitions are based on a directed acyclic graph (DAG) corresponding to an Up*/Down* [204] graph. TOR prevents deadlock by requiring that when a packet traverses from a down-link to an up-link it makes a transition to the next higher VC.

In order to support tra c-containment as well as connectivity within each partition, our framework prescribes the use of a topology agnostic routing algorithm. TOR is topology agnostic, supports a flexible shortest path routing, and thus has the potential of providing excellent performance. For a partition of any shape TOR can calculate a set of routes that guarantees tra c-containment as well as connectivity between all the nodes of the partition (given that a su cient number of VCs are available). Thus, for our framework we consider TOR a suitable example of an algorithm that can be used with the routing module.

6.2.3 Experiment setup

Our simulation experiments demonstrate application of the framework proposed in Section 6.2.1, and evaluate performance – with respect to both allocation and communication – assuming the use of the processor allocation and routing strategies described in Section 6.2.2.

Recall from Section 3.1 that, in order to overcome the challenges related to the di erent timescales of the two types of experiments, two di erent simulator models – an allocation simulator and a communication simulator – are used. Also recall that the packet-exchange (communication) experiments will be run only for a number of snapshots output from the allocation experiments, not for every allocated partition. A snapshot (like one of those in Figure 6.10) depicts the jobs running at a certain moment in time.

The first subsection below describes the setup of the experiments to investigate the e ciency of fully flexible contiguous processor allocation. For these experiments the allocation simulator is used. The second subsection below describes the setup of the packet-exchange experiments. For these experiments the communication simulator is used.

The allocation model

The main purpose of this set of experiments is to study whether our framework's support of contiguous partitions with fully arbitrary shapes will result in a significantly improved utilization of a system's processing resources.

The experiments compare the performance of CMC and CMC1x1 with the performance of the UDFlex strategy presented in Section 6.1.1. UDFlex is implemented in accordance with the description in Section 6.1.2 (which explains how, with future allocation attempts in mind, some optimizations are included in order to keep regions of free nodes defragmented). For UDFlex, the root of the Up*/Down* graph is in the upper left corner[3] of the topology. We also include a sub-mesh allocating strategy similar to Adaptive Scan [61] in our experiments. Furthermore, the Random [137] allocation strategy is included. Recall that Random is a non-contiguous, fragmentation-free, allocation strategy that fails only when the next job to be served requires a higher number of nodes than the number of currently free nodes. Random is included to visualize the upper performance benchmark with respect to our metrics (not considering communication overhead).

We consider both a mesh and torus topology of size 16×16. In order to evaluate the performance of the allocation algorithms we use the metrics *system utilization* and *queuing time*. Assume that a mesh or torus has width w and height h; that the position of any node can be expressed as a pair of coordinates (x, y) where $1 \leq x \leq w$ and $1 \leq y \leq h$; that the node in position (i, j) has been busy for the aggregated time $busy_{i,j}$; and that data have been collected for a period of time T. Then, as in [262], we define the system utilization to be $\frac{\sum_{1 \leq i \leq w, 1 \leq j \leq h} busy_{i,j}}{w \times h \times T}$, and the *system fragmentation* is then $1 -$ *system utilization*. The queuing time is the average time that a job is held in the waiting queue, from the time of the arrival of a job until its requested nodes have been allocated.

CMC and the sub-mesh allocating strategy expect the requests for nodes to be in the form of sub-meshes, whereas CMC1x1, UDFlex and Random expect requests for certain numbers of nodes. For a fair comparison, the resource demand of jobs (the number of nodes requested) must be equal for either group of allocation strategies.

In the first subset of experiments, we let each job request $a \times b$ nodes, which is considered a sub-mesh by CMC and the sub-mesh allocating strategy, and a product of numbers by UDFlex and Random. The values a and b are drawn from separate uniform distributions with minimum sizes a_{min} and b_{min} and maximum sizes a_{max} and b_{max}, respectively. For 80% of the jobs $a_{min} = 1$, $b_{min} = 1$, $a_{max} = 4$ and $b_{max} = 4$, and we refer to this group of jobs as *small jobs*. The remaining 20% of the jobs are referred to as *large jobs* and have $a_{min} = 8$, $b_{min} = 8$, $a_{max} = 12$ and $b_{max} = 12$.

In the second subset of experiments, CMC1x1 is compared with UDFlex and Random. (The sub-mesh allocating strategy is not applicable as the request is not for a sub-mesh of nodes.) The

[3] As in Section 6.1, in order to simplify the discussion, we visualize a torus topology laid out in a plane as a mesh topology with wraparound channels.

resource demand, $|R|$, of each job is drawn from a uniform distribution. 80% of the jobs, the small jobs, have $|R|_{min} = 1$ and $|R|_{max} = 16$, whereas the remaining 20%, the large jobs, have $|R|_{min} = 64$ and $|R|_{max} = 144$.

Recall from Section 3.1.2 that for a mesh or torus of width w and height h the input *load* is $\frac{|R|_{mean} \times ST_{mean}}{w \times h \times IT_{mean}}$, where $|R|_{mean}$ is the mean number of nodes requested by the jobs; ST_{mean} is the mean service time (running time) of jobs; and IT_{mean} is the mean interarrival time of jobs. In our experiments, ST_{mean} is fixed at 1 000 cycles and the desired load levels result from variation of IT_{mean}.

For subsequent input to the communication simulator, 25 snapshots were stored. These snapshots were collected for a load of 1.0, and evenly spread out over the data collection period. For every experiment, data collection spans the execution of 200 000 jobs.

The communication model

Reduced fragmentation is the main motivation for allowing allocation of arbitrarily shaped regions of nodes to parallel jobs. However, the performance of the allocation strategy should also be considered in connection with packet communication. A high communication overhead could cause longer service times of jobs, which would increase the queuing time of subsequent jobs and reduce the throughput of jobs in the system.

When considering fragmentation and communication overhead (and ignoring a potential allocation overhead), it is quite obvious that allowing allocation of arbitrarily shaped partitions is beneficial when applications are not communication intensive. For communication intensive applications, we conduct experiments to investigate the costs and benefits of two alternative approaches to communication: tra c-containment and non-tra c-containment.

As input to these experiments we use the snapshots of running jobs that were output from the allocation experiments. Each of the snapshots contains a number of partitions allocated for a 16×16 mesh or torus topology either by CMC or by CMC1x1.

For the experiments where tra c-containment is not ensured (non-tra c-containment), Dimension-Order Routing (DOR) [231] is used. Recall that for a two-dimensional mesh topology DOR (XY) avoids deadlock by first routing a packet in the X-dimension until the o set in this dimension is zero. Thereafter the packet is routed in the Y-dimension until it reaches its destination. For a two-dimensional torus topology DOR in addition needs two VCs for deadlock avoidance [55].

For the experiments where tra c-containment is ensured, TOR was selected. The choice of TOR as the routing algorithm to be used with our framework was motivated in Section 6.2.2. In particular, we expect that for a sub-mesh shaped partition TOR has the potential of matching the performance of DOR. For every pair of nodes inside a partition, TOR must select a shortest path which ensures both that tra c is contained within the partition and that the number of VCs needed to avoid deadlock is within the number of VCs available. If, for some candidate partition, the number of VCs needed to ensure deadlock-freedom exceeds the number of VCs available, then in a real system the scheduled job cannot be started until another partition becomes available.

These experiments attempt to use a representative number of VCs. An analysis of the snapshots output from the allocation experiments revealed that for the 16×16 mesh only 1 of 459 partitions (0.22%) needed 4 VCs, whereas for the 16×16 torus only 3 of 465 partitions (0.65%)

needed 5 VCs. The snapshots which contained these 4 partitions were not included in the communication experiments, and the number of VCs available for the mesh and torus was set to 3 and 4, respectively.

In order to achieve a balanced load on the VCs, the paths calculated by DOR or TOR are evenly divided among the available VCs.

Two different traffic patterns are applied – one uniform and one hotspot pattern. When using the uniform traffic pattern, for a packet generated by one of the nodes of a partition P each of the other nodes of P is an equally likely destination. Each P has a unique identity-number. For the hotspot traffic pattern, a P with an odd/even numbered ID is a *hotspot/uniform-partition* which communicates in a hotspot/uniform pattern. Within a hotspot-partition, P_h, 50% of the traffic is directed towards a randomly selected hotspot node, whereas the remaining 50% is uniformly distributed within P_h.

Both the ingress and egress buffers of a switch port can hold 2 packets per VC. The transmission queue of a node has space for 4 packets per VC, and overflows when the network cannot deliver packets at the rate they are injected.

The packet injection rate is gradually increased over 100 intervals (each of length 10 000 cycles). Each experiment is run for 1 000 000 cycles in order to ensure representative results.

For a partition P, saturation is reached when the first source node – due to transmission queue overflow – has rejected 10% of the packets it has generated in an interval. Assume that P reaches saturation in an interval where, on average, a node generates a packet every m cycles. Then, the metric $Saturation_*$ is $\frac{t}{m}$, and represents the number of packets generated in a period of time t. In the following equations, which define $SatRatio_{All/One}$ and $SatRatio_{TC/NTC}$, t is cancelled. For each P, $Saturation_{All}$ is measured when all the partitions of a snapshot run together, which means that – unless traffic-containment is ensured – interference may occur between traffic belonging to different partitions. $Saturation_{One}$, on the other hand, is measured when P runs alone on the 16×16 mesh or torus, and there is thus no interference with traffic belonging to any other partition. For each P, both $Saturation_{TC}$ and $Saturation_{NTC}$ are measured when all the partitions of a snapshot run together. Thus, both $Saturation_{TC}$ and $Saturation_{NTC}$ are instances of $Saturation_{All}$. $Saturation_{TC}$ applies when the use of TOR ensures traffic-containment, whereas $Saturation_{NTC}$ applies when the use of DOR implies a risk of non-traffic-containment. We now define $SatRatio_{All/One} = \frac{Saturation_{All} \times 100}{Saturation_{One}}$ and $SatRatio_{TC/NTC} = \frac{Saturation_{TC} \times 100}{Saturation_{NTC}}$. Then, for any P, $SatRatio_{All/One}$ quantifies the effect of the interference between the traffic of P and the traffic of concurrent partitions, whereas $SatRatio_{TC/NTC}$ quantifies the advantage or disadvantage of traffic-containment versus non-traffic-containment.

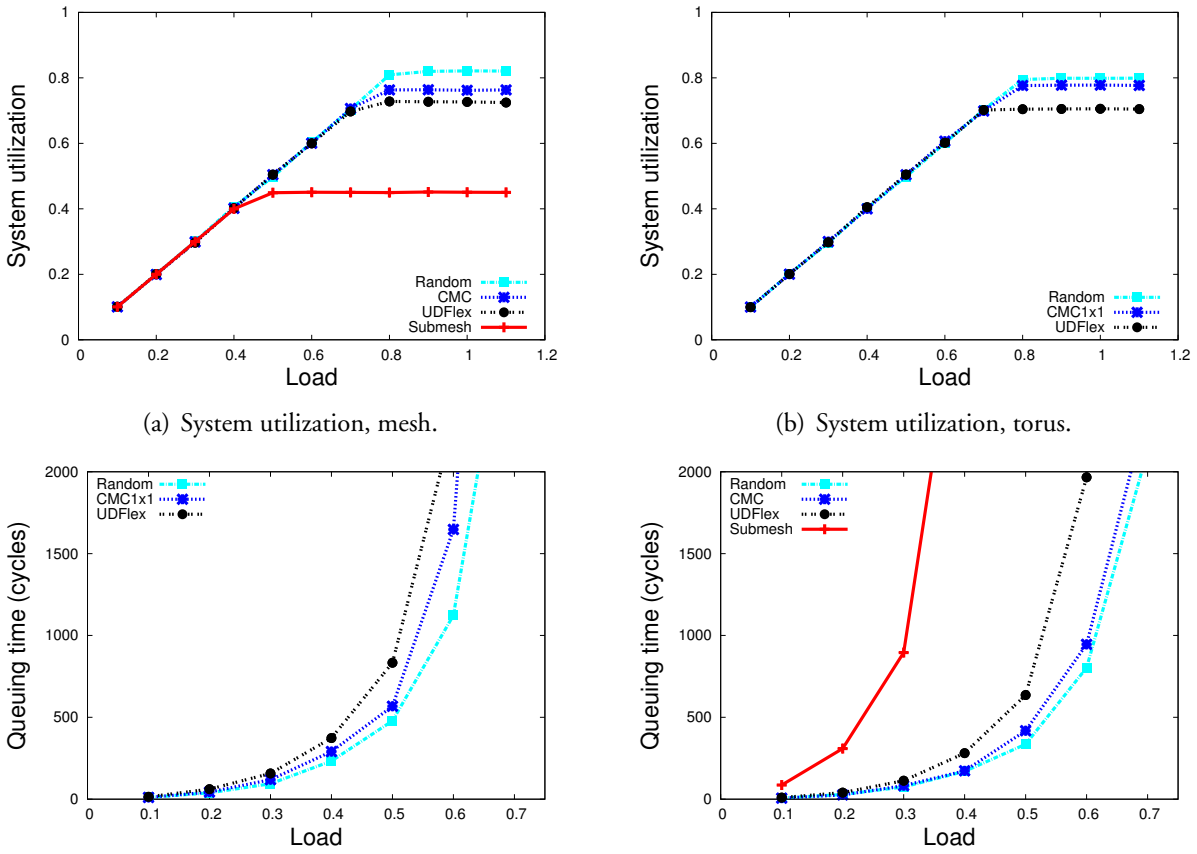
6.2.4 Results

The results of our experiments are presented in three parts. The first subsection below presents the results of the allocation experiments which evaluate the effect of allowing full flexibility contiguous processor allocation. The results of the communication experiments are presented in the second and third subsections below. The former discusses the consequences of allowing interference between traffic belonging to different partitions, whereas the latter discusses the advantages and disadvan-

tages related to enforcement of tra c-containment within partitions. A set of representative plots was selected and included.

Flexible allocation

The system utilization and queuing time for the di erent processor allocation strategies are shown in Figure 6.8. Figures 6.8(a) and 6.8(d) demonstrate that CMC and UDFlex, which allow allocation of non-regular regions of nodes, achieve significantly better performance than when allocations are restricted to sub-meshes. (For the experiments behind Figures 6.8(b) and 6.8(c) allocation of sub-meshes is not applicable as nodes are not requested in the form of sub-meshes.)



(a) System utilization, mesh.

(b) System utilization, torus.

(c) Queuing time, mesh.

(d) Queuing time, torus.

Figure 6.8: System utilization and queuing time for a 16 × 16 mesh and torus.

Figure 6.8 shows that CMC and CMC1x1 improve the system utilization and queuing time when compared to UDFlex. This was expected as CMC and CMC1x1 may allocate any su - ciently large contiguous region of nodes to an incoming job, whereas UDFlex may only allocate a region which constitutes a valid Up*/Down* sub-graph. We also observe that the performance of CMC and CMC1x1 approach that of Random, although contiguous allocation strategies cannot be fragmentation-free.

Consequences of non-tra c-containment

Recall that tra c-containment is not ensured when DOR is used. Figure 6.9 illustrates the performance consequences of allowing the tra c of one job to be routed into regions of nodes allocated to

other jobs. Some jobs achieve a $SatRatio_{All/One}$ of 100% which indicates that their performance is not significantly a ected by tra c from concurrent jobs.[4] For a number of jobs, however, saturation occurs for a significantly lower packet generation rate when tra c belonging to concurrent jobs are allowed to interfere. A comparison of Figure 6.9(a) with Figure 6.9(c) and a comparison of Figure 6.9(b) with Figure 6.9(d) illustrate that the performance degradation caused by such interference depends on the tra c pattern: In general, it is more severe for hotspot tra c than for uniform tra c. In addition, in particular for uniform tra c, the performance degradation is more severe for small jobs than for large jobs.

In order to study in detail how concurrent partitions may a ect each other when tra c-containment is not ensured, let us consider some of the partitions. Figure 6.10 depicts a few of the snapshots of allocated partitions that were input to our communication experiments. Partition 7 in the snapshot of Figure 6.10(a) is a 9-node partition which achieves a $SatRatio_{All/One}$ of 53% and 16% for the uniform and hotspot tra c patterns, respectively (these numbers are extracted from our data files). Figure 6.10(a) reveals that, as DOR (XY) routing is used, tra c belonging to the 99-node partition 4 is routed through – and reduces the performance of – partition 7.

Partition 5 in Figure 6.10(b) is a 108-node partition which is significantly a ected by the tra c of the surrounding 99-node partition 2. For the hotspot tra c pattern, 5 is a uniform partition whereas 2 is a hotspot partition. The $SatRatio_{All/One}$ of partition 5 is 71% and 9% for uniform and hotspot tra c, respectively. These results demonstrate how the performance of partition 5 depends on the communication pattern of partition 2 when tra c-containment within each partition is not enforced.

When comparing the results for partitions allocated using CMC with the results for partitions allocated using CMC1x1, we observed no principal di erences.

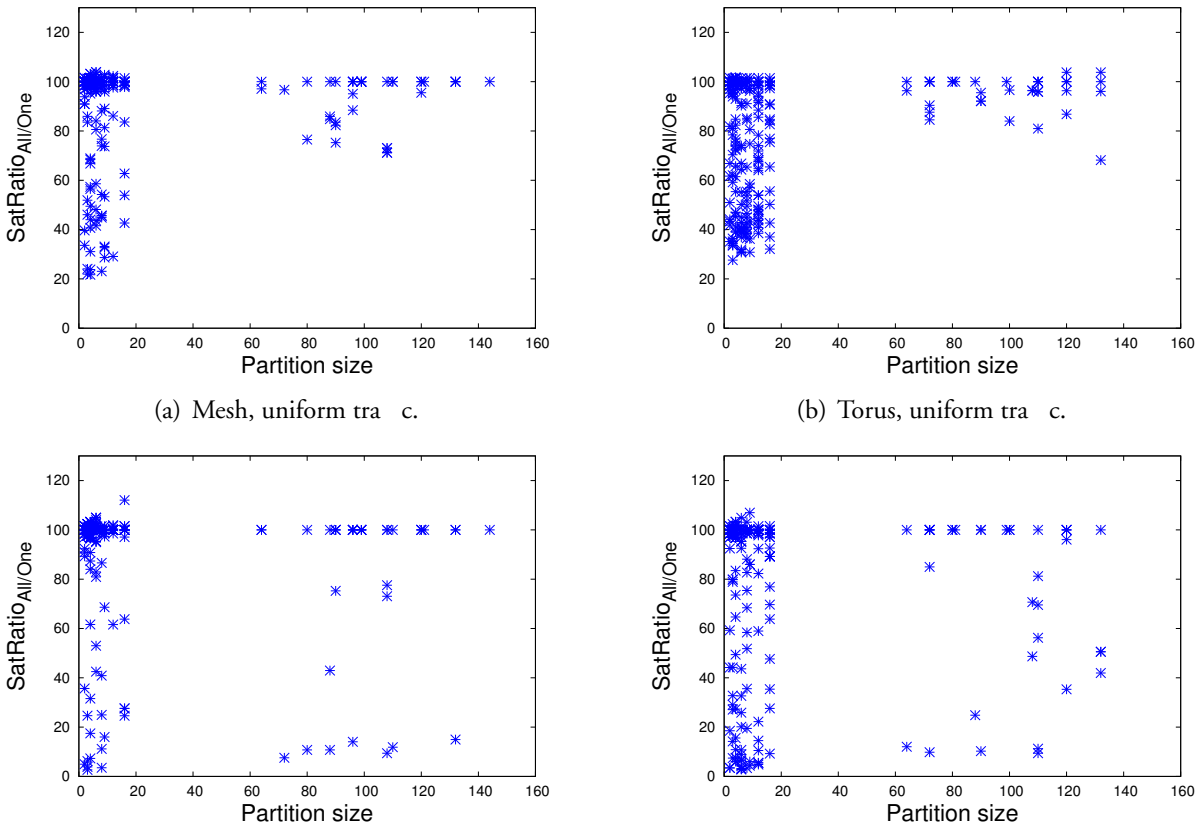
Tra c-contained versus non-tra c-contained communication

We have seen that a lack of tra c-containment negatively a ects the performance for a number of jobs, and that the performance of a job may depend on the communication patterns of concurrent jobs. An important goal of this study is to investigate whether the enforcement of tra c-containment (as prescribed by our framework) is advantageous with respect to performance. Thus, in the following, we evaluate consequences of enforcing tra c-containment versus allowing non-tra c-containment. Recall that tra c-containment ensures isolation between concurrent partitions (meaning that the performance of one job is independent of the packet-exchange of any other job), whereas non-tra c-containment implies a risk of interference between tra c belonging to di erent partitions. Further recall that in these experiments TOR is used to calculate a set of paths that ensures tra c-containment, whereas DOR is used to calculate a set of paths that does not ensure tra c-containment.

Figure 6.11 illustrates that the majority of the small jobs benefit from tra c-containment. These jobs sustain a higher tra c load before saturating in the contained case than in the non-contained case.[5] For hotspot tra c, Figure 6.11(c) shows that for the mesh topology a few of the small jobs achieve a $SatRatio_{TC/NTC}$ as high as 5 000%, and Figure 6.11(d) shows that for the

[4]The few instances where $SatRatio_{All/One}$ is above 100% are due to the coarse granularity in which the packet generation rate is increased, especially for the lower load levels.

[5]We use the terms *contained* and *non-contained*, respectively, for cases where tra c-containment is and is not ensured.



(a) Mesh, uniform tra c.

(b) Torus, uniform tra c.

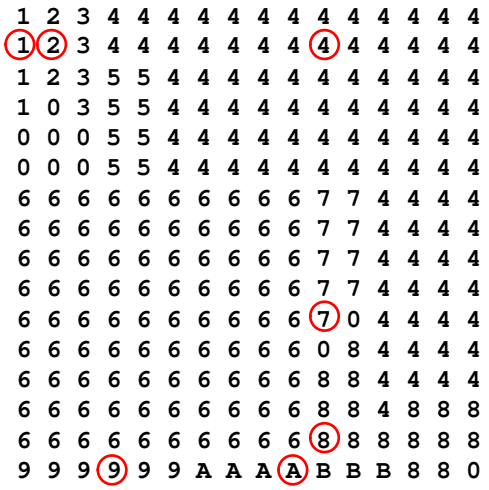
(c) Mesh, hotspot tra c.

(d) Torus, hotspot tra c.

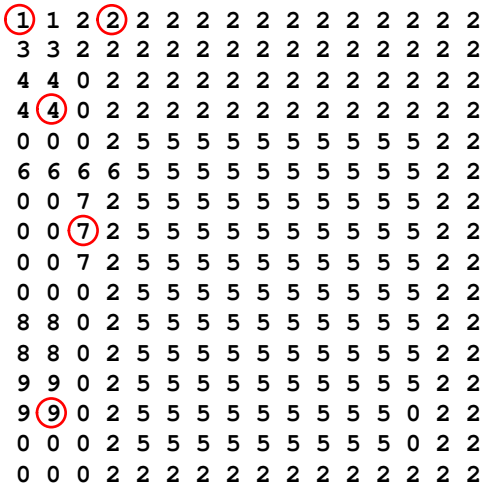
Figure 6.9: Non-tra c-containment: $SatRatio_{All/One}$ for a 16×16 mesh and torus, DOR, CMC.

torus topology a few of the small jobs achieve a $SatRatio_{TC/NTC}$ of more than 3 500%. According to Figures 6.11(a) and 6.11(b), the maximum $SatRatio_{TC/NTC}$ that any job achieves for the uniform tra c pattern is an order of magnitude less – below 500% for the mesh topology and around 350% for the torus topology. Nevertheless, we do observe that a few of the small jobs perform significantly worse when tra c-containment is enforced. A comparison of Figure 6.11(a) with Figure 6.11(e) and a comparison of Figure 6.11(b) with Figure 6.11(f) demonstrate that the performance degradation is comparable for the uniform and hotspot tra c patterns. (Figures 6.11(e) and 6.11(f) are sections of Figures 6.11(c) and 6.11(d), respectively.)

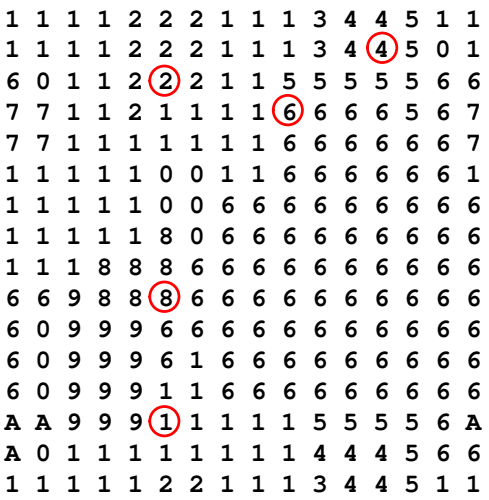
Let us consider the 14-node partition 5 in Figure 6.10(c). This partition has a long, narrow shape, and in the contained case such a shape inevitably implies a significant load on the communication channels. In addition, the contained paths are longer than the non-contained paths for a significant part of the source-destination pairs. For instance, the longest contained path includes 12 hops (between the upper and lower leftmost nodes), whereas the corresponding non-contained path includes only 6 hops. On the other hand, in the non-contained case the tra c of partition 5 interferes with the tra c of partitions 1, 3, 4 and 6. For uniform tra c, partition 5 achieves a $SatRatio_{TC/NTC}$ of only 77%, which means that in this case the drawbacks of tra c-containment are more significant than the drawbacks of non-tra c-containment. For hotspot tra c, on the other hand, partition 5 achieves a $SatRatio_{TC/NTC}$ as high as 587%. Both 4 and 6 are hotspot partitions with their hotspot nodes located close to partition 5 (which communicates in a uniform pattern). This is another example that the benefit of tra c-containment versus



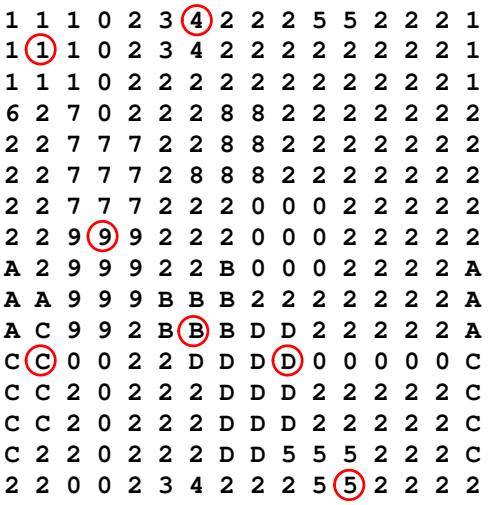
(a) Mesh, CMC, snapshot 11.



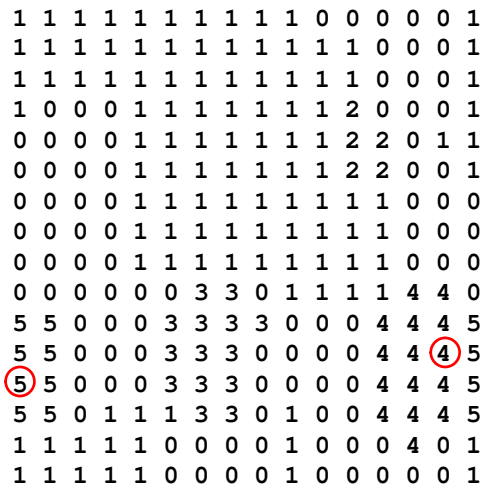
(b) Mesh CMC, snapshot 23.



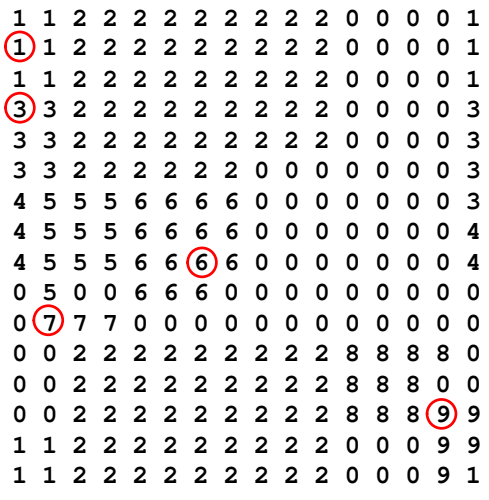
(c) Torus, CMC1x1, snapshot 19.



(d) Torus, CMC1x1, snapshot 1.

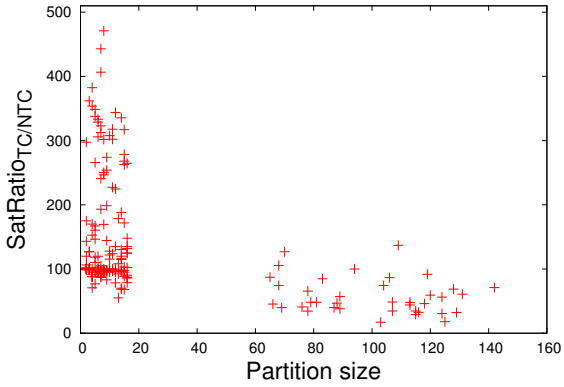


(e) Torus, CMC1x1, snapshot 9.

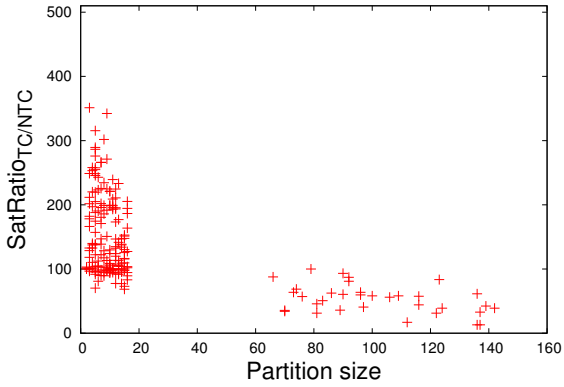


(f) Torus, CMC1x1, snapshot 3.

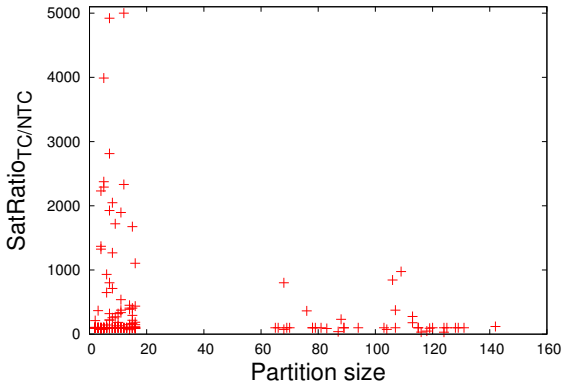
Figure 6.10: Snapshots of allocations for a 16×16 mesh and torus (wraparound channels are not shown). Each running job is represented by a number or letter (0 indicates a free node). Hotspot nodes are marked by a circle (applicable only for the hotspot tra c pattern).



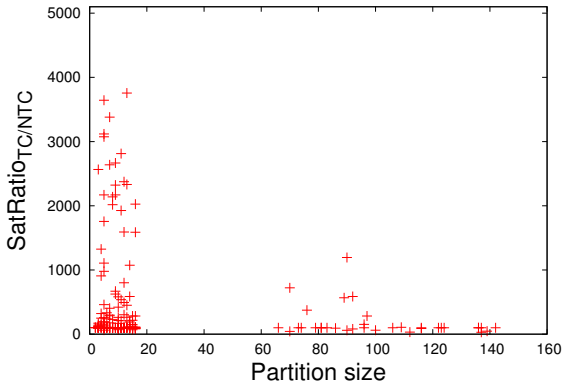
(a) Mesh, uniform tra c.



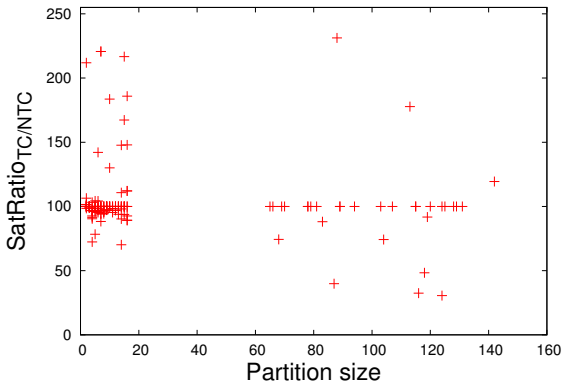
(b) Torus, uniform tra c.



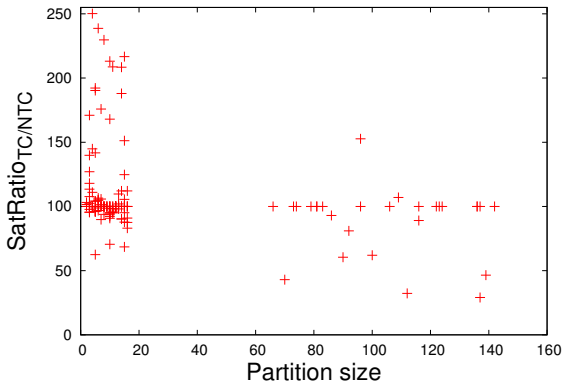
(c) Mesh, hotspot tra c.



(d) Torus, hotspot tra c.



(e) Mesh, hotspot tra c, section of Figure 6.11(c).



(f) Torus, hotspot tra c, section of Figure 6.11(d).

Figure 6.11: Tra c-containment versus non-tra c-containment: $SatRatio_{TC/NTC}$ for a 16×16 mesh and torus, CMC1x1.

non-tra c-containment for a job depends on the communication patterns of concurrent jobs.

Whereas the majority of the small jobs benefit from tra c-containment, regardless of the tra c pattern, for uniform tra c only a few of the large jobs achieve a $SatRatio_{TC/NTC}$ above 100% (see Figures 6.11(a) and 6.11(b)). For hotspot tra c, on the other hand, a significantly larger amount of the large jobs benefit from tra c-containment. Figures 6.11(c) – 6.11(f) illustrate that tra c-containment is an advantage for roughly 25% – and a disadvantage for another 25% – of the large jobs. Thus, for hotspot tra c, approximately 50% of the large jobs show similar performance for tra c-containment and non-tra c-containment.

In order to exemplify consequences of tra c-containment versus non-tra c-containment, let us consider some of the large partitions. The 137-node partition 2 in Figure 6.10(d) is highly irregular and forms several obvious bottlenecks. It achieves a $SatRatio_{TC/NTC}$ of only 13% and 29% for uniform and hotspot tra c, respectively. For a number of source-destination pairs the contained paths are significantly longer than the non-contained paths. Let us denote the lower left and upper right corner nodes (0, 0) and (15, 15), respectively. Then, for instance, the contained paths between nodes (2, 3) and (4, 3), between nodes (14, 3) and (14, 5), and between nodes (7, 9) and (11, 9) are 13, 7 and 4 times longer, respectively, than the corresponding non-contained paths. In addition, some of the non-contained paths cross unallocated areas.

The 112-node partition 1 in Figure 6.10(e) has a less irregular shape which to a lesser extent forms bottlenecks. Nevertheless, its $SatRatio_{TC/NTC}$ is only 17% and 32% for uniform and hotspot tra c, respectively. Again, a number of the contained paths are longer than the non-contained paths, and a number of the non-contained paths cross the unallocated areas that surround partition 1.

Partition 2 in Figure 6.10(f) is near sub-mesh shaped and consists of 96 nodes. Some of the contained paths for partition 2 are longer than the corresponding non-contained paths. Let us again denote the lower left and upper right corner nodes (0, 0) and (15, 15), respectively. Then, for instance, the contained path between nodes (2, 4) and (2, 10) is 67% longer than the corresponding non-contained path. When DOR is used, the tra c of partition 2 is contained in the X-direction, but interferes with the tra c of partitions 5 and 6 in the Y-direction.[6] (No communication channels are shared with partition 7.) In addition, part of the tra c of partition 2 traverses unallocated areas. These factors favour non-tra c-containment as long as all partitions communicate in a uniform pattern – then the $SatRatio_{TC/NTC}$ of partition 2 is 64%. For the hotspot tra c pattern, the 15-node partition 6 is a hotspot partition, whereas partitions 2 and 5 still communicate in a uniform pattern. In this case, partition 2 achieves a $SatRatio_{TC/NTC}$ of 153%, which means that tra c-containment is advantageous as it prevents propagation of the congestion of partition 6.

In summary, we found that the benefit of tra c-containment versus non-tra c-containment for a job is dependent on the communication patterns of concurrent jobs. However, the majority of smaller jobs benefit from tra c-containment, regardless of the communication pattern. We identified the following main potential drawbacks related to tra c-containment: bottlenecks due to the shape of partitions; higher load on communication channels; and longer paths.

[6] In our implementation, a path that does not cross a wraparound channel is preferred over a path of equal length that crosses a wraparound channel.

6.2.5 An alternative approach

The performance evaluation presented in Sections 6.2.3 and 6.2.4 showed that an unrestricted contiguous allocation strategy reduces fragmentation, but also that the enforcement of tra c-containment has potential drawbacks due to bottlenecks formed by highly irregular partitions; higher load on communication channels; and longer paths. In the experiments of Sections 6.2.3 and 6.2.4, a scheduled job is always started if a su ciently large contiguous region of nodes is available, even if the region has a shape that obviously forms bottlenecks or implies long contained paths. This section investigates whether, in some cases, a better alternative could be to postpone the allocation, awaiting a more fortunate partition to become available.

We propose a simple extension to the CMC allocation strategy, and evaluate whether it could improve the support of tra c-containment. The main purpose of the extension is to limit unfortunate irregularities of partitions, and, as far as possible, the advantages of CMC with respect to fragmentation should be maintained.

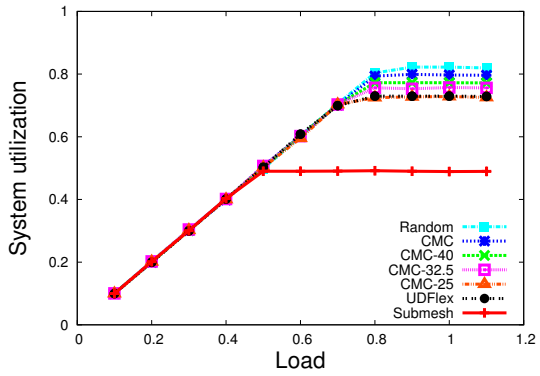
Section 6.2.2 described how the CMC allocation strategy minimizes the dispersal of the nodes that constitute a partition by selecting the allocation alternative with the minimum cost. In the following, this minimum cost of CMC will be denoted $Cost_{CMC}$. The extended approach introduces a new cost metric, the *relative cost*. The relative cost takes the partition size, $|R|$, into account, and is calculated as $\frac{Cost_{CMC} \times 100}{|R|}$. The extended approach simply sets an upper limit, $MaxCost$, on the relative cost, and a scheduled job is started only if the relative cost of an allocation is less than $MaxCost$. The motivation is the assumption that a partition with disadvantages such as severe bottlenecks often has a higher relative cost than a more fortunately shaped partition.

First, we conducted a set of allocation experiments in order to study how the $MaxCost$ a ects the system utilization and queuing time for a 16×16 torus. The extended allocation strategy is denoted CMC-N, where N is a parameter that represents the $MaxCost$. Figure 6.12(a) shows, not surprisingly, that the lower the $MaxCost$ is set, the lower system utilization is achieved. Similarly, Figure 6.12(b) shows that the lower the $MaxCost$ is set, the higher the queuing time becomes. In particular, when comparing CMC-25 with UDFlex, we observe that, for some of the load levels, CMC-25 achieves inferior queuing time.

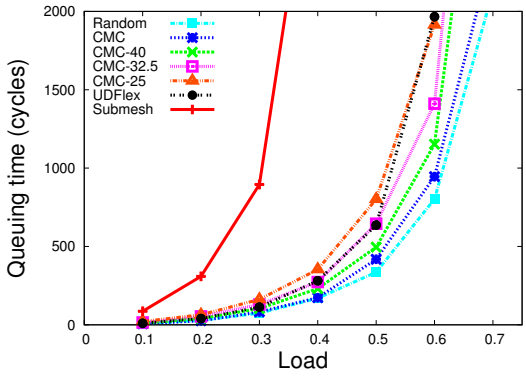
Second, we conducted a set of packet-exchange experiments in order to compare the $SatRatio_{TC/NTC}$ of partitions allocated with, respectively, CMC (no cost limit) and CMC-32.5. Based on the results presented in Figures 6.12(a) and 6.12(b) as well as on a manual inspection of a sample of partitions, a $MaxCost$ of 32.5% appeared to give a reasonable tradeo between fragmentation and irregularity of partitions.

$SatRatio_{TC/NTC}$ for the uniform and hotspot tra c patterns is shown in Figures 6.12(c) and 6.12(e), respectively, for CMC, and in Figures 6.12(d) and 6.12(f), respectively, for CMC-32.5. Only large jobs are included in these experiments since the results presented in Section 6.2.4 showed that a number of large jobs su er from the enforcement of tra c-containment, whereas the majority of the small jobs benefit. For the hotspot tra c pattern, only a section of the y-axis is included, as we are mainly interested in the partitions with a low $SatRatio_{TC/NTC}$. The two horizontal lines indicate a $SatRatio_{TC/NTC}$ of 50% and 75%.

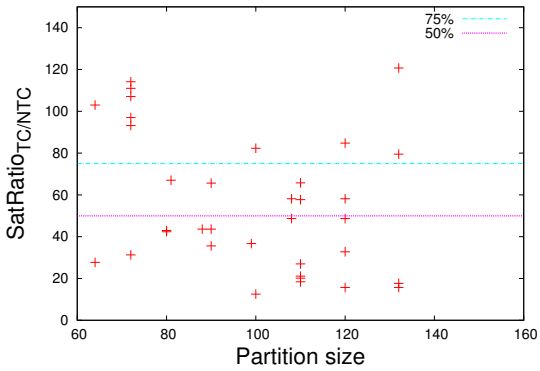
A comparison of Figure 6.12(c) with Figure 6.12(d) and a comparison of Figure 6.12(e) with Figure 6.12(f) indicate that the introduction of a $MaxCost$ of 32.5% reduces the fraction of large jobs that achieve a $SatRatio_{TC/NTC}$ less than or equal to, respectively, 50% and 75%. This is



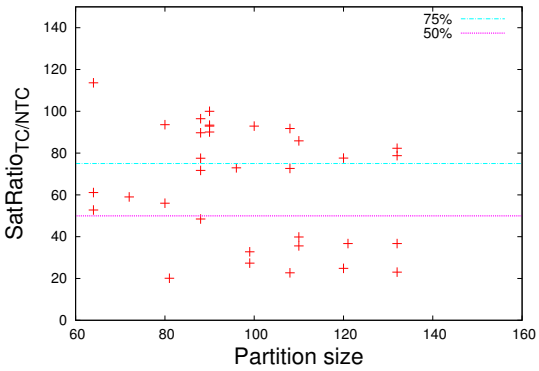
(a) System utilization.



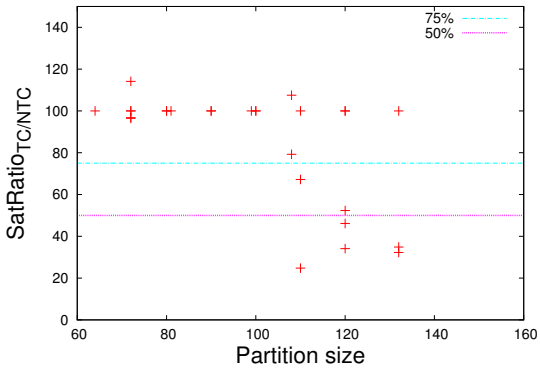
(b) Queuing time.



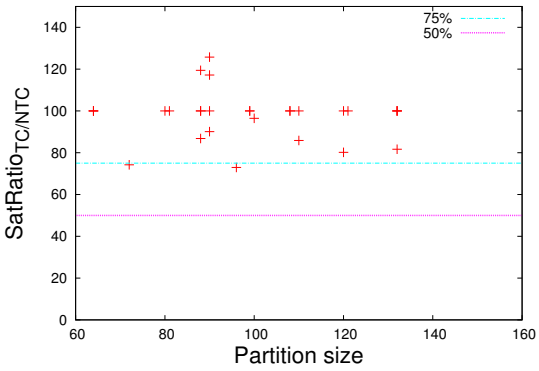
(c) $SatRatio_{TC/NTC}$, uniform tra c, CMC (no cost limit).



(d) $SatRatio_{TC/NTC}$, uniform tra c, CMC-32.5 ($MaxCost = 32.5\%$).



(e) $SatRatio_{TC/NTC}$, hotspot tra c, CMC (no cost limit), section of y-axis.



(f) $SatRatio_{TC/NTC}$, hotspot tra c, CMC-32.5 ($MaxCost = 32.5\%$), section of y-axis.

Figure 6.12: The e ect of limiting the irregularity of partitions by introducing a $MaxCost$, 16×16 torus.

Table 6.2: The fraction of large jobs with a $SatRatio_{TC/NTC}$ less than or equal to, respectively, 50% and 75%.

(a) Uniform tra c.

	No cost limit	**MaxCost = 32.5%**
$\mathbf{SatRatio_{TC/NTC} \leq 50\%}$	54%	33%
$\mathbf{SatRatio_{TC/NTC} \leq 75\%}$	71%	55%

(b) Hotspot tra c.

	No cost limit	**MaxCost = 32.5%**
$\mathbf{SatRatio_{TC/NTC} \leq 50\%}$	14%	0%
$\mathbf{SatRatio_{TC/NTC} \leq 75\%}$	20%	6%

confirmed by Table 6.2 which summarizes the results presented in Figures 6.12(c) – 6.12(f). Thus, limiting the irregularity of partitions reduces the fraction of large jobs for which tra c-containment is a significant disadvantage.

6.3 Related work and our contribution

Our contribution to the field of contiguous processor allocation includes the introduction of two new and flexible approaches. In Section 6.1 we presented the UDFlex processor allocation strategy, and in Section 6.2 we presented a framework for developing processor allocation strategies. Both approaches are topology agnostic and support tra c-containment as well as a high utilization of the available processing resources in a system. Furthermore, both approaches presuppose a close collaboration between an allocator which applies a flexible processor allocation strategy and a routing module which applies a topology agnostic routing algorithm.

To the best of our knowledge, UDFlex was the first processor allocation strategy that exploited the characteristics of a topology agnostic routing algorithm in order to provide flexible partitions as well as tra c-containment. Although both UDFlex and our framework aim at achieving isolation of tra c between partitions while supporting a low level of fragmentation, the approach taken by UDFlex is fundamentally di erent from the approach taken by the framework. UDFlex requires a, possibly irregularly shaped, partition to constitute a valid sub-graph of an overall Up*/Down* [204] graph. Apart from any faults of nodes or communication channels, the overall Up*/Down* graph remains fixed during the operation of a system. Before a job is started on a new partition, reconfiguration is only needed in the form of activation and deactivation of paths in order to keep tra c contained within the partition as well as to ensure connectivity between every pair of nodes that are part of the partition. The framework, on the other hand, allows allocation of any su ciently large contiguous region of nodes to a parallel job, and therefore supports a further reduction of fragmentation when compared to UDFlex. Before a job is started, the framework directs that a topology agnostic routing algorithm must calculate a new set of paths that prevents packets from being routed outside the partition, and that the a ected region of the interconnection network must be reconfigured (static reconfiguration is su cient). The framework is not tied to a particular pair of allocation strategy and routing algorithm, and also in that respect it supports a high degree of flexibility. In particular, as opposed to UDFlex, the framework supports the use of topology agnostic routing algorithms with better performance than Up*/Down*.

In order to evaluate our framework, we introduced and used the contiguous processor allo-

cation strategies CMC and CMC1x1. These strategies are the result of a simple modification of the non-contiguous strategies MC [146] and MC1x1 [34], and allow allocation of an arbitrarily shaped contiguous region of nodes to a parallel job. Not surprisingly, our studies show that, when compared to UDFlex, CMC and CMC1x1 further alleviate fragmentation. After our studies were performed, [110] presented an optimization of MC1x1 that aims to select the best allocation alternative in cases where several allocation alternatives have the same (lowest) cost. We expect that adoption of this optimization could reduce fragmentation even further.

The results of our communication experiments show that there are unpredictabilities related to both tra c-containment and non-tra c-containment, and that the benefit of tra c-containment versus non-tra c-containment for a job is highly dependent on the communication patterns of concurrent jobs. We also introduced a simple extension to the CMC allocation strategy, and this extension shows promise with regard to increasing the benefit of tra c-containment (at the cost of increased fragmentation).

Like CMC and CMC1x1 which are used in the evaluation of our framework, the contiguous version of the Leak strategy [194,195] supports allocation of any su ciently large contiguous region of nodes to a parallel job, regardless of the shape of the region. However, no attempts were made to ensure tra c-containment for the arbitrarily shaped partitions allocated by Leak, and we observe in particular that the contiguous version of Leak is not tra c-contained when Dimension-Order Routing (DOR) [231] is used. (For the communication experiments of [194], XY-routing was used in combination with a non-contiguous version of Leak.)

Arbitrarily shaped contiguous partitions are also supported by the allocation strategy presented in [165] for on-chip environments based on two-dimensional mesh topologies. (Dynamic changes of the size of partitions are also supported.) Like the Leak strategy, [165] is inspired by some of the qualities of fluids. An allocation is compared to a fluid pouring out from an unallocated node until a drop (partition) of the required size is formed. Routing is not discussed in [165], although it seems to be presupposed that a set of paths can be identified within a partition. The allocation strategy presented in [165] is used in the performance evaluation of [166] which introduces a router that supports simultaneous transmission of both packet switched and circuit switched data. For packet switched data transmission, DOR XY-routing is applied in order to avoid deadlock. For circuit switched data transmission, a router may alternate between XY-routing and YX-routing. (Deadlock does not occur as communication channel resources are reserved in advance.) It seems, however, that connections that are set up are allowed to leave and re-enter partitions. Thus, tra c-containment is not guaranteed for arbitrarily shaped partitions for either type of data transmission.

The study presented in [98] concludes that network noise – which includes interference between concurrently running parallel jobs – primarily depends on the type of interconnection network as well as on the routing algorithm and allocation strategy used.

Recall that fragmentation is an inherent issue for contiguous allocation strategies, and that only a non-contiguous allocation strategy can achieve fragmentation-freedom. However, due to our requirement of tra c-containment, non-contiguous processor allocation strategies (such as those presented in [2, 18, 22, 38, 247]) are not a main focus of this thesis.

To the best of our knowledge, few of the existing contiguous processor allocation strategies are independent of topology. Thus, for lack of topology agnostic contiguous allocation strategies, we have given the strategies that target mesh and torus topologies a larger focus in our discussions, although both UDFlex and our framework are topology agnostic.

An example of an approach that may be categorized as highly topology-specific is found in [14], which describes how a multi-toroidal topology (a particular torus topology with additional communication channels that was introduced with the Blue Gene/L supercomputer [29]) may promote tra c isolation between partitions. On the other hand, [59] which focuses on the Blue Gene/P system [101] (Blue Gene/L's successor) shows that allocation of mesh-shaped – rather than torus-shaped – partitions may be advantageous. Di erent racks of nodes may share wraparound channels that only one partition can use at a time. Thus, a job running on a partition A may hinder the start of another job on a di erent partition B if A and B share such wraparound channels.

For mesh and torus topologies, a number of contiguous processor allocation strategies are available in the literature. A selection of those strategies was included – and the strategies' manners of operation were explained – in Section 2.7.1. Many traditional processor allocation strategies, such as First Fit [262], Best Fit [262], Adaptive Scan [61] and Flexfold [91], restrict the allocated regions to sub-meshes (or sub-cubes), and this is also the case for the strategies presented in [4,43,44,46,58,113,147,181,228,229,251,253]. This restriction results in a high level of fragmentation and thus a relatively low utilization of a system's processing resources. The performance evaluation presented in this chapter confirms that, when compared to the allocation of sub-meshes, both of our approaches significantly alleviate fragmentation, and thereby support an improved utilization of the available processing resources in systems based on two-dimensional meshes and tori.

Allocation of sub-mesh shaped regions of nodes in a mesh topology ensures tra c-containment when DOR is used. Thus, assuming the use of DOR, processor allocation strategies such as those proposed in [43, 61, 91, 253, 262] support tra c-containment. However, for a topology which includes wraparound channels, such as a torus or k-ary n-cube, strategies that allocate sub-meshes or sub-cubes can often not guarantee tra c-containment when DOR is used. Recall that in such cases the shortest path between two nodes that are part of the same partition may include intermediate nodes outside the partition, and that packets that are routed outside their partition risk interference with packets belonging to concurrently running jobs. This is an issue for allocation strategies such as [44, 46, 113, 181], whereas the k-ary Partner strategy [251] is able to ensure tra c-containment in a k-ary n-cube topology.

Some of the processor allocation strategies for mesh and torus topologies, such as Frame Sliding [45] and k-ary Partner [251], are not always able to recognize an available sub-mesh or sub-cube, and the result is a high level of fragmentation. UDFlex, on the other hand, can recognize a free valid sub-graph of an Up*/Down* graph given that one is available. Likewise, our framework supports the use of processor allocation strategies, such as CMC and CMC1x1, that can recognize a contiguous region of free nodes given that one is available.

We explained in Section 2.7.1 that some of the allocation strategies for mesh and torus topologies, such as Two-dimensional Buddy [130] and k-ary Partner [251], are a ected by internal fragmentation. Furthermore, although most strategies that allocate sub-meshes or sub-cubes in meshes or tori do not cause as severe internal fragmentation as [130] does, these strategies are nevertheless restricted to allocate a number of nodes that can be expressed as a product of numbers $a \times b \times \ldots$. Consider e.g. a two-dimensional mesh or torus of size $w \times h$ where internal fragmentation may occur if the required number of nodes cannot be expressed as a product $a \times b$, where $a \leq w$ and $b \leq h$, and as a result a higher number of nodes than required must be allocated. UDFlex, on the other hand, can allocate any number of nodes, and thus eliminates this source of internal fragmentation. In addition, our framework supports the use of strategies, such as CMC1x1, that may

allocate any number of nodes.

Many of the traditional contiguous allocation strategies for mesh and torus topologies will not perform satisfactorily for faulty meshes and tori. Therefore, particular solutions such as those described in [40,115] have been proposed for faulty topologies. UDFlex is a topology agnostic processor allocation strategy. Thus, no particular considerations are needed for faulty regular topologies when using UDFlex. Our framework is also topology agnostic, and therefore in itself able to support faulty regular topologies – thus, the fault tolerance abilities depend on the actual processor allocation strategy applied.

Processor allocation has previously been studied for systems that use topology agnostic routing – see e.g. [128] where Up*/Down* routing was used initially (a shortest path routing based on the principles of [55] was later developed). However, although message contention and processor locality were discussed in [128], the properties of the Up*/Down* routing algorithm were not utilized to realize tra c-containment within partitions.

We mentioned in Section 2.5 that the Logic-Based Distributed Routing (LBDR) [73] algorithm targets on-chip networks that conform to a defined set of regular and semi-irregular mesh-based topologies. LBDR can be used for topologies where the set of paths connecting all nodes corresponds to the shortest paths in an imaginary surrounding regular two-dimensional mesh. This includes topologies shaped as the characters "+", "q", "p", "b" and "d". LBDR is thus not topology agnostic, but can, for the defined set of topologies, implement the forwarding restrictions imposed by such routing algorithms as Up*/Down* and Segment-Based Routing [151]. Furthermore, at the expense of a more complex switch design, an extension of LBDR, uLBDR [192], increases flexibility by supporting any topology than can be derived from a two-dimensional mesh. It was briefly mentioned in [73] that LBDR can implement tra c isolation (containment) for a set of regular and semi-irregular partitions. This was further developed and investigated in [191] for bLBDR, a broadcast and multicast extension of LBDR. With LBDR, each port of a node has a so called *connectivity bit* which states whether or not the node is connected to another node through this port. Exploiting this feature of LBDR by simply resetting the connectivity bits for the ports along the borders of network regions, bLBDR implements tra c isolation for the regions.

Like UDFlex and our framework, [191] discusses how the routing function should be taken into account in order to achieve connectivity and tra c isolation within partitions. For a network where the forwarding restrictions of Up*/Down* is applied, [191] also provides an example which e ectively illustrates UDFlex allocation.[7] Thus, algorithmic routing schemes such as LBDR, uLBDR, bLBDR and their recent successors mentioned below could promote the use of UDFlex in multicore chip environments where the use of routing tables may be disadvantageous.

As in the performance evaluation of our framework, tra c isolation versus tra c contention is a topic of both [74] and [191]. The focus of [191] is di erent from our focus. Whereas we study the results of tra c-containment versus non-tra c-containment for the exact same set of partitions (and for unicast tra c), [191] studies broadcasts within defined domains (partitions) versus broadcasts across an entire 8×8 mesh network. Furthermore, in the evaluation of our framework we investigate a significant number of arbitrarily shaped partitions, whereas only four partitions are included in the performance evaluation of [191] (three of those partitions are sub-mesh shaped and the fourth partition is shaped like the character "d"). As opposed to UDFlex and our framework, bLBDR is not topology agnostic. Furthermore, in its original form bLBDR does

[7] UDFlex [225] was published before [191].

not support arbitrarily shaped partitions. Recently, however, the mechanisms of bLBDR (and a successor denoted Signal Bit Based Multicast) and the mechanisms of uLBDR have been combined into eLBDR which supports irregular partitions in a two-dimensional mesh-based topology [193]. Like the family of algorithms based on LBDR, another logic-based routing algorithm – Flexible DOR (FDOR) [209] – is also able to support tra c isolation within non-rectangular partitions in a mesh-based on-chip environment.

Similar experiments as those presented in [191] and [225] were also included as examples in [74]. In addition, [74] briefly included simulation results which demonstrated that a lack of tra c isolation caused reduced throughput and increased latency for a partition when its neighbouring partition was congested. Better throughput and latency were achieved when LBDR was used to implement the forwarding restrictions of the Up*/Down* and Segment-Based Routing algorithms, both of which provided routes that ensured tra c isolation within each of the partitions. Whereas the evaluation of our framework included a large number of arbitrarily shaped partitions, only two partitions (one sub-mesh shaped and one shaped as the character "b") were studied in the examples included in [74].

Like bLBDR, the strategy recently presented in [239] is based on LBDR and realizes tra c isolation between network regions in an on-chip environment by resetting the connectivity bits for the node ports along the borders of the regions. The partitions can take on any of the shapes that are supported by LBDR. Whereas only a few static scenarios are considered in [239], reconfiguration of LBDR's connectivity bit and two routing bits (per node port) is proposed in [238] in order to support tra c-containment for a changing set of partitions. As with our framework, which is based on a similar principle, static reconfiguration is su cient. The performance evaluations of [239] and [238] also have similarities with the performance evaluation of our framework. For a given set of partitions, performance in the tra c-contained case is compared with performance in the non-tra c-contained case. (In addition, a random assignment of nodes to applications is included as a benchmark in [239] and [238].) Other metrics are used in [239] and [238] than in our study. Moreover, the conclusions of [239] and [238] appear to be more unambiguously in favour of tra c-containment than our conclusions are. We believe the main reason is that, when compared to our experiments, the experiments of [239] and [238] include only small partitions (of size 4 nodes in [239] and of size 2 – 8 nodes in [238]). Recall that our study also shows that the majority of smaller jobs benefit from tra c-containment. The network topology modelled in [239] and [238] is a 4×4 mesh, as opposed to the 16×16 mesh and torus topologies considered in our study. On the other hand, the workload models used in [239] and [238] are in general more realistic than our synthetic workload models.

Several other studies which target the consequences of interference between tra c belonging to di erent partitions are available in the literature, and in the following we include some of the most relevant studies. In [137] the e ect of contention between messages belonging to concurrently running jobs is evaluated for several non-contiguous processor allocation strategies, using the contiguous First-Fit allocation strategy as a benchmark. Furthermore, the impact of di erent message sizes is studied. Similar experiments are conducted in [194], although their focus is different from the focus of [137]. In [194] a version of the Leak processor allocation strategy that allows non-contiguous partitions is implemented in a distributed manner. Nodes exchange control messages as part of the allocation algorithm, and the focus of the experiments is on potential contention between such control messages and the messages related to the execution of the parallel

jobs (assuming that a separate network for control tra c[8] is not available). Our study, on the other hand, focuses on contiguous processor allocation and assumes that control tra c is transmitted in a separate network.

Advantages and disadvantages of contention-free allocation versus low-contention allocation and fragmentation-free allocation are discussed in [156]. Among their conclusions is the observation that contention-free allocation is advantageous for applications which are communication intensive or exchange large messages. Our focus is di erent from the focus of [156]. We study the costs and benefits of tra c-containment versus non-tra c-containment for the exact same set of partitions, whereas [156] compares partitions (of the same size) generated using di erent allocation principles (contention-free, low contention, and fragmentation-free allocation). In [156], two concurrent jobs – each of size 8 nodes – are running in a 4×4 mesh or in a 16 node multistage interconnection network, which means that all the nodes are busy at any time. Our study includes a larger interconnection network and a large set of partitions, where the contiguous partitions are of variable size and shape. Furthermore, in our study, not all the nodes are busy at all times. We observed in Section 6.2.4 that this may have an impact on the conclusions about the advantages of tra c-containment versus non-tra c-containment.

Interference between concurrent partitions is also discussed in [249] which addresses a Cray XT3 system [51] (a three-dimensional torus). Several optimized allocation strategies are suggested in [249]. The di erent allocation strategies target di erent categories of jobs, and partitions may be either sub-mesh shaped, non-regular contiguous or non-contiguous. When compared to a non-contiguous allocation strategy that was originally used, the use of an optimized allocation strategy with a higher degree of contiguity results in significant performance improvements for communication intensive applications. In [246] the implementation of the optimized allocation strategies is further elaborated on. We observe, however, that the optimized allocation strategies in general cannot guarantee tra c-containment when DOR is used.

6.4 Critique

Compared to many of the strategies that allocate sub-meshes, both UDFlex and our framework are expected to have a higher allocation and communication overhead. The UDFlex example implementation discussed in Section 6.1.2 has a higher computational complexity than many of the traditional strategies that allocate sub-mesh shaped partitions have. However, as discussed in Section 6.1.2, a simpler "first-fit" approach to the implementation of UDFlex is possible, and such an approach may reduce the computational complexity of UDFlex (probably at the expense of increased fragmentation). UDFlex is tied to the Up*/Down* routing algorithm which has known performance issues. Most importantly, the Up*/Down* routing algorithm is prone to congestion in the area around the root node of the Up*/Down* graph. In addition, due to prohibited down-link to up-link turns, a legal path from one node to another is not necessarily the shortest path in the physical topology. (For interconnection networks that use virtual cut-through switching, the latter concern is normally of less importance.) Two decades after its release, the topology agnostic Up*/Down* routing algorithm still plays an important part in the operation of systems that contain a non-regular interconnection network – possibly as a result of faults in an originally regular

[8] Recall that *control tra c* refers to tra c that is not part of the intra-partition tra c related to the execution of a parallel job.

network. Our results indicate that, when compared to traditional sub-mesh allocating strategies, an allocation and communication overhead of at least 60% could be tolerated before the advantages of UDFlex with respect to fragmentation are neutralized. Thus, we believe that UDFlex, which takes advantage of the characteristics of the Up*/Down* routing algorithm in order to realize tra c-containment, is a useful contribution.

For our framework, the allocation and communication overhead will depend on the chosen pair of allocation strategy and routing algorithm. However, regardless of the pair of algorithms chosen, the framework directs that in order to ensure tra c-containment a static reconfiguration process must in general be performed before a new job is started. Such a reconfiguration will inevitably imply a certain amount of allocation overhead when compared to traditional allocation strategies (which do normally not require reconfiguration).

An increased allocation overhead will in general be more acceptable for jobs that are running for a long – rather than a short – period of time. When compared to traditional contiguous strategies that allocate sub-mesh shaped partitions, both UDFlex and our framework have significant advantages with respect to reduced fragmentation. Thus, the choice between using a traditional allocation strategy versus using one of our new approaches implies a tradeo where tolerance towards such factors as fragmentation, allocation overhead and communication overhead should be taken into account, as should the expected running times of jobs.

As opposed to the use of many of the strategies that allocate sub-meshes, the use of UDFlex and our framework presupposes flexibility with respect to the relative distances and locations of the nodes that constitute a partition. This is also the case for a number of other processor allocation strategies, however – see e.g. Random [137], MC [146], ARS [4] and Flexfold [91].

Our strong focus on tra c-containment could perhaps be criticized. Tra c-containment presupposes a contiguous processor allocation algorithm. As opposed to non-contiguous allocation algorithms, contiguous allocation algorithms cannot be fragmentation-free, and are thus bound to achieve a lower utilization of a system's processing resources. Furthermore, as the communication experiments conducted for our framework demonstrate – in particular for large partitions – tra c-containment is not always beneficial over non-tra c-containment. Our results show that, when tra c-containment is enforced within unfortunately shaped partitions, a significant communication overhead may result from such factors as bottlenecks, higher load on communication channels and longer paths. However, our results also indicate that, rather than uncritically assigning an unfortunately shaped partition to a scheduled job, enforcing an upper limit on the irregularity of partitions reduces the fraction of jobs for which tra c-containment is disadvantageous. (Recall that we developed a simple metric in order to quantify the irregularity of partitions.) Increased fragmentation is a cost of such a more selective allocation strategy, and a reasonable balance must be found between an acceptable level of fragmentation and an acceptable level of partition irregularity. The benefit of tra c-containment over non-tra c-containment for a job is highly dependent on the communication patterns of concurrently running jobs. An important advantage of tra c-containment is that it prevents propagation of congestion from one partition to other partitions. For environments which require that the communication overhead of one job is independent of concurrently running jobs, or for environments where a firm insurance is needed that one job cannot analyze or tamper with tra c belonging to other jobs, a contiguous processor allocation strategy that ensures tra c-containment may be preferred. Consider as an example applications running in a cloud computing [245] data center. Such applications are diverse and generally not under control

of the cloud provider. When applications ranging from those in a debugging phase of development to applications in the high performance computing segment are running concurrently, isolation of the tra c belonging to di erent partitions becomes a crucial issue. For environments with less strict requirements on tra c isolation, on the other hand, a non-contiguous processor allocation strategy may be advantageous. A logical separation of tra c could then be realized if more than one virtual channel is o ered per physical communication channel.

In their current versions, both UDFlex and our framework target homogeneous environments where all the resources are equivalent and where single processors are the only type of resource eligible for allocation. In addition to single processors, real systems may consist of various other types of resources, such as multicore chips, hardware accelerators, gateway nodes and storage nodes. Although a homogeneous system model is a simplification of most real systems,[9] such a homogeneous environment is commonly assumed in studies on processor allocation. Thus, the adoption of this assumption allows us to compare our new solutions with established processor allocation strategies.

In the evaluation of UDFlex we did not perform packet-exchange experiments. On the other hand, we conducted a set of experiments where the probable allocation and communication overhead of UDFlex were represented by an extended service time (running time) for the jobs. These experiments allowed us to quantify the amount of allocation and communication overhead that can be tolerated for UDFlex until its advantages (with respect to fragmentation) over traditional sub-mesh allocating strategies are eliminated.

In the evaluation of our framework, separate simulator models were used for the allocation experiments and the communication experiments, and the packet-exchange experiments were performed only on a number of snapshots output from the allocation experiments. Section 3.1 explains the reasons behind such a simulator model setup, and the usability of the setup was further discussed in Section 3.1.3.

The metrics used in the evaluation of our framework, $Saturation_*$ and $SatRatio_{*/*}$, are perhaps relatively complex and not immediately intuitive. We did, however, consider using an alternative pair of metrics – $Throughput_*$ and $ThrRatio_{*/*}$, where, for a partition P, $Throughput_*$ is the total number of packets received by all nodes of P and the relation between $ThrRatio_{*/*}$ and $Throughput_*$ corresponds to the relation between $SatRatio_{*/*}$ and $Saturation_*$. For a few jobs, we observed that $Throughput_{All}$ was significantly higher than $Throughput_{One}$. That is, for such a job, j, a significantly higher throughput was observed when all the jobs of a snapshot ran together than when j ran alone. We found the following explanation for this counterintuitive e ect, which was only observed for large jobs: Interfering tra c from one or more of the other jobs of the snapshot unintentionally imposes congestion control on j by slowing down tra c for part of j, and thereby allowing speed-up of tra c for other parts of j. This means that the $ThrRatio_{All/One}$ metric has significant limitations. It assumes that the tra c of one node of a partition P is fully independent of the tra c of the other nodes of P. This assumption is in general not valid for jobs that require some kind of barrier synchronization. Thus, we believe that $Saturation_*$ and $SatRatio_{*/*}$ – which register when the first source node of a partition reaches saturation – constitute a more representative pair of metrics.

The use of mesh and torus topologies in the experiments conducted to evaluate UDFlex and our framework – both of them topology agnostic approaches to processor allocation – could perhaps be

[9] An example of a homogeneous system is Intel's [106] terascale prototype processor [100, 244] with its 80 homogeneous cores.

criticized. We believe, however, that meshes and tori are relevant topologies for our experiments, and give reasons for this opinion in Section 3.1.3.

Our framework was evaluated on 16×16 mesh and torus topologies. The reasons for not including larger topologies in the evaluation are related to the feasibility of the communication experiments. First, for every snapshot, we performed a relatively long-running simulation experiment where all the partitions of the snapshot ran together. Then, for every partition of every snapshot, we performed a relatively long-running simulation experiment where the partition ran alone on the 16×16 mesh or torus. Thus, a large number of simulation experiments were needed. Given similar sizes of the partitions, the number of partitions per snapshot would have increased for larger topologies (recall that the snapshots were collected for a load of 1.0). Conducting the communication experiments for significantly larger topologies was therefore considered too demanding.

The processor allocation strategies used in the evaluation of our framework, CMC and CMC1x1, are based on MC [146] and MC1x1 [34], respectively, which were developed for mesh and torus topologies. Thus, CMC and CMC1x1 are not topology agnostic, and the use of such allocation strategies could be objected to since the framework itself supports topology agnosticism. To the best of our knowledge, few existing contiguous processor allocation strategies are independent of topology. We found that CMC and CMC1x1 are suitable examples of strategies that can be used with our framework. CMC and CMC1x1 are conceptually simple; able to allocate partitions of any shape; able to recognize any available and su ciently large contiguous region of nodes; and able to minimize the dispersal of the allocated nodes. On the other hand, we could probably have realized a topology agnostic allocation algorithm by basing the calculation of the contiguous partitions on ordinary breadth-first searches around the free nodes. We do not expect, however, that the use of such an algorithm would have altered our conclusions.

In order to increase the benefit of tra c-containment, an upper limit on the irregularity of partitions created by CMC was introduced in Section 6.2.5. The enforcement of such an upper limit was evaluated for only a limited scenario, however. Only the torus topology and large jobs were studied, and the number of jobs included in the experiments was restricted. Nevertheless, the promising results motivate further work aiming to develop more sophisticated allocation strategies which increase the benefit of tra c-containment.

6.5 Future work

Potential hindrances and opportunities for the development of cloud computing are discussed in [16], which concludes with an optimistic prediction of the future of cloud computing. We argued in [142] that emerging cloud computing environments need new and more flexible solutions to the problem of partitioning (or virtualization) of their resources. This problem is related to processor allocation, but is in general more challenging. Recall that for the development of UDFlex and our framework we assumed a homogeneous environment where all the resources are equivalent and where single processors are the only type of resource to be allocated. This is a common assumption in traditional studies on processor allocation. A cloud computing data center, on the other hand, represents a more heterogeneous environment, where the existing resources can roughly be grouped into the following three categories: computing nodes – ranging from single processors to di erent types of multicore processors; storage nodes which include storage media (such as disks or tapes) with a range of di erent characteristics; and gateway nodes towards external networks.

During the operation of a cloud computing data center, the resources (which are linked by an interconnection network) are dynamically partitioned into *virtual servers*. The virtual servers are assigned on demand to customers, and released when no longer needed. A customer is typically billed according to the amount of resources assigned as well as the duration of the assignment.

The space-sharing model is commonly assumed in the literature on processor allocation, and is also adopted by us for the development of UDFlex and our framework. Recall that space-sharing implies that a set of resources is exclusively assigned to a single parallel job, which runs uninterruptedly until completion. For several reasons, a space-sharing model may be unsuitable for a cloud computing environment. A cloud computing data center will typically demand a high profitability, and thus also a high utilization of its resources. Overlapping partitions may increase the utilization of resources – or grant several jobs access to a scarce resource. The resources that are used by more than one partition must then be accessed in a time-sharing fashion (that is, the partitions alternate in using the shared resource). Migration of running jobs may be desirable, for instance in order to escape from faulty resources. Moreover, migration of running jobs could also allow – or enhance – the placement of a scheduled job, and thereby increase the overall utilization of the system's resources. During the lifetime of a virtual server, its amount of resources may need to be scaled up or down. For instance, a scale up of a virtual server could be the result of a customer requesting additional resources, whereas a scale down could be the result if a higher priority virtual server required additional resources (possibly due to faults in some of its assigned resources).

Although we refer to UDFlex and our framework as topology agnostic, their usefulness in a system based on a multistage interconnection network is questionable. We argue in [142] that multistage interconnection networks are highly relevant for cloud computing environments, and such topologies should be considered during the development of new resource partitioning strategies for cloud computing data centers.

A number of di erent applications – ranging from applications in the high performance computing segment to applications in a debugging phase of development – may run concurrently in a cloud computing data center. In general, the applications running in a cloud computing data center are not under control of the cloud provider; the quality of the applications cannot be guaranteed; and misbehaving applications must be expected. Such an environment poses a number of challenges, with respect to security, anonymity as well as performance.

Tra c-containment could have solved the security and anonymity issues by ensuring isolation between partitions. If such isolation is implemented, one job cannot analyze the tra c – nor tamper with the resources, application, data or tra c – belonging to a concurrently running job. However, although tra c-containment could also have ensured that the communication overhead for a job is independent of concurrently running jobs, the results presented in Section 6.2.4 demonstrate that the enforcement of tra c-containment does not always result in a lower communication overhead for a job. Furthermore, due to such factors as high requirements on resource utilization and a probable need to share scarce resources among a number of partitions, a cloud computing data center can in general not rely on tra c-containment. Thus, other means must in general be applied in order to isolate the tra c that belongs to di erent partitions. For instance, the use of virtual channels could promote tra c isolation on shared communication channels. A combination of Multiple Roots (MRoots) [71,143,144] and UDFlex could potentially support tra c-containment within each virtual layer for a system that allows overlapping partitions. (Recall from Section 2.8.1 that MRoots is a topology agnostic routing algorithm that calculates a separate directed acyclic

graph (DAG) for each virtual layer and distributes the root nodes of each DAG throughout the network topology.)

For several reasons discussed above, our new solutions for processor allocation presented in this chapter are not directly portable to a cloud computing environment. However, we believe that the main idea behind UDFlex and our framework – utilization of a topology agnostic routing algorithm in order to achieve flexible partitioning – may also be useful in the development of flexible and fault-tolerant partitioning solutions for cloud computing data centers. Furthermore, in order to minimize the interference between jobs, maximizing the degree of isolation between partitions may be advantageous even if complete tra c-containment is not feasible. (Recall that a misbehaving job could induce severe performance degradation in concurrently running jobs.)

In addition to flexible partitioning of resources, in [142] we emphasized predictable service and fault tolerance as important challenges to be addressed for an interconnection network in a cloud computing environment. The jobs running in a cloud computing data center will typically have di erent service requirements and importance. In order to successfully o er predictable services in such an environment, mechanisms for implementing service di erentiation at the level of partitions are needed, as well as mechanisms for congestion control and admission control. A fault in a resource node, switch or communication channel of a cloud computing data center will probably a ect only a small subset of the running jobs. Thus, a fault tolerance mechanism for such an environment should be aware of partitions, and aim to minimize the negative influence on jobs that are not directly a ected by the fault. Any reassignment of resources or termination of jobs should be performed in accordance with the importance of the jobs involved. Finally, the combination of mechanisms for flexible partitioning of resources, predictable service and fault tolerance is an important challenge facing an interconnection network in a cloud computing environment.

Chapter 7

Conclusion

Topology agnostic methods for routing and reconfiguration of interconnection networks are main topics of this thesis. In addition, we utilized such methods to realize flexible processor allocation strategies that support tra c-containment (tra c isolation) within partitions. The solutions proposed in this thesis do not target specific computing systems, but are versatile mechanisms that can be used for a number of di erent computing systems such as supercomputers, data centers and multicore processor integrated circuit chips.

7.1 Routing

We performed a systematic evaluation of a number of existing routing algorithms in order to identify an appropriate routing algorithm for the Advanced Switching Interconnect (ASI) [150] technology. The motivation for such an evaluation was that the ASI specification [17] did not prescribe a particular routing algorithm. (Later, the ASI specification process was discontinued, and the ASI technology is now incorporated in Dolphin Express [122].) We argued that a routing algorithm for ASI should support deterministic routing, be topology agnostic and deadlock-free; provide shortest paths and high e ciency for regular topologies; and not require transitions from one virtual layer to another. One routing algorithm, Layered Shortest Path (LASH) [144, 218], was found to fulfil all of these criteria. We proposed that LASH is used with ASI, and discussed main principles regarding the realization of LASH in ASI.

LASH is a topology agnostic routing algorithm that uses virtual layers for deadlock avoidance. In addition to being essential mechanisms that enable communication in interconnection networks with irregular topologies, topology agnostic routing algorithms may be important fault tolerance mechanisms for networks with regular topologies. We studied the usability of LASH as a static fault tolerance mechanism, and demonstrated how LASH supports a graceful degradation of performance as an increasing number of switches fail in networks with mesh and torus topologies.

We also proposed an optimization that allows LASH to select the same paths, and thus achieve the same performance, as Dimension-Order Routing (DOR) [231] in a fault-free mesh topology. With this optimization a single virtual layer ensures deadlock-freedom for a fault-free mesh. Furthermore, the abilities of LASH as a static fault tolerance mechanism are maintained.

The main weakness of LASH is perhaps that the number of virtual layers needed for practical applications is not fully predictable. We studied LASH's requirements of virtual layers for regular topologies such as fat-trees, meshes and tori. In addition to the fault-free topologies, faulty network

switches were considered for the latter two categories of topologies. The results show that, in most cases, the number of virtual layers required is within the number of virtual channels (VCs) supported by the ASI specification (even though some of the investigated topologies are larger than the maximum size of an ASI fabric).

In order to support service differentiation when LASH is used with ASI, we proposed a novel layer to priority-class mapping. For cases where LASH needs less than half of the available number of VCs to ensure deadlock-freedom, at least two different priority levels can be provided for service differentiation. (In the cases where LASH settles for a single layer, no such novel mappings are required.)

We explained how most of our considerations regarding routing in ASI are also relevant for routing in InfiniBand [105]. The requirements for a routing algorithm to be used with ASI are also reasonable guidelines for selecting a routing algorithm to be used with InfiniBand. Furthermore, a number of issues related to realization of LASH in ASI are also relevant for realization of LASH in InfiniBand. In addition, the results of our evaluation of the performance of LASH in ASI are also representative of the performance of LASH in InfiniBand. Our considerations were taken into account when LASH was included into the OpenFabrics Enterprise Distribution [163] software stack for InfiniBand.

7.2 Reconfiguration

The change-over from one routing function to another – reconfiguration – is a deadlock prone operation. Some reconfiguration strategies include a deadlock avoidance mechanism that causes a significant performance degradation, and thus a reduced network service, during the routing change-over. We proposed a new dynamic reconfiguration method called RecTOR, and expanded the area of application for Overlapping Reconfiguration (OR) [139,140] – one of the most versatile and efficient reconfiguration methods available in the literature.

RecTOR is based on a simple principle which, during a transition from one routing function to another, allows packets routed according to either – or both – of the routing functions to coexist in the network without restrictions. Our results confirm that RecTOR ensures deadlock-freedom during a reconfiguration process without causing performance penalties. RecTOR is based on the flexible Transition-Oriented Routing (TOR) algorithm [200] which, given sound path selection, achieves excellent performance. TOR depends on VCs for deadlock avoidance, and RecTOR does not require additional VCs during a reconfiguration process. RecTOR is applicable for both source and distributed routing systems.

OR was previously only usable for distributed routing systems. We proposed an adaptation of OR in order to enable using it also in source routing systems. A main part of the adaptation is a new procedure that allows source routing switches to acquire channel dependency information while forwarding data packets.

For source routing systems, we also proposed an optimization of the OR algorithm in order to improve the network service offered to running applications during a reconfiguration. OR depends on special packets – tokens – for deadlock avoidance, and our results demonstrate how the optimization, which implies a relaxed token propagation procedure, reduces performance penalties.

We studied how different degrees of synchronization of the token injection may affect the performance of OR, and concluded that the degree of synchronization has a significant impact on the

increased latency, decreased throughput and congestion effects related to a reconfiguration process managed by OR.

A comparison of OR and RecTOR shows that there are advantages and disadvantages related to either of the methods. OR can support in-order packet delivery, whereas RecTOR cannot. OR can be used in systems where VCs are not available for the routing function, whereas RecTOR is based on TOR which depends on VCs for deadlock avoidance. OR can be used between any pair of routing algorithms, and is in this respect more flexible than RecTOR is. On the other hand, RecTOR supports an uninterrupted network service during a reconfiguration process, which OR in general cannot. RecTOR does not require additional functionality in the network switches (except perhaps for some packet loop prevention mechanism for distributed routing environments), and is thus simpler to realize than OR which requires relatively complex switches. Finally, as opposed to the performance of OR, the performance of RecTOR is independent of how synchronized the start of a reconfiguration process is.

7.3 Processor allocation (virtualization)

Minimization of fragmentation and of interference between different partitions are two main challenges for a processor allocation algorithm. Traffic-containment ensures isolation of intra-job traffic within each partition. Unfortunately, fragmentation – and thus a suboptimal utilization of a system's processing resources – is a consequence of enforcing traffic-containment (which presupposes contiguous processor allocation). Many of the existing processor allocation strategies cannot guarantee traffic-containment. Moreover, a number of allocation strategies are rigid approaches which achieve traffic-containment only at the cost of a significantly reduced resource utilization. We introduced two new and flexible approaches – the UDFlex processor allocation strategy and a framework for developing processor allocation strategies. Both approaches are conceptually simple, and support traffic-containment as well as a high utilization of the available processing resources in a system.

Both UDFlex and the framework presuppose a close collaboration between an allocator which applies a flexible processor allocation strategy and a routing module which applies a topology agnostic routing algorithm (and, when required, initiates a reconfiguration). Allowing arbitrarily shaped partitions is a key to reduce fragmentation. Although both UDFlex and the framework exploit the characteristics of a topology agnostic routing algorithm in order to provide traffic-containment within arbitrarily shaped partitions, their approaches are fundamentally different. UDFlex in some cases requires a simple activation and deactivation of paths in order to ensure traffic-containment and connectivity within a partition. The framework, on the other hand, directs that a new set of paths must be calculated for a new partition, and that the affected region must be statically reconfigured. UDFlex assumes the use of the Up*/Down* routing algorithm [204], and allocates a valid sub-graph of an overall Up*/Down* graph to a parallel job. The framework, on the other hand, is not tied to a particular pair of allocation strategy and routing algorithm, and allows allocation of any sufficiently large contiguous region of nodes to a parallel job. Thus, the framework supports the use of topology agnostic routing algorithms with better performance than Up*/Down*, and the use of contiguous processor allocation strategies which, when compared to UDFlex, further reduce fragmentation. Examples of the latter include CMC and CMC1x1, allocation strategies which were introduced in this thesis, and which result from simple modifications of the non-contiguous

allocation strategies MC [146] and MC1x1 [34], respectively.

In addition to supporting allocation of arbitrarily shaped partitions, two other qualities of UDFlex and the framework contribute to low fragmentation. First, UDFlex can recognize a free valid sub-graph of an Up*/Down* graph given that one is available, and the framework supports the use of processor allocation strategies that can recognize any suitable contiguous region of free nodes. Second, UDFlex can allocate any number (not restricted to e.g. a product of numbers) of nodes, and the framework supports the use of processor allocation strategies that can allocate any number of nodes.

On the negative side, the use of either UDFlex or the framework is expected to entail a relatively high allocation and communication overhead. Our results indicate, however, that – when compared to traditional sub-mesh allocating strategies – an allocation and communication overhead of at least 60% could be tolerated before the advantages of UDFlex with respect to fragmentation are neutralized. In addition to the overhead related to reconfiguration, the allocation and communication overhead for the framework will depend on the chosen pair of allocation strategy and routing algorithm.

When considering fragmentation and communication overhead (and ignoring a potential allocation overhead), it is quite obvious that allowing allocation of arbitrarily shaped partitions is beneficial when applications are not communication intensive. For communication intensive applications, we investigated the costs and benefits of tra c-containment (as prescribed by the framework) versus non-tra c-containment. Our results show that there are unpredictabilities related to both approaches to communication, and that the benefit of tra c-containment versus non-tra c-containment for a job is highly dependent on the communication patterns of concurrent jobs. The main advantages of tra c-containment are that the communication overhead of one job is independent of concurrently running jobs, and that one job cannot analyze or tamper with the tra c belonging to other jobs. On the other hand, we identified bottlenecks; higher load on communication channels; and longer paths as main potential drawbacks related to the enforcement of tra c-containment within unfortunately shaped partitions. Subsequently, we introduced a simple extension to the CMC allocation strategy that shows promise with regard to increasing the benefit of tra c-containment (at the cost of a higher fragmentation).

Many of the traditional processor allocation strategies for regular topologies will not perform satisfactorily for faulty topologies. Thus, particular solutions may be required when faults emerge. UDFlex, on the other hand, is a topology agnostic processor allocation strategy, and is thus also useful for non-regular topologies that may result from faults in regular topologies. For the framework, which supports various processor allocation strategies (including topology agnostic ones), the fault tolerance abilities depend on the actual allocation strategy applied.

7.4 Future work

Topology agnostic methods have a number of advantages, and are therefore expected to be attractive mechanisms for emerging and future computing systems. Such methods are flexible with respect to the topology of an interconnection network that connects various parts of a computing system, and thus simplify addition or removal of network components as well as merging or splitting of systems. Furthermore, topology agnostic methods may also simplify the provisioning of fault tolerance, and this aspect becomes even more important as the probability of component faults increases when

computing systems grow larger.

E cient routing and reconfiguration are expected to play important roles in the operation of the interconnection networks of future and emerging computing systems. For the routing and reconfiguration algorithms under study in this thesis – LASH, OR and RecTOR – some challenges remain. Improved predictability with respect to LASH's usage of virtual layers is desired. For OR, an open question is whether a small change of the characteristics of an interconnection network, such as the introduction of virtual output queues, could increase the benefit of the optimization of OR for scenarios where the gain is currently limited. Furthermore, in order to provide an even more powerful fault tolerance mechanism than RecTOR alone, RecTOR could be combined with the main principles of a fault-tolerant routing algorithm such as Transition-Based Fault-Tolerant Routing (TFTR) [154].

Stricter requirements on energy-e cient operation of computing systems are expected in the future. Thus, in addition to an e cient operation of an interconnection network, a high utilization of a system's computing resources may become even more important in future systems than it is in current systems.

Our new processor allocation strategy, UDFlex, could be used in a number of di erent computing systems. Nevertheless, its (and Up*/Down*'s) dependency on routing tables may be disadvantageous for some multicore chip environments. We believe that algorithmic routing schemes, such as the family of algorithms [193] based on Logic-Based Distributed Routing (LBDR) [73] which is able to implement the forwarding restrictions of Up*/Down*, could promote the use of UDFlex in multicore chip environments where routing tables are undesirable.

We argued that cloud computing environments need new and more flexible solutions to the problem of partitioning (or virtualization) of their resources [142]. This problem is related to processor allocation, but is in general more challenging. For the development of UDFlex and our framework we adopted the common assumptions of a space-sharing model and of a homogeneous environment where all the resources are equivalent. These assumptions are in general not useful for a cloud computing environment. A cloud computing data center represents a heterogeneous environment. It typically demands a high profitability, and thus a high utilization of its resources. Such an environment may require overlapping partitions; migration of running jobs; as well as up or down scaling of the amount of resources allocated to a running job. Furthermore, di erent jobs may have di erent degrees of priority. In addition, the applications running in a cloud computing data center are not under control of the cloud provider; the quality of the applications cannot be guaranteed; and misbehaving applications must be expected. Such an environment poses a number of challenges, with respect to security, anonymity as well as performance. Tra c-containment – which could have solved the security and anonymity issues and ensured that the communication overhead for a job is independent of concurrently running jobs – is in general not an option for a cloud computing environment. Nevertheless, maximizing the degree of isolation between partitions may be desirable, and for the development of flexible and fault-tolerant partitioning solutions the use of topology agnostic routing methods may be advantageous.

Bibliography

[1] I. Ababneh. An e cient free-list submesh allocation scheme for two-dimensional mesh-connected multicomputers. *Journal of Systems and Software*, 79(8):1168–1179, August 2006.

[2] I. Ababneh. Availability-based noncontiguous processor allocation policies for 2D mesh-connected multicomputers. *Journal of Systems and Software*, 81(7):1081–1092, July 2008.

[3] I. Ababneh. On submesh allocation for 2D mesh multicomputers using the free-list approach: Global placement schemes. *Performance Evaluation*, 66(2):105–120, February 2009.

[4] I. Ababneh, S. Bani-Mohammad, and M. Ould-Khaoua. All shapes contiguous submesh allocation for 2D mesh multicomputers. *International Journal of Parallel, Emergent and Distributed Systems*, 25(5):411–421, October 2010.

[5] ACM Digital Library. http://dl.acm.org/.

[6] J. R. Acosta and D. R. Avresky. Dynamic network reconfiguration in presence of multiple node and link failures using autonomous agents. In *1st International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2005.

[7] J. R. Acosta and D. R. Avresky. Intelligent dynamic network reconfiguration. In *21st International Parallel and Distributed Processing Symposium*, 2007.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, pages 63–74, 2008.

[9] Amazon.com, Inc. http://www.amazon.com/.

[10] AMD Virtualization. http://www.amd.com/us/products/technologies/virtualization/.

[11] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference (spring)*, pages 483–485, 1967.

[12] T. E. Anderson, D. E. Culler, and D. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[13] K. V. Anjan and T. M. Pinkston. DISHA: A deadlock recovery scheme for fully adaptive routing. In *9th International Parallel Processing Symposium*, pages 537–543, 1995.

[14] Y. Aridor, T. Domany, O. Goldshmidt, Y. Kliteynik, E. Shmueli, and J. E. Moreira. Multi-toroidal interconnects for tightly coupled supercomputers. *IEEE Transactions on Parallel and Distributed Systems*, 19(1):52–65, January 2008.

[15] Y. Aridor, T. Domany, O. Goldshmidt, J. E. Moreira, and E. Shmueli. Resource allocation and utilization in the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):425–436, March/May 2005.

[16] M. Armbrust, A. Fox, R. Gri th, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, University of California at Berkeley, 2009.

[17] ASI-SIG. Advanced Switching core architecture specification. http://www.picmg.org/v2internal/ASISpecifications.htm, 2004.

[18] S. Attari and A. Isazadeh. Processor allocation in mesh multiprocessors using a hybrid method. In *7th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 492–496, 2006.

[19] D. Avresky and N. Natchev. Dynamic reconfiguration in computer clusters with irregular topologies in the presence of multiple node and link failures. *IEEE Transactions on Computers*, 54(5):603–615, May 2005.

[20] S. M. Balle and D. J. Palermo. Enhancing an open source resource manager with multi-core/multi-threaded support. In *13th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 37–50, 2007.

[21] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[22] S. Bani Mohammad. *E cient processor allocation strategies for mesh-connected multicomputers*. PhD thesis, University of Glasgow, 2008.

[23] M. A. Bender, D. P. Bunde, E. D. Demaine, S. P. Fekete, V. J. Leung, H. Meijer, and C. A. Phillips. Communication-aware processor allocation for supercomputers: Finding point sets of small average distance. *Algorithmica*, 50(2):279–298, 2008.

[24] V. E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Tra c*. Academic Press, 1965.

[25] A. Bermúdez, R. Casado, F. J. Quiles, and J. Duato. Handling topology changes in InfiniBand. *IEEE Transactions on Parallel and Distributed Systems*, 18(2):172–185, February 2007.

[26] G. Bhanot, A. Gara, P. Heidelberger, E. Lawless, J. C. Sexton, and R. Walkup. Optimizing task layout on the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):489–500, March/May 2005.

[27] A. Bhatele and L. V. Kale. Application-specific topology-aware mapping for three dimensional topologies. In *22nd International Parallel and Distributed Processing Symposium*, 2008.

[28] G. K. Bhattacharyya and R. A. Johnson. *Statistical Concepts and Methods*. John Wiley & Sons, Inc., 1977.

[29] The BlueGene/L Team (N. R. Adiga et al.). An overview of the BlueGene/L supercomputer. In *ACM/IEEE Conference on Supercomputing*, pages 1–22, 2002.

[30] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[31] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[32] S. H Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30(3):207–214, March 1981.

[33] R. V. Boppana and S. Chalasani. Fault-tolerant wormhole routing algorithms for mesh networks. *IEEE Transactions on Computers*, 44(7):848–864, July 1995.

[34] D. P. Bunde, V. J. Leung, and J. Mache. Communication patterns and allocation strategies. In *18th International Parallel and Distributed Processing Symposium*, pages 248–255, 2004.

[35] C. Carrión, R. Beivide, J. A. Gregorio, and F. Vallejo. A flow control mechanism to avoid message deadlock in *k*-ary *n*-cube networks. In *4th International Conference on High Performance Computing*, pages 322–329, 1997.

[36] R. Casado, A. Bermúdez, J. Duato, F. J. Quiles, and J. L. Sánchez. A protocol for deadlock-free dynamic reconfiguration in high-speed local area networks. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):115–132, February 2001.

[37] S. Chalasani and R. V. Boppana. Communication in multicomputers with nonconvex faults. *IEEE Transactions on Computers*, 46(5):616–622, May 1997.

[38] C. Chang and P. Mohapatra. Performance improvement of allocation schemes for mesh-connected computers. *Journal of Parallel and Distributed Computing*, 52:40–68, 1998.

[39] C.-L. Chen and G.-M. Chiu. A fault-tolerant routing scheme for meshes with nonconvex faults. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):467–475, May 2001.

[40] H.-L. Chen and S.-H. Hu. Submesh determination in faulty tori and meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(3):272–282, March 2001.

[41] L. Cherkasova, V. Kotov, and T. Rokicki. Fibre Channel fabrics: Evaluation and design. In *29th Hawaii International Conference on System Sciences*, volume 1, pages 53–62, 1996.

[42] A. A. Chien and J. H. Kim. Planar-adaptive routing: Low-cost adaptive networks for multiprocessors. In *19th International Symposium on Computer Architecture*, pages 268–277, 1992.

[43] G.-M. Chiu and S.-K. Chen. An e cient submesh allocation scheme for two-dimensional meshes with little overhead. *IEEE Transactions on Parallel and Distributed Systems*, 10(5):471–486, May 1999.

[44] H. Choo, S.-M. Yoo, and H. Y. Youn. Processor scheduling and allocation for 3D torus multicomputer systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(5):475–484, May 2000.

[45] P.-J. Chuang and N.-F. Tzeng. An e cient submesh allocation strategy for mesh computer systems. In *11th International Conference on Distributed Computing Systems*, pages 256–263, 1991.

[46] P.-J. Chuang and C.-M. Wu. An e cient recognition-complete processor allocation strategy for *k*-ary *n*-cube multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 11(5):485–490, May 2000.

[47] CiteSeer^x. http://citeseerx.ist.psu.edu/.

[48] C. Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, 32:406–424, March 1953.

[49] The Computational Plant project. http://www.cs.sandia.gov/cplant/.

[50] Condor. http://research.cs.wisc.edu/condor/.

[51] Cray XT3 Datasheet. http://www.craysupercomputers.com/downloads/CrayXT3/CrayXT3_Datasheet.pdf.

[52] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[53] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):466–475, April 1993.

[54] W. J. Dally and C. L. Seitz. The torus routing chip. *Distributed Computing*, 1:187–196, 1986.

[55] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36(5):547–553, May 1987.

[56] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Design Automation Conference*, pages 684–689, 2001.

[57] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.

[58] D. Das Sharma and D. K. Pradhan. A fast and e cient strategy for submesh allocation in mesh-connected parallel computers. In *IEEE Symposium on Parallel and Distributed Processing*, pages 682–689, 1993.

[59] N. Desai, D. Buntinas, D. Buettner, P. Balaji, and A. Chan. Improving resource availability by relaxing network allocation constraints on Blue Gene/P. In *38th International Conference on Parallel Processing*, pages 333–339, 2009.

[60] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[61] J. Ding and L. N. Bhuyan. An adaptive submesh allocation strategy for two-dimensional mesh connected systems. In *22nd International Conference on Parallel Processing*, pages II:193–200, 1993.

[62] J. Domke, T. Hoefler, and W. E. Nagel. Deadlock-free oblivious routing for arbitrary topologies. In *25th International Parallel and Distributed Processing Symposium*, pages 616–627, 2011.

[63] J. Duato. On the design of deadlock-free adaptive routing algorithms for multicomputers: Design methodologies. In *Parallel Architectures and Languages Europe*, 1991.

[64] J. Duato. A necessary and su cient condition for deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1055–1067, October 1995.

[65] J. Duato. A necessary and su cient condition for deadlock-free routing in cut-through and store-and-forward networks. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):841–854, August 1996.

[66] J. Duato, I. Johnson, J. Flich, F. Naven, P. Garcia, and T. Nachiondo. A new scalable and cost-e ective congestion management strategy for lossless multistage interconnection networks. In *11th International Symposium on High-Performance Computer Architecture*, pages 108–119, 2005.

[67] J. Duato, O. Lysne, R. Pang, and T. M. Pinkston. Part I: A theory for deadlock-free dynamic network reconfiguration. *IEEE Transactions on Parallel and Distributed Systems*, 16(5):412–427, May 2005.

[68] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers, 2003.

[69] D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling - A status report. In *Job Scheduling Strategies for Parallel Processing*, pages 1–16, 2005.

[70] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 1–34, 1997.

[71] J. Flich, P. López, J. C. Sancho, A. Robles, and J. Duato. Improving InfiniBand routing through multiple virtual networks. In *4th International Symposium on High Performance Computing*, pages 49–63, 2002.

[72] J. Flich, M. P. Malumbres, P. López, and J. Duato. Performance evaluation of a new routing strategy for irregular networks with source routing. In *14th International Conference on Supercomputing*, pages 34–43, 2000.

[73] J. Flich, S. Rodrigo, and J. Duato. An e cient implementation of distributed routing algorithms for NoCs. In *2nd ACM/IEEE International Symposium on Networks-on-Chip*, pages 87–96, 2008.

[74] J. Flich, S. Rodrigo, J. Duato, T. Sødring, Å. G. Solheim, T. Skeie, and O. Lysne. On the potential of NoC virtualization for multicore chips. *Scalable Computing: Practice and Experience*, 9(3):165–177, 2008.

[75] J. Flich, T. Skeie, A. Mejia, O. Lysne, P. López, A. Robles, J. Duato, M. Koibuchi, T. Rokicki, and J. C. Sancho. A survey and evaluation of topology agnostic deterministic routing algorithms. To appear in *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[76] P. Francois and O. Bonaventure. Avoiding transient loops during IGP convergence in IP networks. In *24th IEEE INFOCOM*, volume 1, pages 237–247, 2005.

[77] B. Fu, Y. Han, J. Ma, H. Li, and X. Li. An abacus turn model for time/space-e cient reconfigurable routing. In *38th International Symposium on Computer Architecture*, pages 259–270, 2011.

[78] P. J. Garcia, F. J. Quiles, J. Flich, J. Duato, I. Johnson, and F. Naven. E cient, scalable congestion management for interconnection networks. *IEEE Micro*, 26(5):52–66, September 2006.

[79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[80] C. J. Glass and L. M. Ni. The Turn Model for adaptive routing. In *19th International Symposium on Computer Architecture*, pages 278–287, 1992.

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

[82] Gnuplot. http://www.gnuplot.info/.

[83] L. K. Goh and B. Veeravalli. Design and performance evaluation of combined first-fit task allocation and migration strategies in mesh multiprocessor systems. *Parallel Computing*, 34(9):508–520, 2008.

[84] M. E. Gómez, N. A. Nordbotten, J. Flich, P. López, A. Robles, J. Duato, T. Skeie, and O. Lysne. A routing methodology for achieving fault tolerance in direct networks. *IEEE Transactions on Computers*, 55(4):400–415, April 2006.

[85] Google Inc. http://www.google.com/.

[86] Google Scholar. http://scholar.google.com/.

[87] E. G. Gran, M. Eimot, S.-A. Reinemo, T. Skeie, O. Lysne, L. P. Huse, and G. Shainer. First experiences with congestion control in InfiniBand hardware. In *24th International Parallel and Distributed Processing Symposium*, 2010.

[88] The Green500 List. http://www.green500.org/.

[89] W. L. Guay and S.-A. Reinemo. A scalable method for signalling dynamic reconfiguration events with OpenSM. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 332–341, 2011.

[90] W. L. Guay, S.-A. Reinemo, O. Lysne, T. Skeie, B. D. Johnsen, and L. Holen. Host side dynamic reconfiguration with InfiniBand. In *IEEE International Conference on Cluster Computing*, pages 126–135, 2010.

[91] V. Gupta and A. Jayendran. A flexible processor allocation strategy for mesh connected parallel systems. In *25th International Conference on Parallel Processing*, pages III:166–173, 1996.

[92] M. Gusat, D. Craddock, W. Denzel, T. Engbersen, N. Ni, G. Pfister, W. Rooney, and J. Duato. Congestion control in InfiniBand networks. In *13th Symposium on High Performance Interconnects*, pages 158–159, 2005.

[93] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and E. Bu. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[94] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950.

[95] C.-T. Ho and L. Stockmeyer. A new approach to fault-tolerant wormhole routing for mesh-connected parallel computers. *IEEE Transactions on Computers*, 53(4):427–438, April 2004.

[96] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[97] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage switches are not crossbars: E ects of static routing in high-performance networks. In *IEEE International Conference on Cluster Computing*, pages 116–125, 2008.

[98] T. Hoefler, T. Schneider, and A. Lumsdaine. The e ect of network noise on large-scale collective communications. *Parallel Processing Letters*, 19(4):573–593, 2009.

[99] T. Hoefler, T. Schneider, and A. Lumsdaine. Optimized routing for large-scale InfiniBand networks. In *17th Symposium on High-Performance Interconnects*, pages 103–111, 2009.

[100] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51–61, September 2007.

[101] IBM Blue Gene team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2):199–220, January/March 2008.

[102] IEEE Standards Association. IEEE 802.3-2008. http://standards.ieee.org/about/get/802/802.3.html.

[103] IEEE Xplore. http://ieeexplore.ieee.org/Xplore/.

[104] IET Inspec. http://www.theiet.org/resources/inspec/.

[105] InfiniBand Trade Association. *InfiniBand Architecture Specification v. 1.2.1*. http://www.infinibandta.org/specs/, 2007.

[106] Intel Corporation. http://www.intel.com/.

[107] Intel Virtualization Technology. http://www.intel.com/technology/virtualization/technology.htm.

[108] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Inc., 1991.

[109] Java Platform, Standard Edition 6, API Specification. http://docs.oracle.com/javase/6/docs/api/.

[110] C. R. Johnson, D. P. Bunde, and V. J. Leung. A tie-breaking strategy for processor allocation in meshes. In *39th International Conference on Parallel Processing Workshops*, pages 331–338, 2010.

[111] A. Jouraku, M. Koibuchi, and H. Amano. An e ective design of deadlock-free routing algorithms based on 2D turn model for irregular networks. *IEEE Transactions on Parallel and Distributed Systems*, 18(3):320–333, March 2007.

[112] A. Jouraku, M. Koibuchi, H. Amano, and A. Funahashi. Routing algorithms based on 2D turn model for irregular networks. In *International Symposium on Parallel Architectures, Algorithms and Networks*, pages 254–259, 2002.

[113] M. Kang, C. Yu, H. Y. Youn, B. Lee, and M. Kim. Isomorphic strategy for processor allocation in k-ary n-cube systems. *IEEE Transactions on Computers*, 52(5):645–657, May 2003.

[114] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[115] G. Kim and H. Yoon. On submesh allocation for mesh multicomputers: A best-fit allocation and a virtual submesh allocation for faulty meshes. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):175–185, February 1998.

[116] J. H. Kim, Z. Liu, and A. A. Chien. Compressionless routing: A framework for adaptive and fault-tolerant routing. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):229–244, March 1997.

[117] M. Koibuchi, A. Funahashi, A. Jouraku, and H. Amano. L-turn routing: An adaptive routing in irregular networks. In *30th International Conference on Parallel Processing*, pages 383–392, 2001.

[118] M. Koibuchi, A. Jouraku, and H. Amano. The impact of path selection algorithm of adaptive routing for implementing deterministic routing. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1431–1437, 2002.

[119] M. Koibuchi, A. Jouraku, K. Watanabe, and H. Amano. Descending layers routing: A deadlock-free deterministic routing using virtual channels in system area networks with irregular topologies. In *32nd International Conference on Parallel Processing*, pages 527–536, 2003.

[120] V. Krishnan. Towards an integrated IO and clustering solution using PCI Express. In *IEEE International Conference on Cluster Computing*, pages 259–266, 2007.

[121] V. Krishnan and D. Mayhew. A localized congestion control mechanism for PCI Express Advanced Switching fabrics. In *12th IEEE Hot Interconnects Symposium*, 2004.

[122] V. Krishnan, T. Miller, and H. Paraison. Dolphin Express: A transparent approach to enhancing PCI Express. In *IEEE International Conference on Cluster Computing*, pages 464–467, 2007.

[123] S. O. Krumke, M. V. Marathe, H. Noltemeier, V. Radhakrishnan, S. S. Ravi, and D. J. Rosenkrantz. Compact location problems. *Theoretical Computer Science*, 181(2):379–404, 1997.

[124] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä, and A. Hemani. A network on chip architecture and design methodology. In *IEEE Computer Society Annual Symposium on VLSI*, 2002.

[125] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[126] C. E. Leiserson. Fat-trees: Universal networks for hardware-e cient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, October 1985.

[127] V. J. Leung, E. M. Arkin, M. A. Bender, D. Bunde, J. Johnston, A. Lal, J. S. B. Mitchell, C. Phillips, and S. S. Seiden. Processor allocation on Cplant: Achieving general processor locality using one-dimensional allocation strategies. In *4th IEEE International Conference on Cluster Computing*, pages 296–304, 2002.

[128] V. J. Leung, C. A. Phillips, M. A. Bender, and D. P. Bunde. Algorithmic support for commodity-based parallel computing systems. Technical Report SAND2003-3702, Sandia National Laboratories, 2003.

[129] L. Levitin, M. Karpovsky, and M. Mustafa. Deadlock prevention by turn prohibition in interconnection networks. In *9th Workshop on Communication Architecture for Clusters*, 2009.

[130] K. Li and K. Cheng. A two-dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system. *Journal of Parallel and Distributed Computing*, 12(1):79–83, May 1991.

[131] X. Liao, W. Jigang, and T. Srikanthan. Brief announcement: A temperature-aware virtual submesh allocation scheme for NoC-based manycore chips. In *20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 182–184, 2008.

[132] J. A. Libak. Routing and job allocation in high performance clusters. Master's thesis, University of Oslo, 2008.

[133] D. A. Lifka. The ANL/IBM SP scheduling system. In *1st Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, 1995.

[134] D. H. Linder and J. C. Harden. An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes. *IEEE Transactions on Computers*, 40(1):2–12, January 1991.

[135] Y. Liu, C. Dwyer, and A. R. Lebeck. Routing in self-organizing nano-scale irregular networks. *ACM Journal on Emerging Technologies in Computing Systems*, 6(1):3:1–3:21, March 2008.

[136] V. Lo, J. Mache, and K. Windisch. A comparative study of real workload traces and synthetic workload models for parallel job scheduling. In *4th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 25–46, 1998.

[137] V. Lo, K. Windisch, W. Liu, and B. Nitzberg. Non-contiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):712–726, July 1997.

[138] O. Lysne and J. Duato. Fast dynamic reconfiguration in irregular networks. In *29th International Conference on Parallel Processing*, pages 449–458, 2000.

[139] O. Lysne, J. M. Montañana, J. Flich, J. Duato, T. M. Pinkston, and T. Skeie. An e - cient and deadlock-free network reconfiguration protocol. *IEEE Transactions on Computers*, 57(6):762–779, June 2008.

[140] O. Lysne, J. M. Montañana, T. M. Pinkston, J. Duato, T. Skeie, and J. Flich. Simple deadlock-free dynamic network reconfiguration. In *11th International Conference on High Performance Computing*, pages 504–515, 2004.

[141] O. Lysne, T. M. Pinkston, and J. Duato. Part II: A methodology for developing deadlock-free dynamic network reconfiguration processes. *IEEE Transactions on Parallel and Distributed Systems*, 16(5):428–443, May 2005.

[142] O. Lysne, S.-A. Reinemo, T. Skeie, Å. G. Solheim, T. Sødring, L. P. Huse, and B. D. Johnsen. Interconnection networks: Architectural challenges for utility computing data centers. *IEEE Computer*, 41(9):62–69, September 2008.

[143] O. Lysne and T. Skeie. Load balancing of irregular system area networks through multiple roots. In *2nd International Conference on Communications in Computing*, 2001.

[144] O. Lysne, T. Skeie, S.-A. Reinemo, and I. Theiss. Layered routing in irregular networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(1):51–65, January 2006.

[145] J. Mache and V. Lo. The e ects of dispersal on message-passing contention in processor allocation strategies. In *3rd Joint Conference on Information Sciences, Sessions on Parallel and Distributed Processing*, pages 223–226, 1997.

[146] J. Mache, V. Lo, and K. Windisch. Minimizing message-passing contention in fragmentation-free processor allocation. In *10th International Conference on Parallel and Distributed Computing Systems*, pages 120–124, 1997.

[147] W. Mao, J. Chen, and W. Watson III. E cient subtorus processor allocation in a multi-dimensional torus. In *8th International Conference on High-Performance Computing in Asia-Pacific Region*, 2005.

[148] J. C. Martínez, J. Flich, A. Robles, P. López, and J. Duato. Supporting fully adaptive routing in InfiniBand networks. In *17th International Parallel and Distributed Processing Symposium*, 2003.

[149] R. Martínez, F. J. Alfaro, and J. L. Sánchez. A framework to provide quality of service over Advanced Switching. *IEEE Transactions on Parallel and Distributed Systems*, 19(8):1111–1123, August 2008.

[150] D. Mayhew and V. Krishnan. PCI Express and Advanced Switching: Evolutionary path to building next generation interconnects. In *11th Symposium on High Performance Interconnects*, pages 21–29, 2003.

[151] A. Mejia, J. Flich, J. Duato, S.-A. Reinemo, and T. Skeie. Segment-Based Routing: An e cient fault-tolerant routing algorithm for meshes and tori. In *20th International Parallel and Distributed Processing Symposium*, 2006.

[152] J. M. Montañana, J. Flich, and J. Duato. Epoch-Based Reconfiguration: Fast, simple, and e ective dynamic network reconfiguration. In *22nd International Parallel and Distributed Processing Symposium*, 2008.

[153] J. M. Montañana, J. Flich, A. Robles, and J. Duato. A scalable methodology for computing fault-free paths in InfiniBand torus networks. In *6th International Symposium on High-Performance Computing*, pages 79–92, 2005.

[154] J. M. Montañana, J. Flich, A. Robles, P. López, and J. Duato. A transition-based fault-tolerant routing methodology for InfiniBand networks. In *18th International Parallel and Distributed Processing Symposium*, 2004.

[155] J. M. Montañana, M. Koibuchi, H. Matsutani, and H. Amano. Stabilizing path modification of power-aware on/o interconnection networks. In *5th International Conference on Networking, Architecture, and Storage*, pages 218–227, 2010.

[156] S. Q. Moore and L. M. Ni. The e ects of network contention on processor allocation strategies. In *10th International Parallel Processing Symposium*, pages 268–273, 1996.

[157] R. Moraveji, P. Moinzadeh, H. Sarbazi-Azad, and A. Y. Zomaya. Multispanning tree zone-ordered label-based routing algorithms for irregular networks. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):817–832, May 2011.

[158] MPI Standard Documents. http://www.mpi-forum.org/docs/.

[159] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, June 2001.

[160] T. Nesson and S. L. Johnsson. ROMM routing on mesh and torus networks. In *7th ACM Symposium on Parallel Algorithms and Architectures*, pages 275–287, 1995.

[161] N. A. Nordbotten. *Fault-Tolerant Routing in Interconnection Networks*. PhD thesis, University of Oslo, 2008.

[162] N. A. Nordbotten and T. Skeie. A routing methodology for dynamic fault tolerance in meshes and tori. In *14th International Conference on High Performance Computing*, pages 514–527, 2007.

[163] OpenFabrics Alliance. http://www.openfabrics.org/.

[164] OpenMP Specifications. http://openmp.org/wp/openmp-specifications/.

[165] F. Palumbo, D. Pani, L. Ra o, and S. Secchi. A surface tension and coalescence model for dynamic distributed resources allocation in massively parallel processors on-chip. In *2nd International Workshop on Nature Inspired Cooperative Strategies for Optimization*, 2007.

[166] F. Palumbo, S. Secchi, D. Pani, and L. Ra o. A novel non-exclusive dual-mode architecture for MPSoCs-oriented network on chip designs. In *8th International Workshop on Systems, Architectures, Modeling, and Simulation*, 2008.

[167] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance evaluation and design trade-o s for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, 54(8):1025–1040, August 2005.

[168] J. A. Pascual, J. Navaridas, and J. Miguel-Alonso. E ects of topology-aware allocation policies on scheduling performance. In *Job Scheduling Strategies for Parallel Processing*, pages 138–156, 2009.

[169] J. H. Patel. Performance of processor-memory interconnections for multiprocessors. *IEEE Transactions on Computers*, C-30(10):771–780, October 1981.

[170] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD Record*, 17(3):109–116, June 1988.

[171] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware / Software Interface*. Morgan Kaufmann Publishers, 2009.

[172] K. Pawlikowski. Steady-state simulation of queueing processes: A survey of problems and solutions. *ACM Computing Surveys*, 22(2):123–170, June 1990.

[173] PCI-SIG. PCI Express base specification 1.0a. http://www.pcisig.com/specifications/pciexpress/base/archive/, 2003.

[174] F. De Pellegrini, D. Starobinski, M. G. Karpovsky, and L. B. Levitin. Scalable cycle-breaking algorithms for Gigabit Ethernet backbones. In *23rd IEEE INFOCOM*, volume 4, pages 2175–2184, 2004.

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

[176] F. Petrini and M. Vanneschi. *k*-ary *n*-trees: High performance networks for massively parallel architectures. In *11th International Parallel Processing Symposium*, pages 87–93, April 1997.

[177] G. F. Pfister. An introduction to the InfiniBand architecture. Chapter 42 in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, edited by H. Jin, R. Cortes and R. Buyya, pages 617–632. Wiley-IEEE Press, 2001.

[178] G. F. Pfister and V. A. Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, 34(10):943–948, October 1985.

[179] T. M. Pinkston, R. Pang, and J. Duato. Deadlock-free dynamic reconfiguration schemes for increased network dependability. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):780–794, August 2003.

[180] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[181] W. Qiao and L. M. Ni. E cient processor allocation for 3D tori. In *9th International Parallel Processing Symposium*, pages 466–471, 1995.

[182] W. Qiao and L. M. Ni. Adaptive routing in irregular networks using cut-through switches. In *25th International Conference on Parallel Processing*, volume 1, pages 52–60, 1996.

[183] S.-A. Reinemo and T. Skeie. E ective shortest path routing for Gigabit Ethernet. In *IEEE International Conference on Communications*, pages 6419–6424, 2007.

[184] S.-A. Reinemo, T. Skeie, T. Sødring, O. Lysne, and O. Tørudbakken. An overview of QoS capabilities in InfiniBand, Advanced Switching Interconnect, and Ethernet. *IEEE Communications Magazine*, 44(7):32–38, July 2006.

[185] A. Robles-Gómez, A. Bermúdez, R. Casado, and Å. G. Solheim. Deadlock-free dynamic network reconfiguration based on close Up*/Down* graphs. In *14th International Euro-Par Conference*, pages 940–949, 2008.

[186] A. Robles-Gómez, A. Bermúdez, and R. Casado. Implementing a change assimilation mechanism for source routing interconnects. In *15th International Euro-Par Conference*, pages 1029–1039, 2009.

[187] A. Robles-Gómez, A. Bermúdez, and R. Casado. A deadlock-free dynamic reconfiguration scheme for source routing networks using close Up*/Down* graphs. *IEEE Transactions on Parallel and Distributed Systems*, 22(10):1641–1652, October 2011.

[188] A. Robles-Gómez, A. Bermúdez, R. Casado, F. J. Quiles, T. Skeie, and J. Duato. A proposal for managing ASI fabrics. *Journal of Systems Architecture*, 54(7):664–678, July 2008.

[189] A. Robles-Gómez, A. Bermúdez, R. Casado, Å. G. Solheim, T. Sødring, and T. Skeie. A new distributed management mechanism for ASI based networks. *Computer Communications*, 32(2):294–304, February 2009.

[190] T. L. Rodehe er and M. D. Schroeder. Automatic reconfiguration in Autonet. In *13th ACM Symposium on Operating Systems Principles*, pages 183–197, 1991.

[191] S. Rodrigo, J. Flich, J. Duato, and M. Hummel. E cient unicast and multicast support for CMPs. In *41st IEEE/ACM International Symposium on Microarchitecture*, pages 364–375, 2008.

[192] S. Rodrigo, J. Flich, A. Roca, S. Medardoni, D. Bertozzi, J. Camacho, F. Silla, and J. Duato. Addressing manufacturing challenges with cost-e cient fault tolerant routing. In *4th ACM/IEEE International Symposium on Networks-on-Chip*, pages 25–32, 2010.

[193] S. Rodrigo Mocholí. *Cost E ective Routing Implementations for On-chip Networks*. PhD thesis, Polytechnic University of Valencia, 2010.

[194] C. A. F. De Rose, H.-U. Heiss, and B. Linnert. Distributed dynamic processor allocation for multicomputers. *Parallel Computing*, 33(3):145–158, April 2007.

[195] C. A. F. De Rose, H.-U. Heiss, and P. A. O. Navaux. Distributed processor allocation in large PC clusters. *9th IEEE International Symposium on High-Performance Distributed Computing*, pages 288–289, 2000.

[196] J. C. Sancho and A. Robles. Improving the Up*/Down* routing scheme for networks of workstations. In *6th International Euro-Par Conference*, pages 882–889, 2000.

[197] J. C. Sancho, A. Robles, and J. Duato. A flexible routing scheme for networks of workstations. In *3rd International Symposium on High Performance Computing*, pages 260–267, 2000.

[198] J. C. Sancho, A. Robles, and J. Duato. A new methodology to compute deadlock-free routing tables for irregular networks. In *4th International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications*, pages 45–60, 2000.

[199] J. C. Sancho, A. Robles, and J. Duato. An e ective methodology to improve the performance of the Up*/Down* routing algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):740–754, August 2004.

[200] J. C. Sancho, A. Robles, J. Flich, P. López, and J. Duato. E ective methodology for deadlock-free minimal routing in InfiniBand networks. In *31st International Conference on Parallel Processing*, pages 409–418, 2002.

[201] Sandia National Laboratories. http://www.sandia.gov/.

[202] Scalable Intelligent Video Server System (SIVSS). http://www.sivss.org/.

[203] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[204] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodehe er, E. H. Satterthwaite, and C. P. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. SRC Research Report 59, Digital Equipment Corporation, 1990.

[205] R. L. Schwartz. *Learning Perl*. O'Reilly & Associates, Inc., 1993.

[206] R. Seifert. *Gigabit Ethernet: Technology and Applications for High-Speed LANs*. Addison Wesley Longman, Inc., 1998.

[207] R. Seifert. *The Switch Book: The Complete Guide to LAN Switching Technology*. John Wiley & Sons, Inc., 2000.

[208] F. O. Sem-Jacobsen and O. Lysne. Fault tolerance with shortest paths in regular and irregular networks. In *22nd International Parallel and Distributed Processing Symposium*, 2008.

[209] F. O. Sem-Jacobsen, S. Rodrigo, T. Skeie, A. Strano, and D. Bertozzi. An e cient, low-cost routing framework for convex mesh partitions to support virtualisation. To appear in *ACM Transactions on Embedded Computing Systems*, 2012.

[210] F. O. Sem-Jacobsen, Å. G. Solheim, O. Lysne, T. Skeie, and T. Sødring. E cient and contention-free virtualisation of fat-trees. In *Communication Architecture for Scalable Systems*, pages 754–760, 2011.

[211] K.-H. Seo. Fragmentation-e cient node allocation algorithm in 2D mesh-connected systems. In *8th International Symposium on Parallel Architectures, Algorithms and Networks*, 2005.

[212] F. Silla and J. Duato. Improving the e ciency of adaptive routing in networks with irregular topology. In *4th International Conference on High Performance Computing*, pages 330–335, 1997.

[213] F. Silla and J. Duato. High-performance routing in networks of workstations with irregular topology. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):699–719, July 2000.

[214] A. Singh, W. J. Dally, A. K. Gupta, and B. Towles. GOAL: A load-balanced adaptive routing algorithm for torus networks. *ACM SIGARCH Computer Architecture News*, 31(2):194–205, May 2003.

[215] A. Singh, W. J. Dally, B. Towles, and A. K. Gupta. Locality-preserving randomized oblivious routing on torus networks. In *14th ACM Symposium on Parallel Algorithms and Architectures*, pages 9–19, 2002.

[216] T. Skeie. Handling multiple fauls in wormhole mesh networks. In *4th International Euro-Par Conference*, pages 1076–1088, 1998.

[217] T. Skeie, O. Lysne, J. Flich, P. López, A. Robles, and J. Duato. LASH-TOR: A generic transition-oriented routing algorithm. In *10th International Conference on Parallel and Distributed Systems*, pages 595–604, 2004.

[218] T. Skeie, O. Lysne, and I. Theiss. Layered shortest path (LASH) routing in irregular system area networks. In *2nd Workshop on Communication Architecture for Clusters*, 2002.

[219] T. Skeie, F. O. Sem-Jacobsen, S. Rodrigo, J. Flich, D. Bertozzi, and S. Medardoni. Flexible DOR routing for virtualization of multicore chips. In *International Symposium on System-on-Chip*, pages 73–76, 2009.

[220] J. Skovira, W. Chan, H. Zhou, and D. A. Lifka. The EASY - LoadLeveler API project. In *2nd Workshop on Job Scheduling Strategies for Parallel Processing*, pages 41–47, 1996.

[221] Å. G. Solheim, O. Lysne, A. Bermúdez, R. Casado, T. Sødring, T. Skeie, and A. Robles-Gómez. E cient and deadlock-free reconfiguration for source routed networks. In *9th Workshop on Communication Architecture for Clusters*, 2009.

[222] Å. G. Solheim, O. Lysne, and T. Skeie. RecTOR: A new and e cient method for dynamic network reconfiguration. In *15th International Euro-Par Conference*, pages 1052–1064, 2009.

[223] Å. G. Solheim, O. Lysne, T. Skeie, T. Sødring, and S.-A. Reinemo. A framework for routing and resource allocation in network virtualization. In *16th International Conference on High Performance Computing*, pages 129–139, 2009.

[224] Å. G. Solheim, O. Lysne, T. Skeie, T. Sødring, I. Theiss, and I. Johnson. Routing for the ASI Fabric Manager. *IEEE Communications Magazine*, 44(7):39–44, July 2006.

[225] Å. G. Solheim, O. Lysne, T. Sødring, T. Skeie, and J. A. Libak. Routing-contained virtualization based on Up*/Down* forwarding. In *14th International Conference on High Performance Computing*, pages 500–513, 2007.

[226] SpringerLink. http://www.springerlink.com/.

[227] P. J. S. Srikanth, K. Praveen, L. Harish Subramaniam, and T. Srinivasan. Heuristic processor allocation in mesh-connected systems via a coloring mechanism. In *International Conference on Computing & Informatics*, 2006.

[228] T. Srinivasan, J. Seshadri, A. Chandrasekhar, and J. B. Siddharth Jonathan. A minimal fragmentation algorithm for task allocation in mesh-connected multicomputers. In *International Conference on Advances in Intelligent Systems: Theory and Applications*, 2004.

[229] J. Srisawat and N. A. Alexandridis. A new "quad-tree-based" sub-system allocation technique for mesh-connected parallel machines. In *13th International Conference on Supercomputing*, pages 60–67, 1999.

[230] V. Subramani, R. Kettimuthu, S. Srinivasan, J. Johnston, and P. Sadayappan. Selective buddy allocation for scheduling parallel jobs on clusters. In *4th IEEE International Conference on Cluster Computing*, pages 107–116, 2002.

[231] H. Sullivan and T. R. Bashkow. A large scale, homogeneous, fully distributed parallel machine, I. *ACM SIGARCH Computer Architecture News*, 5(7):105–117, March 1977.

[232] W. Tang, Z. Lan, N. Desai, D. Buettner, and Y. Yu. Reducing fragmentation on torus-connected supercomputers. In *25th International Parallel and Distributed Processing Symposium*, pages 828–839, 2011.

[233] D. Teodosiu, J. Baxter, K. Govil, J. Chapin, M. Rosenblum, and M. Horowitz. Hardware fault containment in scalable shared-memory multiprocessors. *ACM SIGARCH Computer Architecture News*, 25(2):73–84, May 1997.

[234] I. Theiss. *Modularity, Routing and Fault Tolerance in Interconnection Networks*. PhD thesis, University of Oslo, 2004.

[235] I. Theiss and O. Lysne. LORE - Local reconfiguration for fault management in irregular interconnects. In *18th International Parallel and Distributed Processing Symposium*, 2004.

[236] I. Theiss and O. Lysne. FRoots: A fault tolerant and topology-flexible routing technique. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1136–1150, October 2006.

[237] Top500 Supercomputer Sites. http://www.top500.org/, June 2011.

[238] F. Triviño, F. J. Alfaro, J. L. Sánchez, and J. Flich. NoC reconfiguration for CMP virtualization. In *10th IEEE International Symposium on Network Computing and Applications*, pages 219–222, 2011.

[239] F. Triviño, J. L. Sánchez, F. J. Alfaro, and J. Flich. Virtualizing network-on-chip resources in chip-multiprocessors. *Microprocessors and Microsystems*, 35(2):230–245, 2011.

[240] H.-Y. Tyan. *Design, Realization and Evaluation of a Component-Based Compositional Software Architecture for Network Simulation*. PhD thesis, Ohio State University, 2002.

[241] University of Oslo. http://www.uio.no/.

[242] University of Oslo Library – Science Library, Informatics. http://www.ub.uio.no/english/about/organisation/ureal/inf/.

[243] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, 1982.

[244] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-W teraflops processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, January 2008.

[245] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: Towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, January 2009.

[246] C. Vizino. Enabling contiguous node scheduling on the Cray XT3. In *Cray User Group*, 2008.

[247] P. Walker, D. P. Bunde, and V. J. Leung. Faster high-quality processor allocation. In *11th LCI International Conference on High-Performance Clustered Computing*, 2010.

[248] M. A. Weiss. *Data Structures and Algorithm Analysis*. The Benjamin/Cummings Publishing Company, Inc., 1995.

[249] D. Weisser, N. Nystrom, C. Vizino, S. T. Brown, and J. Urbanic. Optimizing job placement on the Cray XT3. In *Cray User Group*, 2006.

[250] Wikipedia. http://en.wikipedia.org/wiki/Amdahl's_law.

[251] K. Windisch, V. Lo, and B. Bose. Contiguous and non-contiguous processor allocation algorithms for *k*-ary *n*-cubes. In *24th International Conference on Parallel Processing*, pages II:164–168, 1995.

[252] K. Windisch, J. V. Miller, and V. Lo. ProcSimity: An experimental tool for processor allocation and scheduling in highly parallel systems. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 414–421, 1995.

[253] F. Wu, C.-C. Hsu, and L.-P. Chou. Processor allocation in the mesh multiprocessors using the leapfrog method. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):276–289, March 2003.

[254] Xyratex Technology Limited. http://www.xyratex.com/.

[255] B. S. Yoo and C. R. Das. A fast and e cient processor allocation scheme for mesh-connected multicomputers. *IEEE Transactions on Computers*, 51(1):46–60, January 2002.

[256] S.-M. Yoo, H. Y. Youn, and B. Shirazi. An e cient task allocation scheme for 2D mesh architectures. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):934–942, September 1997.

[257] B. Zafar, T. M. Pinkston, A. Bermúdez, and J. Duato. Deadlock-free dynamic reconfiguration over InfiniBand networks. *International Journal of Parallel, Emergent and Distributed Systems*, 19(2):127–143, June 2004.

[258] G. Zarza, D. Lugones, D. Franco, and E. Luque. A multipath fault-tolerant routing method for high-speed interconnection networks. In *15th International Euro-Par Conference*, pages 1078–1088, 2009.

[259] L. Zhang. Comments on "A fast and e cient processor allocation scheme for mesh-connected multicomputers". *IEEE Transactions on Computers*, 52(2):255–256, February 2003.

[260] J. Zhou and Y.-C. Chung. Tree-turn routing: An e cient deadlock-free routing algorithm for irregular networks. *The Journal of Supercomputing*, 2010.

[261] X. Zhu and W.-M. Lin. Allocation time-based processor allocation scheme for 2D mesh architecture. In *International Conference on Parallel and Distributed Systems*, pages 447–451, 1998.

[262] Y. Zhu. E cient processor allocation strategies for mesh-connected parallel computers. *Journal of Parallel and Distributed Computing*, 16(4):328–337, 1992.

[263] D. Zydek and H. Selvaraj. Hardware implementation of processor allocation schemes for mesh-based chip multiprocessors. *Microprocessors and Microsystems*, 34(1):39–48, February 2010.

[264] D. Zydek and H. Selvaraj. Fast and e cient processor allocation algorithm for torus-based chip multiprocessors. *Computers and Electrical Engineering*, 37(1):91–105, January 2011.

[265] D. Zydek, H. Selvaraj, G. Borowik, and T. Luba. Energy characteristic of a processor allocator and a network-on-chip. *International Journal of Applied Mathematics and Computer Science*, 21(2):385–399, 2011.

All online references were accessed December 12, 2011.