Dynamic Reconfiguration in Interconnection Networks

Wei Lin Guay

Thesis submitted for the degree of Philosophiae Doctor
Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo
2014

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1450

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the thesis defence. Kindly direct all inquiries regarding the thesis to the copyright holder or the unit which grants the doctorate.

Abstract

Interconnection networks are widely deployed as the communication fabric in different systems, ranging from internal networks in clusters to the wide area networks in public clouds. In the event of rerouting due to failure of networking components, congestion control, and migration of jobs, the underlying network characteristic of a system must be reconfigured. Due to the fact that the large networks in clusters and enterprise environments are becoming too complex to manage on an individual system-by-system basis, self-managing and self-configuring networks are a requirement in future systems. In this thesis, we study dynamic reconfiguration and our contributions are in three different contexts.

The first contribution is a fault tolerant mechanism that can dynamically reconfigure the characteristic of the interconnection networks when a fault happens. The main methodology is based on event forwarding in the network manager to notify the affected nodes with the updated path information. Using this method, network traffic is not required to be stopped during the reconfiguration, enabling a fault tolerant mechanism that is transparent to the applications. In addition, an extension is provided to reduce the management message overheads during the event forwarding mechanism. The second contribution is a congestion control mechanism that can reconfigure the route dynamically to alleviate the negative effects of head-of-line blocking. This methodology provides a framework, with only using virtual lanes, that can improve the network performance in the presence of hot-spots. The network manager monitors the network periodically, detects hot-spots, and reconfigures the network traffic autonomously by isolating the congested flows from the normal traffic. The third contribution is a methodology enabling dynamic reconfiguration of a high-speed networking device, that is attached to a virtual machine (VM), during live migration. In this method, the reconfiguration of the underlying hardware managed resources of the high-speed networking device is performed dynamically when a VM is migrated. The on-going network operations of the high-speed networking device can be resumed after the live migration without any manual configuration.

Acknowledgements

First and foremost, I wish to express my sincere gratitude to my primary supervisor Dr. Sven-Arne Reinemo for his patient guidance, encouragement and advice he has provided throughout the work of this thesis. I would also like to thank to my secondary supervisors Prof. Tor Skeie and Prof. Olav Lysne for their valuable support and guidance, and for giving me this opportunity to pursue my Ph.D.

Furthermore, I would like to thank Bjørn Dag Johnsen, Line Holen, Dr. Lars Paul Huse, Chien-Hua Yen, Kurt Tjemsland, Jørn Raastad and Ola Tørudbakken at Oracle Corporation for supporting my Ph.D technically and financially. I also wish to thank fellow colleagues, Ernst Gunnar Gran and Bartosz Bogdanski for good collaboration and Dr. Thomas Dreibholz for reviewing this thesis.

A special thanks to all my family. I would like to thank my beloved wife Mindy Lim, my parents Guay Hock Kim and Ng Lian Eng for their understanding, love, care and support. The sacrifices that they made throughout the period of my Ph.D studies are simply ineffable. Finally, I thank God for letting me through all the difficulties, and giving me strength all the way until this stage to finish my Ph.D studies.

Contents

Chapter 1

Introduction

High Performance Computing (HPC) systems, such as supercomputers and data centres, are no longer exclusive to the scientific community. Today, these systems are also demanded by different sectors in the enterprise community such as social network providers, financial services, retail or manufacturing [86]. All of these organizations have a common problem; they have collected a large amount of data which need to be analyzed, e.g. Big Data [70]. However, these datasets are too large and complex to be processed in a reasonable amount of time by a relational database [8]. Thus, HPC is being used, together with a software framework that supports data–intensive applications such as Hadoop [2], as the mainstream solution for data analytics in enterprise systems.

In the HPC system, the interconnection network, which is the communication infrastructure that interlinks compute nodes and transports data among them, is one of the important components that contributes to the overall performance. An interconnection network can be characterized by its topology, routing algorithms and switching techniques, and each of these characteristics is application–centric [36]. Therefore, in this thesis we study methodologies for dynamic reconfiguration of the interconnection network. In general, a *reconfigurable* architecture is an architecture that can alter the functionality and structure of its components [111]. If the reconfiguration is performed without any manual effort, and with minimal or no overhead, it is said to be *dynamic*. As a result, the term *dynamic reconfiguration* is defined as the ability to make changes to its subsystem, without impacting the running applications, while the system is running [73]. In this thesis, *dynamic reconfiguration* refers to mechanisms that can dynamically *reconfigure* the network charateristics of the interconnection network after identifying the occurrence of an event, such as a fault, a traffic pattern change or a job migration event.

1.1 Motivation

Ideally, an interconnection network should have the capability to monitor, manage and reconfigure the network charateristics and resources without human intervention after the occurance of an event such as topology change, network congestion or job migration. The reconfiguration should also be application transparent and only impose minimum service downtime to the running applications. However, this is not the case in reality because the reconfiguration is usually being performed statically [18]. An external component is used to detect the events in the interconnection network. Subsequently, the running applications are halted, then a new set of network resources is applied manually before the running applications are restarted again. Thus, this motivates us to study dynamic reconfiguration of the interconnection network. In our opinion, there are two reasons why dynamic reconfiguration of the interconnection networks is challenging. First, there is no ideal and general *configuration* for the interconnection network that is applicable to all traffic patterns. In this thesis, the term *configuration* refers to the network topology, the routing algorithm, the switching technique and the device resources in the interconnection network. Second, *system virtualization* has been introduced to improve the flexibility, utilization, and productivty of the HPC systems. Virtualization not only adds new features, but comes at a cost creating additional complexity in managing the configuration of the interconnection network.

The configuration of an interconnection network

The configuration in a large-scale HPC system needs to be changed frequently to maintain the connectivity and performance due to the traffic pattern changes, occurrence faults or job migration. Today, the checkpoint-restart mechanism is a widely used solution to perform reconfiguration [38]. The checkpoint-restart mechanism works in such a way that when there is a need to modify the configuration, the application can be halted, a new configuration can be implemented while the network is empty of traffic, and finally the application can be restarted from the last checkpoint. However, the reconfiguration becomes challenging as the size of the HPC system increases, in particular in the perspective of large Petascale systems and future Exascale systems. It is anticipated that Exascale systems will experience various kind of reconfiguration events, especially faults [18]. If the momentum of the growth in the number of computational nodes continues, the manual reconfiguration, using checkpoint-restart, will soon not be agile enough to adapt to the frequency of the network changes [41, 42]. As soon as the new configuration is applied, the changes in the underlying network structure requires another new configuration. Therefore we study methodologies

that can dynamically reconfigure a network configuration after the occurence of events such as component failure or network congestion. More specifically, we address the challenges describe in the following research questions:

- Component failure in a large-scale system is inevitable because there are thousands of operating nodes, and millions in the future. From the interconnection network point of view, apart from generating a new deadlock-free routing structure, a fault tolerant mechanism should reconfigure the existing connections with the updated network configuration and management system. Thus, our first research question is: *how can we provide an application–transparent fault–tolerant mechanism that can dynamically reconfigure the established connections after a fault happens in the interconnection network?*

- Congestion is said to occur in a network when the resource demands exceed the network resources capacity and packets are blocked due to queuing in the network. In the interconnection network, congestion can happen due to a traffic pattern change, after component failure, or during job migration. Today, most of the congestion control mechanisms available are technology specific. To apply a similiar concept in other technologies, it is required to add new hardware features. Thus, the second research question is: *how can we provide a technology–independent congestion control mechanism that can readjust the network configuration dynamically during congestion to minimize the negative effect of head-of-line blocking [95] in a lossless interconnection network?*

The configuration of a network device

One key feature of system virtualization is *live migration*, the capability to move virtual machines and its corresponding applications and data between physical servers with minimal service interruption [26]. On the other hand, one key property of the interconnection network is the capability to carry out high throughput and low latency communication, such as remote direct memory access (RDMA) [1], a mechanism to access memory of a remote computing without the involvement of operation system. Thus, an open challenge is how to preserve the high throughput and low latency properties of an RDMA device in the virtualized environment. More specifically:

- The single root IO virtualization (SR-IOV) specification was introduced by PCI-SIG [4], to provide scalability while preserving the high throughput and low latency properties of interconnection networks. Each PCI device exposes multiple virtual devices that can be assigned to multiple virtual machine [92]. However, SR-IOV has the same drawback as PCI passthrough,

it complicates live migration. Thus, the third research question is: *how can we provide an application–transparent mechanism that can dynamically reconfigure the device resources and properties of the RDMA operations associated with live migration of a VM?*

1.2 Research Methods

In addition to the background study on related literature, we apply two research methods during this work: *prototyping* and *simulation* [101]. Prototyping is a research method that designs, implements, and evaluates the proposed concept on a real system. On the other hand, simulation is a research method that builds a software model that can simulate the environment of a system and imitate different scenarios without having the risk of disrupting a real system. The following paragraphs briefly explain the reasons why these methods are chosen and discuss in which scenarios they are well-suited.

Prototyping is an expensive and time-consuming method. Nevertheless, it is the main research method in this work. One reason is that we need the complete software stack and hardware components in a HPC system to study various events, such as faults, traffic pattern changes, or virtual machine migration. A complete system can help us to understand the complete flow of the interaction between components in a HPC system. E.g, how an exception is being handled by the application after the generation of a hardware interrupt. However, the design and implemention of simulation environment modelling of a complete HPC system, consisting of the software stack, the hardware components and different applications in a HPC system, from scratch is very time-consuming.

Another reason for not choosing simluation is that the timing of the interactions among hardware and software components, such as how fast an interrupt is serviced by a service routine, cannot be simulated or controlled accurately. As a result, prototyping is preferred over simulation in this work. Another reason is related to the objective of this work. Herein, we need to evaluate the agility of the dynamic reconfiguration methodologies. For example, multiple faults were generated back-to-back to evaluate whether the running applications survive after the reconfiguration. Thus, prototyping, using a real cluster, is more practical to generate realistic scenarios as well as to determine whether a prototype is adequate.

The discrete–event simulation model is a well-known method in evaluating network performance of routing algorithms or in the field of network traffic simulation. A dynamic event, influenced by multiple factors, may not be suitable due to too many variables that cannot be predicted precisely using a discrete–event based simulation model. Nevertheless, a discrete–event based simulation model can be combined with mathematical modeling [64] to assess the scalability of a

concept that has been evaluated experimentally using a small–scale cluster. Thus, we have created large-scale HPC systems using the OMNet++ simulator [54, 112] in order to do *proof of concept* scalability tests. These tests are according to a set of initial parameters assumed for the real world system configuration, such as the number of generated events.

1.3 Thesis Outline

This thesis is organized into two parts: The summary and the research papers. The summary part of the thesis consists of this chapter and the following chapters: Chapter 2 discusses the background information of this thesis. Section 3 describes the contributions of each research paper included in this thesis. Then, the second part is a collection of published research papers that contains our detailed algorithms, designs, implementations and evaluation results.

Chapter 2

Background

In this section, the necessary background knowledge to understand the rest of the thesis is presented.

2.1 Interconnection Networks

Interconnection networks are used for a variety of purposes, from on-chip connection between components within computer systems (from I/O devices to processors and memories), to off-chip connection between thousands of computational nodes in a multiprocessor system. In this thesis, we only focus on the interconnection networks that connect external computer systems together in the multiprocessor systems such as data centres or supercomputers. The performance of this type of interconnection networks are highly dependent on the network configuration such as topology, switching technique or routing algorithm [32]. The following subsections briefly describe each of these properties.

2.1.1 Topologies

A network topology represents the arrangement of the channels and the nodes in an interconnection network. The selection of a suitable network topology is important because it is the first step in designing a network. The routing algorithms and the switching techniques are usually built according to the selected network topology. Based on a network topology, the interconnection network can be classified as a *shared-medium network*, *direct network*, *indirect network*, or *hybrid network*, which is a combination of any of the classes [32, 36].

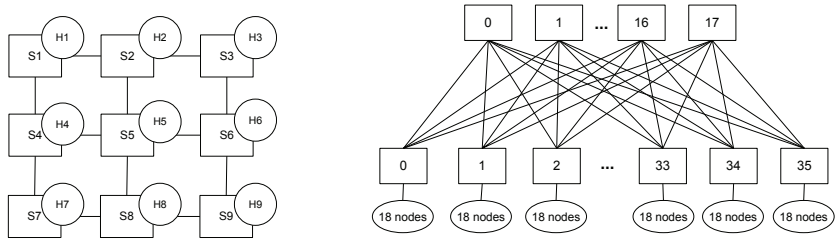
A shared-medium network has the least complex topology structure among the classes. It can be constructed with fairly low cost and supports broadcast

communication by nature. However, the transmission medium is shared among all the communicating devices where only one device is allowed to use the transmission medium at a time. Two examples of shared-medium networks are the bus topology and the ring topology.

In a direct network the network topology consists of a set of nodes that are being connected directly to the neighbouring node via a point-to-point link. A node can also communicate with a non-neighbouring node by going through one or more intermediate nodes. Each node consists of a host (or several hosts) and a switch (or a forwarding unit), which handles the communication messages among nodes. On the other hand, in an indirect network the communication between two nodes, where each node is only a host, has to go through a switch. A switch then can connect via a point-to-point link to another switch, host, or a combination of host and switch. Although a direct network can easily be converted to an indirect network by separating the host and the switch in each node and connect these two components using a point-to-point link, there is a clear distinction between these two classes of networks [36]. In a direct network, each node consists of both a switch and a host. Thus, each switch is connected to at least one host. On the other hand, each node in an indirect network can be a host or a switch. So, each switch in an indirect network may be connected to zero, one, or more hosts.

In this thesis, we use both *direct* and *indirect networks*. Examples of direct networks include mesh, torus, and high-radix topologies such as flattened butterfly [76] and dragonfly [77]. Torus and mesh topologies, used in research paper I and II [58, 61], are referred as k-ary-n-cubes and k-ary-n-meshes, respectively, where k represents the number of nodes in each dimension and n represents the number of dimensions in a network topology. k-ary-n-cubes is a k-ary-n-meshes with a wrap-around links. Fig. 2.1(a) shows 3-ary-2-mesh, which is a two dimensional mesh with three nodes in each dimension. k-ary-n-cube topologies are also a popular topology in the top500 list installations such as the 3D torus topology in Oak Ridge National Laboratory's Titan [80], the 5D torus topology in Lawrence Livermore National Laboratory's Sequoia [68], and the 6D torus topology in Riken's K computer [9].

In indirect networks, there are a wide range of topologies that have been proposed, ranging from regular topologies to irregular topologies [6, 12, 47, 81, 87]. A regular topology has regular connection patterns between switches, whereas an irregular topology does not. A regular topology, nevertheless, can turn into an irregular topology when a component fails. Today, the dominating class of topology in indirect network is the multistage interconnection network (MIN) [79]. One of the commonly used MINs is the fat-tree [82] as shown in Fig. 2.1(b). The Fat-tree, which is the topology used in research paper III and IV [57, 60], is a layered network topology following the m-port n-tree [66] definition or the k-ary n-tree [93] definition. Given an m-port n-tree, it has the following characteris-



(a) (b)

Figure 2.1: Fig. (a) is a *3*-ary *2*-mesh topology. In this topology, S{1..9} represent switches and H{1..9} represent compute nodes. Fig. (b) is a *2*-stage *648*-port fat-tree topology with 18 root switches, 36 leave switches and 648 compute nodes.

tic: The tree consists of $2 * (m/2)^n$ processing nodes and $(2n-1) * (m/2)^{n-1}$ communication switches. Furthermore, each communication switch has m communication ports. On the other hand, given a k-ary n-tree, k represents half the number of ports of each switch and n represents the number of levels of the tree. Thus, a k-ary n-tree has n levels made out of k^{n-1} switches, each having arity of k. The compute nodes are usually connected to the leaf switches. A leaf switch is the switch located at the bottom layer of a fat-tree. Transmission bandwidth between switches is increased by adding more links in parallel as the switches are closer to the root switch. The Fat-tree became the topology of choice due to its inherent deadlock freedom, fault tolerance, and full bisection bandwith propoperties. It is used in many installations in the Top500 list, including the NUDT's TianHe-1A [116], TACC's Stampede [19] and Leibniz Rechenzentrum's SuperMUC [20].

As we have discussed, both direct and indrect network topologies are implemented in the current supercomputers. To be specific, fat-tree topologies and k-ary-n-cube topologies are the two most popular ones.

2.1.2 Switching

Switching techniques determine how messages are forwarded through the network, herein determining how and when buffers and switch ports are allocated and released; thereby, determining the timing when packets or packet/message fragments can be forwarded [36]. A switching technique is also tightly coupled with flow control and buffer management. Today, there are various types of switching techniques. Each of these techniques has its own advantages and dis-

advantages, and the following paragraphs briefly discuss four of these switching techniques.

In *circuit switching*, a physical path from the source to the destination must be reserved before the transmission. The established communication channel guarantees full bandwidth and remains connected until the session ends. Nevertheless, the major drawback of circuit switching is that all resources must be reserved before the communication starts. For the whole length of the communication session between the two communicating nodes, the communication channel is dedicated and exclusive, and released only when the session terminates. Otherwise, the communication cannot be established if the communication channel is busy. Another problem with circuit switching is the delay during the connection setup phase. If only short messages are required to be transmitted, the channel setup time may take longer than the data transmission time.

Alternatively, another switching technique named *packet switching* has been widely used. In packet switching, the message is partitioned into packets. The first few bytes in a packet contain the routing information. This routing information is used by nodes to transmit each packet from the source to the destination. *Store And Forward (SAF)* switching is one of the fundamental mechanisms in packet switching. Before a packet is forwarded to the next node, the packet is buffered at the intermediate node. Although SAF is easy to implement, the main drawback is that each node requires the buffer size of at least the size of a packet. If the packet size increases, the buffer size at each node must also be increased. Another drawback of this method is that each packet must be completely buffered at a node before it can be forwarded to the next hop. Thus, the latency of a packet is proportional to the number of hops between the source and the destination nodes.

In order to reduce the packet latency of SAF switching, *virtual cut-through (VCT) switching* was introduced [75]. Owing to the fact that the first few bytes of a packet (packet header) usually contains the routing information, the router can forward the packet once the header is received. Thus, VCT requires each packet to be split into smaller units named *flits*. The packet header that contains the routing information should fit into the first flit and the rest of the packet is further split into several data flits. If there is no contention of buffer resources, the router can start forwarding the header flit and the remaining data flits once the routing information has been made. The flit is not buffered at the output buffer, but forwarded to the input buffer of the next node. However, if contention happens and the packet header is blocked because of a busy output channel, VCT behaves similarly to SAF where the entire packet is buffered at the node. Therefore, even though the flit may cut through the output buffer of a node, each node must have the buffer size of a complete packet.

Another alternative switching technique is the *wormhole switching* [31, 30].

One similarity between the wormhole switching and VCT is that each packet is broken into flits. The difference, however, is that the buffer requirement of each node in the wormhole switching are reduced compared to the buffer requirement in the VCT switching. In wormhole switching, similar to VCT, the entire packet is not buffered at a channel if the required output channel is not busy. In a blocking scenario, on the other hand, the wormhole switching reacts differently from VCT and SAF. Instead of buffering a complete packet as in VCT and SAF, the flits of a packet in wormhole switching are occupying several buffers across several nodes in the network, like a worm. Although this property removes the dependency between packet size and buffer size, it complicates the design of deadlock freedom in routing because the blocking resources are distributed across several nodes.

Regardless of having the message divided into the unit of flit or the unit of a packet, messages are queued at the input or output buffer of each physical channel. Therefore, once a buffer is occupied, the physical channel is blocked. In order to overcome this limitation, *virtual channels* were introduced. Virtual channels is a concept that splits a physical channel into several logical or virtual channels, where each channel has its own buffer and private flow control [31, 29]. To implement this, each physical channel is associated with several small buffers that each corresponds to a virtual channel, rather than a single deep buffer. As shown in Fig. 2.2, the physical link is partitioned into several virtual channels, from $VC0$ to VCx, with each virtual channel having its own buffer. Virtual channels is particularly beneficial with wormhole switching. For instance, in the absence of virtual channels, the flits of a blocked packet will occupy the buffer of several physical channels. On the other hand, virtual channels allow packets to pass a blocked packet by using a separate virtual channel, making use of idle channel bandwidth. Today, many interconnection network technologies support virtual channels. For instance, the InfiniBand architecture supports up to 16 virtual channels. Although virtual channels was originally introduced to solve deadlock and also for quality of Service (QoS) in wormhole switching, it has been widely used to improve message latency and network throughput. For instance, virtual channels can be used to avoid the negative phenomenon of Head-of-line blocking (HOL). This issue will be further discussed in the research paper III [57] and IV [60].

2.1.3 Routing

The routing algorithm is a function determining the path that any packet should take when traversing the network. It is tightly coupled with the underlying network topology and the applied switching technique. In the process of assigning the path for each source and destination pair, it must also ensure that the network

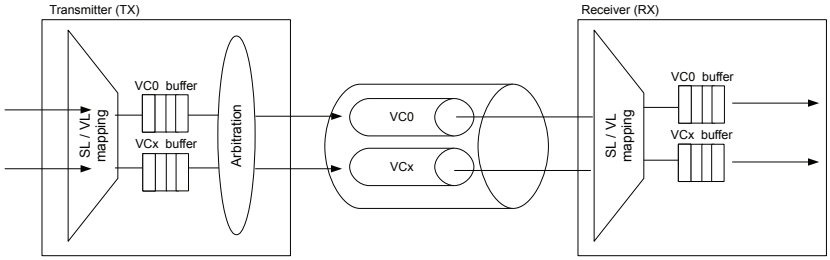


Figure 2.2: The virtual channel.

is always deadlock free. Apart from deadlock freedom, a good routing algorithm should also load balance the traffic across the network paths. The more balanced the channel load is, the higher the throughput of a network will be. A routing algorithm can be designed for regular topologies or for irregular topologies. The routing for regular topologies are heavily depending on the exact structure of the network topology, and cannot be easily transferred to other regular topologies. The routing for irregular topologies can support all network topologies. However, routing restrictions must be imposed to support different structures in various network topologies.

In general, a routing algorithm can be divided into three main categories: *deterministic*, *oblivious* and *adaptive* [32, 36]. The routing function of a determinsitic routing can be defined as $output_{port} = R(current, next, input_{port})$, where *current* is the current node, *next* is the next hop (next destination in the fabric), $input_{port}$ is the port that has received packet at current node, and $output_{port}$ is the port of the next hop. In deterministic routing, all the packets from a given source ($input_{port}$) to a given destination ($output_{port}$) always follow the same path. In non-deterministic routing, on the other hand, a given source to a destination does not always follow the same path. An oblivious routing algorithm routes packets without considering the network state. The function of oblivious routing algorithms can be described as $outputs_{port} = R(current, next, input_{port})$, where $outputs_{port}$ is an array of possible output ports. The selection of the output ports is determined by the algorithm in the oblivious routing function that is a choice between locality or load balancing. For example, Valiant's randomized routing, an oblivious routing algorithm, balances the load of any traffic pattern. A packet is first sent to a random node before the packet is directed to its destination. However, this load balancing comes at an expense of destroying the locality in the traffic pattern. Even the nearest neighbour traffic gives no better performance

than the worst-case traffic.

In adaptive routing the network state, such as the status of a link, the input and output queues for network resources, are used to select the path to deliver a packet. Because of this, an adaptive routing algorithm is intimately coupled with the flow-control mechanism. Designing an adaptive routing algorithm is complex because the routing decision must balance the local load and make sure it does not result in global imbalance. The routing function of an adaptive routing is represented by $outputs_{port} = R(current, next, input_{port}, states)$, where $states$ consists of states of the $input_{port}$ or $outputs_{port}$. As various states can be taken into consideration, great flexibilities is allowed in constructing paths through the network. The adaptivity in the routing, nevertheless, is specific to the interconnect technology. For instance, some technologies may take congestion control into account, others may not. In theory, adaptive routing should be better than an oblivious routing algorithm. However, it is not guarantee in practice because only local information is used.

In this thesis, we only focus on the deterministic routing because InfiniBand, the interconnection network that we use, currently only supports this type of routing. The support of adaptive routing in InfiniBand is still in development when this thesis was carried out. In the following paragraphs, we first discuss the importance of deadlock freedom in a routing algorithm. Then, we discuss two deterministic routing algorithms, LASH routing [84, 108] and fat-tree routing [51, 117], that we have used in this thesis.

Deadlock

Deadlock will occur in a network when a group of messages (packets), are unable to make progress because they are mutually waiting for one another to release resources, usually buffers or channels [36]. Figure 2.3 illustrates an example of a deadlock scenario where the network is halted because each FIFO buffer is waiting for an empty slot from other buffers in a circular manner. One simple way to deal with deadlock is to start dropping packets. However, most interconnection networks, that are capable of performing high bandwidth and low latency communication, are lossless. Dropping packets is not an option in a lossless network.

A deadlock scenario can be identified by studying the channel dependency graph (CDG) of the network. A CDG is a graph where the channels in the network are vertices. When a packet holds a channel and requests another channel, there is a directed edge (dependency) between them. A deadlock may exist in a network if there is a cycle in the CDG. On the other hand, the network is deadlock free if there are no cycles in the CDG. It is fundamental to handle the deadlock problem when designing a routing algorithm in the interconnection network.

There are three strategies for deadlock handing: deadlock prevention, dead-

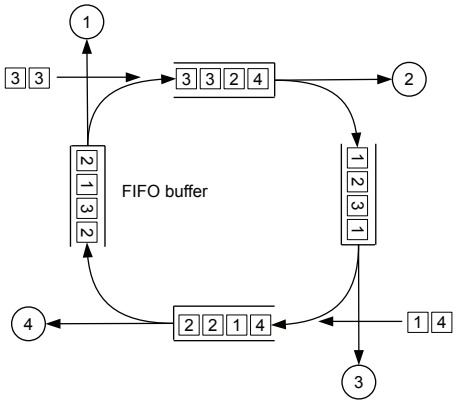


Figure 2.3: A deadlock scenario in a FIFO buffer.

lock recovery and deadlock avoidance [36]. To prevent blocked resources that paralyze the network operations, one of these strategies must be chosen. In deadlock prevention, resources (channels and buffers) are granted to the packet by reserving all the required resources throughout the network before starting the packet transmission. In deadlock recovery, resources are granted to a packet without any check, therefore, deadlock is possible and some detection and recovery method must be provided. In deadlock avoidance, resources are requested as the packet advances through the network, but resources are only granted to a packet if the resulting global state is safe. This can be achieved by imposing routing restrictions to certain paths for packet forwarding to avoid cyclic dependencies, E.g by using dimension ordered routing in k-ary-n-meshes [115]. Another approach is to distribute the source and destination pairs over different virtual channels to break cycles, E.g by using LASH routing [43].

LASH routing

LAyered SHortest path (LASH) routing is a deterministic and topology agnostic routing engine that can guarantee shortest path and deadlock–freedom even in irregular topologies [84, 108]. The concept of this routing is to attain the deadlock–freedom by segregating the traffic into different virtual layers using virtual channels. The LASH routing engine consists of two core functions. The first

function defines the minimal path for each <source, destination> pair. The second function, on the other hand, is responsible to assign a virtual layer (VL) to the <source, destination> pair generated by the first function to ensure deadlock freedom in the network. In addition, the routing algorithm makes sure that each virtual layer is deadlock free by ensuring that the channel dependencies stemming from the <source,destination> pairs of a layer do not generate cycles.

One of the main challenges with the LASH routing algorithm, is that a large number of VLs are needed to ensure deadlock-free in a large network [108]. If the network contains less VLs than needed by the LASH routing algorithm, the remaining <source, destination> pairs can still be routed deadlock free, but not shortest-path [107]. Another drawback of the LASH routing algorithm is that a single failure may result in many <source, destination> pair changes in the VL assignment in a large network.

Fat-tree routing

The fat-tree was first introduced by C. Leiserson in 1985 [82] and has since become a common topology in HPC. The fat-tree routing algorithm, as presented by Zahavi et al. [117], consists of two distinct stages: the upward stage in which the packet is forwarded from the source, and the downward phase when the packet is forwarded toward the destination. The transition between those two stages occurs at the least common ancestor, which is a switch that can reach both the source and the destination through its downward ports. The algorithm not only ensures deadlock-freedom but every path toward the same destination converges at the same root node, which causes all packets toward that destination to follow a single dedicated path in the downward direction [51, 117]. By having dedicated downward paths for every destination, contention in the downward stage is effectively removed (moved to the upward stage). Packets for different destinations have to contend for output ports, is only half of the switches on their paths. In oversubscribed fat-trees, the downward path is not dedicated and is shared by several destinations.

2.1.4 Congestion Control

Network congestion occurs when a number of links (or link) are carrying more packets to a (switch or node) port than it can accommodate. There are usually a limited number of flows that are the cause of this congestion, while the remaining flows are the victim flows that suffer because of it. The congestion behavior of a lossy network is also different than the behavior in a lossless network. In a lossy network like TCP/IP, congestion causes packet drops, and the congestion remains isolated in a small region. However, many high-speed interconnection

networks are designed to be lossless and the congestion could spread to the whole network [53].

In a lossless interconnection network, the credit-based flow control mechanism [29] is used to prevent a switch from transmitting a packet when the downstream switch lacks sufficient buffering to receive it. This property prevents packet dropping at switches and avoids the well-known congestion collapse scenario of traditional networks [72], but it may cause an undesired effect known as congestion spreading or tree saturation [33]. Since 1985, it has been know that hot-spot traffic patterns can cause congestion spreading especially in multistage interconnection networks [94]. Common sources of hot-spot traffic patterns are virtualization, migration of virtual machine images, checkpoint and restore mechanisms for fault tolerance, storage and I/O traffic. With virtualization, algorithmic predictability of network traffic patterns is reduced because multiple virtualized clients reside on the same physical hardware. The network traffic becomes an overlay of multiple traffic patterns that might lead to unpredicatble *hot-spots* in the network. When a hot-spot exists in a network, the flows designated for the hot-spot might reduce the performance for other flows, called *victim flows*, not designated to the hot-spot. This is due to the head-of-line (HOL) blocking phenomena created by the congested *hot-spot* [95].

To overcome the congestion problem mentioned above, there are two different schemes: *congestion avoidance* and *congestion control* [72]. A congestion avoidance mechanism prevents a network from entering the congested state, e.g the network traffic load is being monitored periodically to avoid congestion at common network bottlenecks. On the other hand, congestion control is a recovery mechanism where it helps the network to recover from the congested state. We focus on the congestion control mechanism in this thesis and the following paragraph explains several existing works in congestion control.

In lossy networks, such as LANs and WANs, congestion has been a widely studied problem. Therefore, there are multiple established solutions available today to resolve congestion in such networks. The end-to-end TCP congestion control is based on an implicit detection mechanism. The window control mechanism is used to detect dropped packets or changes in latency [16, 72, 91]. As dropping packets is required in the end-to-end TCP congestion control, this scheme cannot be applied to the lossless networks. An alternative approach to implicit detection of congestion is the Explicit Congestion Notification (ECN) [45, 100] that has been deployed in ATM networks [50] and TCP [44, 99]. A similar explicit congestion notification scheme, is being used in the high-speed lossless networks such as InfiniBand [53, 55, 56, 71] and Data Center Bridging [69, 102]. The ECN will throttle the data transfer rate of source nodes according to the rate allocation algorithm.

2.2 Fault Tolerance

Fault tolerance is defined as the ability of a system to continue operating, even with degradation of the network capacity, in the event of one or more failures in the system component [97]. Although the current-generation hardware component is robust enough to handle faults and does not fail very often by itself, it is imperative that failure can still be anticipated especially in a huge installation that consists of thousands of components connected via an interconnection network and running with multiple applications, e.g. supercomputers or data centres. Capello et. al claimed that if the number of cores in the top500 supercomputers continue to double every 18 months and with a constant failure per socket, the statistic shows that the mean time between failure (MTBF) will reach 1 hour between 2013 to 2016 [18]. The MTBF of the Exascale system is predicted to be worse. As a result, fault tolerance is vital in interconnection networks because it determines the reliability, availability and dependability of a system as a whole.

Fault tolerance in interconnection networks can be divided into two categories, *static* or *dynamic* fault tolerance [36]. Static fault tolerance is for faults that are known when the system is started and do not have the constraint of hard real-time requirement. When a fault happens, static fault tolerance requires the network to be shutdown. Then, the network is reconfigured, either by generating new routing tables or replacing the failed hardware. The fail-over mechanism is handled by the upper-layer applications or operating system by checkpointing regularly. After the reconfiguration, the applications may be restarted from the rollback of the last checkpoint. Although this method can handle multiple faults regardless of the underlying topologies and routing algorithms, its main drawback is that it is not application–transparent.

Dynamic fault tolerance, on the opposite, is an application–transparent approach. Once a fault is detected, actions are taken in order to properly handle the faulty component without shutting down the system. For instance, a source node that detects a faulty component along a path can use an alternative path, that does not use the faulty component, to reach a destination. Therefore, the system keeps working, the network is not emptied, and checkpointing is not required as part of the fault tolerance mechanism.

2.2.1 Static fault tolerance

Static fault tolerance can be achieved by replacing the faulty component in the interconnection networks or by designing appropriate fault tolerant routing algorithms [52, 84, 106, 107]. Both approaches can only be implemented while there is no traffic in the network. Thus, the checkpoint/restart technique, as Elnozahy et al. presented in [38], is required. Although checkpoint/restart can be used for

other reasons, such as recovery for applications crashing, in this thesis we only focus on such usage due to the changes in the interconnect's configuration, such as interconnect unavailability due to switch failure, load balancing on an existing machine, or process migration between machines. There are many approaches to integrate the checkpoint/restart feature into applications, operating systems, or message passing interface (MPI) libraries. Since MPI, a de-facto standard for message passing in HPC systems, controls the interactions between processes, checkpoint/restart techniques can take advantage of the MPI implementation's knowledge of the distributed system's state to ensure correctness in the checkpoint algorithm. Moreover, implementing checkpoint/restart at the MPI libraries is a better solution than implementing checkpoint/restart at the application level, because they do not require the application developers to alter their algorithms. As a result, both openMPI [46] and MVAPICH2 [89] have implemented checkpoint/restart fault tolerance. MVAPICH2 demonstrated transparent MPI checkpoint/restart functionality over InfiniBand interconnects in [48]. On the other hand, OpenMPI has an interconnect–agnostic approach that support transparent checkpoint/restart with different interconnects [67].

Today, the HPC systems have relied primarily on checkpoint/restart techniques. However, future HPC systems, such as exascale systems, are expected to present a much more challenging fault tolerance environment [78]. Additionally, recent studies conclude that for these systems, high failure rates coupled with high checkpoint/restart overheads will render current rollback-recovery approaches infeasible. For example, several independent studies have concluded that potential Exascale systems could spend more than 50% of their time reading and writing checkpoints [37, 90, 103].

2.2.2 Dynamic fault tolerance

Dynamic fault tolerance does not require that faults are known a priori. On the opposite, dynamic fault tolerance mechanisms can tolerate faults occurring at arbitrary times without having to shut down the entire network during the reconfiguration of the network. There has been a substantial amount of work in dynamic fault tolerance that focus on application–transparent fault–tolerance. Application–transparent fault–tolerance depends on a routing algorithm that can reroute the network by avoiding the faulty component in the presence of faults. One of the challenge is to define a generic routing algorithm that can guarantee deadlock-freedom for all network topologies. As a result, almost every interconnect topology has a corresponding fault tolerant routing algorithm. In a fat tree, fault tolerance can be achieved by choosing a different root via a different upward path in the network [105]. The k-ary-n-mesh and the k-ary-n-cube topologies are more difficult to handle in terms of fault tolerance. Ho and Stockmeyer

introduced a fault tolerant routing function that sacrifies a certain number of healthy nodes to perform routing rather than processing. This algorithm needs no more than two virtual channels and it reduces routing time [63]. Boppana and Chalasani defined a protocol for deadlock–free rerouting around faulty regions in meshes [15, 21], and Montanana et al. for torus networks [88]. There is also several works that limit the number of faults. Lysne et al show that a single link fault can be tolerated when using XY routing in a mesh, simply by creating a path around a link fault where the first turn is towards the centre of the mesh [85]. Sem-Jacobsen et al. proposed a solution that is independent of topology and routing functions, but only can tolerate a single fault [104]. The main weakness of these approaches for application transparent fault-tolerance is that they are highly inflexible. They either work for a limited set of topologies, or for a limited set of fault situations. This led to some efforts on *dynamic reconfiguration* in routing. The term dynamic reconfiguration mentioned here only focus on the routing algorithm. On the opposite, the title of this thesis, as explained in Section 1, refers to the configuration of the interconnection networks. Dynamic reconfiguration in routing is usually achieved in two ways. Either the entire network is divided into several virtual networks using virtual channels, allowing one virtual network to be reconfigured while the other virtual networks are in operation, or only parts of the network are reconfigured, allowing the rest of the network to remain in operation. Pinkston el al. proposed a double scheme approach which can support numerous routing algorithms [96]. The idea is to move from one routing function to another while the system is up and running. Although it also supports different topologies and any number of faults, the difficulty of this solution is to ensure deadlock–freedom in the handoff phase from one to another [35].

2.3 Virtualization

The concept of virtualization was pioneered by IBM in the late 1960s to allow IBM mainframes to run multiple applications and processes simultaneously. The IBM's hypervisor (CP-67) enabled memory sharing across virtual machines, providing each user with a private memory space [98]. While its impact was substantial for mainframe users, it took years before a direct descendant of IBM's work came back to life in 1999 when VMware revived the concept and applied it to the x86 platforms.

In general, virtualization refers to the creation of virtual resources (such as hardware devices, storage devices, or network resources) that can be shared among two or more operating systems (OS) and applications that run concurrently [17, 23]. One of the key components in virtualization is the Virtual Machine Monitor (VMM), also known as the *hypervisor*. The VMM is a software abstrac-

tion layer that is responsible for managing the execution of guest OS instances sharing the virtualized hardware resources [109]. A VMM can be categorized into type I or type II [49]: A type I VMM runs directly on the host hardware, has exclusive control over the hardware resources, and is the first software to run after the boot loader. The VMs run in a less privileged mode on top of the VMM. Well known type I VMMs include the original CP/CMS hypervisor [28], VMWare ESXi [22], Microsoft Hyper-V [113] and Xen [13]. A type II VMM runs as a privileged process on top of a conventional operating system and the VMs run on top of this privileged process. The type II VMM controls and schedules the access to hardware resources for the VMs. Well known type II VMMs include the VMware GSX server [114], KVM [62], and VirtualBox [3].

Virtualization has been widely deployed in today's computer systems, ranging from embedded systems to HPC, because of its ability to maximize the hardware utilization. Although virtualization has become a viable technology, it still poses a lot of challenges [40]. One challenge is the bottleneck in I/O virtualization due to the performance cost of a software I/O virtualization layer in the VMM [7]. It is challenging to efficiently virtualize I/O devices because each interaction between a guest OS and an I/O device needs to undergo costly interception by the VMM for security isolation and for data multiplexing and demultiplexing. This problem is particularly acute when virtualizing high-speed networking devices because of its high rate of packet transmission. As each transmission must be trapped by the VMM, the overhead of the high frequency of trapping will impact the performance in the high-speed network. Thus, a lot of research have been carried out in I/O virtualization to find out the balancing point between *efficiency*, *transparency* and *scalability* [13, 24, 25].

I/O Virtualization (IOV) was intially introduced to provide availability of I/O by allowing VMs to access the underlying physical resources. With the increased number of VMs per host that indirectly increases the number of I/O requests, the goal of IOV is not only to provide availability, but to improve efficiency, scalability and transparency of the I/O resources to match the level of performance seen in modern CPU virtualization. Today, there are four well-known IOV techniques: i) *Emulation*, ii) *Paravirtualization*, iii) *PCI pass-through* and, iv) *Single Root-IOV (SR-IOV)*. The benefits and drawbacks of these IOV techniques are summarized in Table 2.1.

The emulation technique emulates the I/O devices in the VMM and exposes an emulated device to the guest OSes [7]. This solution has very high *transparency* because no modification in the OS is required and it works with different OSes. Moreover, the guest OS can be migrated to another platform easily. The emulation technique has medium *scalability*. Even though an emulated device can be accessed by multiple VMs concurrently, when the number of guests increase the load on the emulated device and the I/O domain increases proportionally,

which limits scalability significantly. One major concern with emulation is the poor *efficiency*. This is because each I/O operation needs to be intercepted by the VMM, which degrades the performance of the emulated device.

To overcome the performance bottleneck caused by emulation, paravirtualization was therefore introduced [13]. Unlike emulation, paravirtualization requires modification in the guest OS. Although modifying the guest OS improves the *efficiency* of the virtualized I/O, the *transparency* is sacrificed. The *transparency* decreases because each guest OS must be modified to support a paravirtualized I/O device, which means that a particular OS or distribution may not be readily available for the solution. In terms of *scalability*, the paravirtualization technique is scalable, similar to emulation, where a paravirtualized I/O device can be accessed by multiple VMs simultaneously. Nevertheless, the I/O performance of a paravirtualized device is still worse than the native I/O device.

To further improve the *efficiency* of IOV, PCI pass-through, also known as direct-assignment, was introduced [7, 83]. PCI pass-through provides exclusive access to an I/O device for a given guest OS, which yields near to native performance. However, PCI pass-through can only achieve its full potential with hardware that is equipped with instructions as well as logic for PCI pass-through, such as interrupt remapping and direct memory access remapping. Both Intel and AMD has already provided support for PCI pass-through in their processor architectures [7, 10]. Despite of the high *efficiency* provided by PCI pass-through, it does not *scale* as it can only allocate as many pass-through devices that are physically present in the platform. Moreover, PCI pass-through sacrifies *transparency* because the guest OS with a pass-through device cannot support VM migration in a transparent manner. This is due to the fact that the VMM has no knowledge of the device state as the device is directly assigned to the guest OS [74, 118].

In order to address the lack of *scalability* in PCI pass-through, the SR-IOV specification was created by the PCI-SIG [92]. With SR-IOV, a PCIe device can export not only just a number of physical functions, but a set of virtual functions that share resources on the I/O device. So far, SR-IOV is a scalable technique that can achieve high *efficiency* with near to native performance in IOV. The only drawback with SR-IOV, similar to PCI pass-through, is the *transparency* of the I/O virtualization [65]. Although there are many proposals to overcome each of the transparency challenges in the SR-IOV devices, most of these approaches focus on Ethernet only [34, 74, 118] and they cannot be applied to high-speed, lossless interconnection networks such as InfiniBand, Myrinet [14] and RDMA Ethernet [110]. As a result, we focus on the challenges in live migrating an SR-IOV high-speed networking devices, in research paper V and VI [59], to achieve high *efficiency*, *transparency* and *scalability* in the I/O virtualization of high-speed interconnection networks.

Technique	a) Efficiency	b) Transparency	c) Scalability
Emulation	Low	Very high	Medium
Paravirtualization	Medium	Medium	High
PCI pass-through	High	Low	Low
Single Root-IOV	High	Low	High

Table 2.1: The comparison between different IOV techniques.

2.4 The InfiniBand Architecture

Among all the available interconnects, InfiniBand (IB) is one of the most popular interconnects in the TOP500 supercomputer list due to its properties of low latency and high throughput [5]. The IB Architecture [71] was first standardized in October 2000, as a merge of the two technologies, Future I/O and Next Generation I/O. As with most other recent interconnection networks, IB is a serial point-to-point full-duplex technology. IB networks are referred to as subnets, where a subnet consists of a set of hosts, with *host channel adaptors* (HCAs), interconnected using switches and point-to-point links as illustrated in Fig. 2.4. The link speed of IB was, initially, specified as 2.5 Gbps with the possibility of bundling links in 4x or 12x configurations. Today, the supported link speed of Fourteen data rate (FDR), is specified at 13.64 Gbps. The effective transfer speed of FDR is 54.54 Gbps and 163.64 Gbps in 4x and 12x configurations, respectively. Each link can be divided into 16 virtual lanes (VLs). VL0 must be configured as data traffic and VL15 as the management traffic. Flow control is used to manage data flows between two point-to-point links and it is handled on a per VL basis. Each receiving end of a link supplies credits to the sending device on the link to specify the amount of packets that can be received without loss of data. Data is not transmitted unless the receiver advertises credits indicating that receive buffer space is available.

The IB subnet routing can support inter- and intra-subnet routing. Within a subnet, routing is handled at the link layer using a 16 bit Local identifier (LID), limited to 48K unicast addresses, assigned by the subnet manager (as explained in Section 2.4.1). The packets are sent to the device within a subnet by using the destination LID that is encapsulated in a Local Route Header (LRH) of a packet. The network layer routing handles the routing between subnets. Packets that are sent between subnets contain a Global Route Header (GRH). The GRH contains the 128 bit IPv6 addresses for the source and destination of the packet. Packets are forwarded between subnets through a router, based on each device's 64 bit globally unique ID (GUID). The router modifies the LRH with the proper local address within each subnet. Therefore the last router in the path replaces

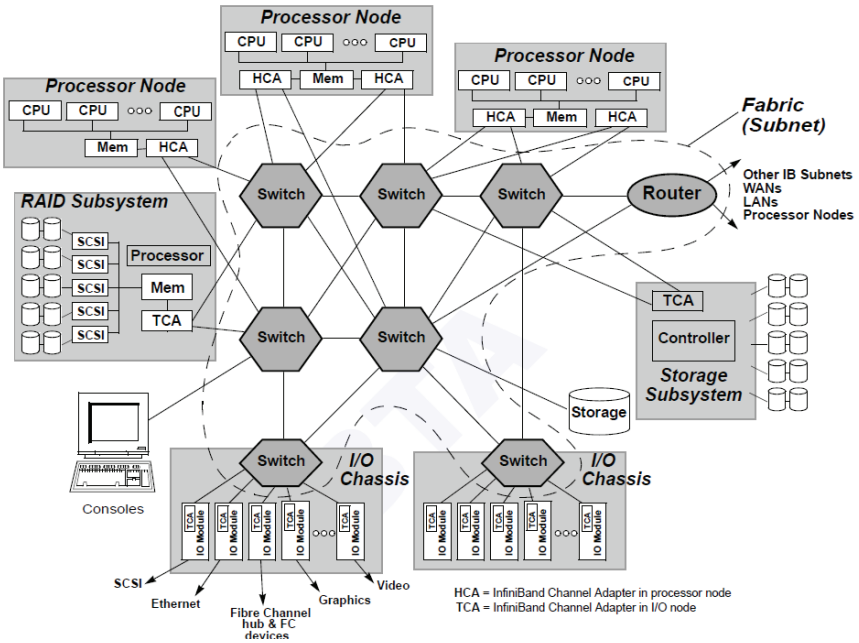


Figure 2.4: The InfiniBand subnet (courtesy of the IB specification). TCA represents the Target Channel Adapter, whereas HCA represents the Host Channel Adapter.

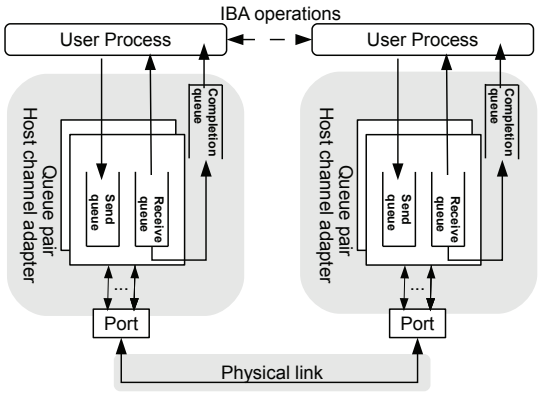


Figure 2.5: The conceptual diagram of the IB communication using Queue Pairs.

the LID in the LRH with the LID of the destination port. Today, most of the popular routing algorithms in IB are determinsitc. E.g. fat-tree, LASH, and dimension-order-routing (DOR).

IB supports a rich set of transport services in order to provide both remote direct memory access (RDMA) and traditional send/receive semantics. Independent of the transport service used, all IB HCAs communicate using queue pairs (QPs). A QP is created during the communication setup, and a set of initial attributes such as QP number, HCA port, destination local identifier, queue sizes, and transport service are supplied. As shown in Fig. 2.5, an HCA can handle many QPs, where each QP consists of a pair of queues, a send queue (SQ) and a receive queue (RQ), and there is one such pair present at each end-node participating in the communication. The SQ holds work requests to be transferred to the remote node, while the RQ holds the work request to handle the data received from the remote node. In addition to the QPs, each HCA has one or more completion queues (CQs) that are associated with a set of send and receive queues. The CQ holds completion notifications for the work requests posted to the send and receive queue. Even though the complexity of the communication is hidden from the user, the QP state information is kept in the HCA.

IB also provides a congestion control (CC) mechanism [71]. The IB CC mechanism is based on an explicit congestion notification, where a switch detecting

congestion marks packets contributing to the congestion by setting a specific bit in the packet headers, the Forward Explicit Congestion Notification (FECN) bit. The packet with the FECN bit set will reach the destination. Then, the destination registers the FECN bit, and returns a packet to the source with the Backward Explicit Congestion Notification (BECN) bit set. The source then temporarily reduces the injection rate to resolve congestion. The exact behaviour of the IB CC mechanism depends upon the values of a set of CC parameters governed by a Congestion Control Manager. These parameters determine characteristics like when switches detect congestion, at what rate the switches will notify destination nodes using the FECN bit, and how much and for how long a source node contributing to congestion will reduce its injection rate. If these parameters are set appropriately, the IB CC should enable the network to resolve congestion, avoiding head-of-line blocking, while still utilizing the network resources efficiently [53].

2.4.1 Subnet Management

An IB subnet requires at least one subnet manager (SM) which is responsible for initialising and bringing up the subnet, including the configuration of all the IB ports residing on switches, routers, and HCAs in the subnet. At the time of initialisation, the SM starts in the *discovering state* where it does a sweep of the network in order to discover all switches and hosts. During this phase, it will also discover any other SMs present and negotiate who should be the master SM. When this phase is complete, the master SM enters the *master state*. In this state, it proceeds with LID assignment, switch configuration, topology discovery, routing table calculations and deployment, and port configuration. When this is done, the subnet is up and ready to use.

After the subnet has been configured, the SM is responsible for monitoring the network for changes (E.g. a link goes down, a device is added, or a link is removed). If a change is detected during the monitoring process, a message (*trap*) is forwarded to the SM and it will reconfigure the network. A major part of the reconfiguration process (also known as "heavy sweep") is the rerouting of the network which must be performed in order to guarantee full connectivity, deadlock freedom, and proper load balancing between all source and destination pairs.

2.4.2 Subnet Administration

The Subnet Administrator (SA) is a subnet database built by the master SM to store different information about a subnet. Communication with the SA is often needed by the two end-nodes to establish a QP. This is accomplished by sending a general service management datagram (MAD). Both sender and receiver require

information such as source/destination LIDs, service level (SL), MTU, etc. to establish a QP, and this information can be retrieved from a data structure known as a path record that is provided by the SA. In order to obtain a path record, the end-node can use the *SubnAdmGet/SubnAdmGetTable* operation to perform a *path record query* to the SA. Then, the SA will return the requested path records to the end-node. The term path record is an IB term that has the same meaning as path information.

Chapter 3

Summary of Research Papers

This section summarizes the contributions of six research papers in this thesis. Five papers have been published in peer-reviewed conferences whereas the sixth paper has been submitted to the IEEE Transactions on Parallel and Distributed Systems journal. The objective of these research papers is to provide methods for dynamic reconfiguration in interconnection networks. In particular, we answered the three research questions mentioned in Section 1.1.

In paper I [61] and II [58] we answered research question one: *how can we provide an application-transparent fault tolerant mechanism that can dynamically reconfigure the established connections after a fault happens in the interconnection network?* We proposed host side dynamic reconfiguration, a fault tolerant mechanism that can reconfigure the path information in the established connections after faults happen. Moreover, this method is application-transparent and the running applications can continue uninterruptedly during the occurrence of faults.

Paper III [57] and IV [60] are in the context of congestion control to answer research question two: *how can we provide a technology–independent congestion control mechanism that can readjust the network configuration dynamically during congestion to minimize the negative effect of head-of-line blocking in a lossless interconnection network?* We proposed *vFtree* and *dFtree*, two routing algorithms that can optimize the network throughput by alleviating the negative effect of HOL blocking during congestion. Both routing algorithms only require virtual channels, a common component that is available in most of the interconnect. Thus, these routing algorithms can easily be applied to different network technologies.

In paper V and VI [59] we study the interoperability of the interconnection network in a virtualized environment to address research question three: *how can we provide an application–transparent mechanism that can dynamically reconfigure the device resources and properties of the RDMA operations associated with live migration of a VM?* As the outcome of this study, we proposed enhancements to the software and hardware architecture of an RDMA device. Our proposal allows the resources and properties of an RDMA device, that is directly assigned to a VM, to be reconfigured dynamically after live migration, while maintaining high throughput and low latency attributes in the interconnection network.

All the proposed methodologies in this thesis were developed using IB, the most popular interconnection network in the top 500 supercomputers list [5]. Nonetheless, the concepts are relevant for other technologies such as Myrinet [14], Virtual Interface Architecture (VIA) [27] and RDMA Ethernet [110]. The following subsections describe the contributions for each paper in detail.

3.1 Paper I: Host Side Dynamic Reconfiguration with InfiniBand

The characteristics of an interconnection network will change after component failures, policy changes or job migration. When these events happen, the routing algorithm is usually robust enough to generate a new routing table that avoids the faulty component. After generating a new routing table, the network manager, with the global view of the network, has the updated path information for any source and destination pair. Nevertheless, the host that has started the communication before the fault might not be aware of these new changes but continues with an old path information. This might cause an unexpected behaviour in the network, such as a deadlock. Paper I proposed a dynamic reconfiguration method to reconstruct the path information at the host. The main method of the proposal is based on event forwarding in the network manager to notify the affected nodes with the updated path information. Then, the host reconfigures the established connections. Using this method, the fault tolerance mechanism is transparent to the applications and the network traffic is not required to be stopped.

The prototype of *host side dynamic reconfiguration* was designed, implemented, and evaluated in a small-scale InfiniBand cluster. With this implementation, we demonstrated a fault tolerance mechanism in InfiniBand networks by combining the host side dynamic reconfiguration with the LASH routing algorithm. The LASH routing algorithm is a topology-agnostic routing function that can ensure deadlock freedom and shortest path routing for every source and destination pairs, even with irregular topologies. Nevertheless, one missing key

feature is a method to update the established connection with updated path information after the network manager has generated a new routing structure [39]. Therefore, the host side dynamic reconfiguration, proposed in this paper, can be used to reconfigure an established connection (queue pair) with the updated path information. This solution is in principle able to let applications run uninterruptedly on the cluster, as long as the topology is physically connected.

Through measurements on our test-cluster, that is a 3x2 mesh direct network routed by the LASH routing algorithm, this paper shows that the increased cost of the proposed method in setup latency is negligible, and that there was only a minor reduction in throughput during reconfiguration.

3.2 Paper II: A Scalable Method for Signalling Dynamic Reconfiguration Events in OpenSM

As mentioned in paper I [61], rerouting around faulty components, on-the-fly policy changes, and migration of jobs, all require reconfiguration of path informations in the Queue Pairs residing in the hosts of an IB cluster. In addition to a proper implementation at the host, the subnet manager (SM) needs to implement a scalable method for signaling reconfiguration events to the hosts. This is due to the nature of handshaking in the IB event forwarding. Excessive use of event forwarding might cause a bottleneck in the SM. Thus, this paper proposed and evaluated three different implementations for signaling dynamic reconfiguration events with OpenSM. The first approach, named *3-way wildcard handshake*, is a simple 3-way handshaking mechanism that only requires minimum modification in the SM. One drawback of this approach, however, is that the SM is unable to identify the changed paths, though it has the updated paths after the fault. This disadvantage will create unnecessary management message overhead in the subnet. The second approach, named *repath-only*, fully utilize the event-forwarding mechanisms in IB [71]. In this method, every forwarding event includes an updated path information. This, however, requires a method that can differentiate between the new and old paths. Therefore, we propose a new mechanism in the SM called the Path Record Distinguisher (PRD). Although this solution is straightforward and does not involve any handshaking, its scalability depends on the total number of path changes. The last signalling method is named the *3-way hybrid handshake*. Due to the fact that the second approach is vulnerable to path changes, we need a mechanism that forwards the updated path records in-bulk, without creating any unncessary management message overhead, as in the first approach. Hence, the third approach is a combination of the first and the second signaling methods, where the 3-way handshaking is combined with a PRD in the SM to minimize unnecessary overhead. This mechanism is particularly

useful for a routing algorithm that will change multiple path records even with a single fault, such as the LASH routing algorithm. If that is the case, instead of sending out a dedicated re-path trap, the path records are sent in bulks via 3-way handshaking.

In our evaluation we demonstrate a scalable solution for signalling host side reconfiguration events in an IB network based on an example where dynamic network reconfiguration combined with a topology-agnostic routing function is used to avoid malfunctioning components. By measurements on our test-cluster and an analytical study we show that our best proposal reduces reconfiguration latency by more than 90%, and in certain situations eliminates it completely. Furthermore, the processing overhead in the SM is shown to be minimal. This contribution combined with our previous work in [58] is a complete prototype for host side dynamic reconfiguration in IB.

3.3 Paper III: vFtree - A Fat-tree Routing Algorithm using Virtual Lanes to Alleviate Congestion

One reason for performance degradation in lossless interconnection networks is congestion. In fact, there are many solutions available today to avoid congestion in interconnection networks, such as CC mechanisms. Nevertheless, CC usually requires additional hardware support.

Thus, this paper focuses on a generic solution to optimize the network during the occurrence of congestion. It has been a well-known fact that multiple virtual lanes can improve performance in interconnection networks, but this knowledge has had little impact on real clusters [31, 29]. Therefore, we propose to use the combination of efficient routing and virtual lanes to alleviation congestion. The algorithm to assign the virtual lanes is based on statistical probability that distributes the traffic across a set of virtual lanes. This approach is generic and can be applied to any technology, but in this paper we are using IB.

Today, a large number of HPC systems using IB are based on fat-tree topologies [5]. A fat-tree topology does not require virtual lanes to be routed deadlock-free. If QoS is not enabled, all the remaining virtual lanes are left idle. As a result, this paper suggested an enhancement to the fat-tree routing algorithm that utilizes virtual lanes to improve performance when hot-spots are present. Even though the bisection bandwidth in a fat-tree is constant, hot-spots are still possible and they will degrade performance for flows not contributing to them, due to head-of-line blocking. Such a situation may happen when multiple virtualized clients reside on the same physical hardware, where the network becomes an overlay of multiple traffic patterns. In this case, the network traffic patterns

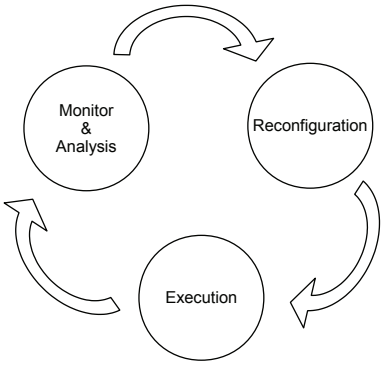


Figure 3.1: The basic feedback loop.

are unpredictable and might lead to hot spots in the network. Although congestion can be alleviated through adaptive routing, this method is not yet readily available in IB and it may not resolve congestion completely [94]. Consequently, as a first step to remedy this problem, we have implemented an enhanced fat-tree routing algorithm in OpenSM that distributes traffic across all available virtual lanes without any manual effort.

We evaluated the performance of the algorithm on a small cluster and did a large-scale evaluation through simulations. In a congested environment, results show that we are able to achieve throughput increases up to 38% on a small cluster and from 221% to 757% depending on the hot-spot scenario for a 648-port simulated cluster.

3.4 Paper IV: dFtree - A Fat-tree Routing Algorithm using Dynamic Allocation of Virtual Lanes to Alleviate Congestion in InfiniBand Networks

As mentioned in paper III [57], end-point hotspots can cause major slowdowns in interconnection networks due to head-of-line blocking. Therefore, avoiding congestion is important to ensure high network throughput. The vFtree algorithm that was proposed in paper III [57] uses a combination of efficient routing and

virtual lanes to alleviate congestion. A drawback with this approach, however, is that it is based on a probability of static distribution of source-destination pairs across a set of VLs. The static behaviour of the vFtree algorithm limits the performance whenever there is a mismatch between the current hot-spot and the precalculated distribution of source-destination pairs across VLs. An example of such scenarios happen when a component failed in an irregular topology. Therefore, we propose a dynamic approach that can identify and differentiate the congested flows. This approach is especially important in situations of permanent congestion, which causes a permanent slowdown. Permanent congestion occurs when traffic has been moved away from a failed link, when multiple jobs run on the same system, and compete for network resources, or when a system is not balanced for the application that runs on it.

The proposed dynamic approach, motivated by the closed loop system in the control theory [11], is using a feedback loop mechanism as shown in Fig. 3.1. The concept of this dynamic approach consists of three states: *Execution*, *Monitor & Analysis*, and *Reconfiguration*. After the network is initialized, it stays in the execution state. Then, the network manager will periodically monitor and analyze the network traffic. If congested flows are identified, the reconfiguration will be performed to optimize the network. Based on this feedback loop, we provide a mechanism for dynamic allocation of virtual lanes and live optimization of the distribution of flows between the allocated virtual lanes. The network traffic are divided into two lanes: the *slow lane* and the *fast lane*. If the flows are destined for an end-point hot-spot, they are placed in the slow lane. On the other hand, the normal flows are placed in the fast lane. Consequently, the flows in the fast lane are unaffected by the head-of-line blocking created by the hot-spot traffic.

We demonstrate the feasibility of this approach using a modified version of OpenSM with fat-tree routing on a small IB cluster. Our experiments show an increase in throughput ranging from 150% to 468% compared to the conventional fat-tree algorithm.

3.5 Paper V: Early Experiences with Live Migration of SR-IOV enabled InfiniBand

Virtualization is the key to efficient resource utilization and elastic resource allocation in a HPC systems and cloud computing. From the perspective of network virtualization, especially for high-speed interconnection networks such as IB, iWARP, VIA or RDMA Ethernet [110], there are two main challenges: First, *performance* and *scalability*. Second, the *transparency* in virtualization. A network interface should be able to assign to multiple virtual machines (VMs), while pre-

serving the properties of low latency and high bandwidth in the interconnection network. Each virtual network interface must provide uninterrupted networking services to the VM, especially during live migration. Fortunately, the single root IO virtualization (SR-IOV) specification addresses the performance and scalability issues. With SR-IOV, a PCI Express device can present itself as multiple virtual devices and each virtual device can be dedicated to a single VM. Each VM has direct access to the virtual device without the overhead introduced by emulation or paravirtualization. SR-IOV does not, however, address the transparency issue. SR-IOV does not permit live migration. As a result, the SR-IOV virtual device must be detached from the VM before migration and this interrupts the networking service of a VM.

Thus, in this paper we propose a mechanism for transparent live migration of VMs over high-speed and lossless SR-IOV devices. This mechanism is prototyped and evaluated using Mellanox–based IB hardware and the Xen–based Oracle Virtual Machine virtualization platform. In this mechanism, we handle: the *detachment of an active device*, the *reallocation of physical resources*, and the *reestablishment of the remote connection*. With these methods, the underlying hardware managed resources of an SR-IOV IB device are reconfigured dynamically when a VM is migrated. The on-going networking operations can be continued, with minimal service downtime during live migration. Through a detailed breakdown of the different contributors to service downtime, we pinpoint the fraction of the cost of migration that can be mitigated by architectural changes in future hardware. Based on this insight, we propose a new design of software and hardware architecture, and argue that, with these changes, the service downtime of live migration with IB SR-IOV devices can be further reduced, as demonstrated with emulated Ethernet devices [34].

3.6 Paper VI: A Scalable Signalling Mechanism for VM Migration with SR-IOV over InfiniBand

This paper is a continuation of the work in paper V. As mentioned earlier, single root I/O virtualization (SR-IOV) is a promising I/O virtualization approach for achieving high performance in the virtualization over IB networks. There are, however, several challenges with Virtual Machine (VM) migration with SR-IOV over IB [65]. One challenge is related to the hardware address assignment for each virtual IB device. In IB devices with SR-IOV support, there are two schemes for the hardware address assignment; static assignment and dynamic assignment. The static assignment is mainly targeted for the legacy applications running in VMs that have the requirement of preserving the hardware address

of its networking interface after VM migration. Thus, static assignment always preserves the hardware address of a virtual IB device that is attached to a VM. A drawback, however, using static assignment, is that its communication will be disconnected after VM migration. On the other hand, the communication can be resumed after VM migration if dynamic assignment is deployed. Nevertheless, the hardware address associated with a VM is not preserved after VM migration. Consequently, a query to the network manager needs to be performed in order to obtain the path information for the new hardware address. These operations introduce additional latency in bringing up the IB virtual function (VF). If hot migration is performed, this delay will increase the total service downtime. In short, each address assignment scheme has its pros and cons.

This paper focuses on enabling VM migration with static assignment. First, we point out the problem related to SR-IOV over IB that breaks the network connections after VM migration, when the static assignment is deployed. The main reason is the lack of notification from the subnet manager (SM). In order to resolve this problem, a SM event, that includes the latest path record, must be triggered after VM migration, to notify the migrated VM peers. Thus, we extend the repath-only signaling mechanism, that was mentioned in paper II [58], to maintain the network connectivity after VM migration. The performance evaluation using an experimental test bed shows that the proposed signaling mechanism does not increase the service downtime during hot migration. We also optimize the signaling method, where the same event can only be forwarded to a physical server once, regardless of the number of hosted VMs, to reduce the management message overhead from $O(n * m)$ to $O(n)$.

Chapter 4

Future Work

Several opportunities for further work can be extended from our contributions mentioned in section 3. The future works can be categorized into two areas: dynamic rerouting and transparent live migration with high–speed interconnects.

In the area of dynamic rerouting, one idea is to merge dFtree [60] with the IB CC mechanism. The IB CC can be used to identify hot-spots and the congestion contributors. This combination can detect the hot-spot flows faster and it is independent of the performance sweeping interval. Nevertheless, one key challenge, is to ensure that the mechanism of redirecting the congested flows to another virtual lane will not break the packet sequencing in a lossless network.

In the area of transparent live migration with high-speed interconnects, improving the interoperability of I/O virtualization with a high–speed interconnect remains as an open challenge. In a long run, we think that there are two areas of further work: The *hardware design* perspective, and the *routing* perspective. All high–speed interconnects are based on distinct hardware architectures, that provides offloading features to increase throughput and to reduce latency. In the virtualization environment, SR-IOV was introduced by PCI-SIG to achieve high performance in IO devices. Each VF is an isolated entity, that has its own memory and PCI configuration registers. However, from the high-speed interconnect perspective, each VF is still sharing the resources of a common offloading engine. This impedes the interoperability between high–speed interconnects and virtualization. Thus, a short term goal is to design, implement and evaluate the hardware architecture that we proposed in paper V.

From the routing perspective, one interesting direction is to study a new routing algorithm that can rearrange the routing information after VM migration. Ideally, the impact of the routing information should only affect the switches of the migrated VM, and should not generate a new routing table for the entire

subnet. Furthermore, the new routing algorithm must also ensure that the new arrangement is deadlock free. Another challenge from the routing perspective is the scalability issue. Due to the fact that each VF is assigned with its own address, and in a ten-of-thousands node cluster with each node hosts multiple VMs, the address space might run out. One option to solve this issue is to extend the address space, but it is not backward compatible with an older hardware.

Bibliography

[1] RFC 5040: A Remote Direct Memory Access Protocol Specification. Technical report, October 2007.

[2] The Apache Hadoop Software Framework. `http://hadoop.apache.org/`, November 2012.

[3] Oracle VirtualBox. `http://www.virtualbox.org/`, November 2012.

[4] The Peripheral Component Interconnect Special Interest Group (PCI-SIG). `http://www.pcisig.com/`, November 2012.

[5] Top 500 supercomputer sites. `http://top500.org/`, November 2012.

[6] G. A. Abandah and E. S. Davidson. Modeling the communication performance of the IBM SP2. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 249–257, Washington, DC, USA, 1996. IEEE Computer Society.

[7] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel Virtualization Technology for Directed I / O. *Intel Technology Journal*, 10(03), 2006.

[8] D. Agrawal, S. Das, and A. El Abbadi. Big Data and Cloud Computing: current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT/ICDT)*, pages 530–533, New York, NY, USA, 2011. ACM.

[9] Y. Ajima, S. Sumimoto, and T. Shimizu. Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers. *IEEE Transactions on Computers*, 42(11):36–40, November 2009.

[10] AMD. AMD I/O Virtualization Technology Specification. Press release, May 2007. http://www.amd.com/us/press-releases/Pages/Press_Release_117440.aspx.

[11] K. J. Åstrom and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, illustrated edition edition, April 2008.

[12] D. R. Avresky, V. Shurbanov, R. Horst, W. Watson, L. Young, and D. Jewett. Performance Modeling of ServerNet SAN Topologies. *Journal of Supercomputing*, 14(1):19–37, July 1999.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 167–177. ACM Press, Octoboer 2003.

[14] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[15] R. V. Boppana and S. Chalasani. Fault-tolerant wormhole routing algorithms for mesh networks. *IEEE Transactions on Computers*, 44(7):848–864, 1995.

[16] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13:1465–1480, 1995.

[17] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[18] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, November 2009.

[19] Texas Advanced Computing Center. Stampede user guide. User Manual, January 2013. http://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide.

[20] Leibniz Supercomputing Centre. Supermuc petascale system. Press release, November 2012. http://www.lrz.de/services/compute/supermuc/systemdescription/.

[21] S. Chalasani and R. V. Boppana. Communication in Multicomputers with Nonconvex Faults. *IEEE Transactions on Computers*, 46(5):616–622, May 1997.

[22] C. Chaubal. The Architecture of VMware ESXi. In *White Paper*. VMware, Oct 2008.

[23] P. M. Chen and B. D. Noble. When Virtual Is Better Than Real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society.

[24] N. M. Chowdhury, Mosharaf K., and Raouf Boutaba. Network virtualization: state of the art and research challenges. *IEEE Commmunication Magazine*, 47(7):20–26, July 2009.

[25] N.M. Chowdhury, Mosharaf K., and Raouf Boutaba. A survey of network virtualization. *Computer Network*, 54(5):862–876, April 2010.

[26] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[27] Compaq, Intel and Microsoft. *Virtual Interface Architecture Specification*, 1.0 edition, December 1997.

[28] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[29] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[30] W. J. Dally and C. L. Seitz. The torus routing chip. *Journal of Parallel and Distributed Computing*, 1(4):187–196, 1986.

[31] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transaction on Computers*, C-36(5):547–543, May 1987.

[32] W. J. Dally and Brian Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2004.

[33] D. M. Dias and M. Kumar. Preventing congestion in multistage networks in the presence of hotspots. In *ICPP (1)*, pages 9–13, 1989.

[34] YZ. Dong, Y Chen, ZH Pan, JQ Dai, and YH Jiang. ReNIC: Architectural extension to SR-IOV I/O virtualization for efficient replication. *ACM Transaction on Architecture and Code Optimization (TACO)*, 8(4):40:1–40:22, January 2012.

[35] 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*, 16:412–427, 2005.

[36] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks An Engineering Approach*. Morgan Kaufmann, revised edition edition, 2003.

[37] E. N. Elnozahy and J. S. Plank. Checkpointing for Peta-Scale Systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable Secure Computing*, 1(2):97–108, April 2004.

[38] E. N. (Mootaz) Elnozahy, L. Alvisi, Y-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computer Survey*, 34(3):375–408, September 2002.

[39] M. Epperson, J. Naegle, J. Schutt, M. Bohnsack, S. Monk, M. Rajan, and D. Doerfler. A 3D Torus IB Interconnect on Red Sky. In *Open Fabrics Workshop 2010*, March 2010.

[40] P. Fabian, J. Palmer, J. Richardson, M. Bowman, P. Brett, R. Knauerhase, J. Sedayao, J. Vicente, C-C. Koh, and S. Rungta. Virtualization in the Enterprise. *Intel Technology Journal*, 10(3):227–242, aug 2006.

[41] K. Ferreira, J. Stearley, James H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Supercomputing (SC), pages 44:1–44:12, New York, NY, USA, 2011. ACM.

[42] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, and K. Ferreira. Poster: detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the 2011 Companion on High Performance Computing Networking, Storage and Analysis Companion*, Supercomputing (SC) Companion, pages 47–48, New York, NY, USA, 2011. ACM.

[43] 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. *IEEE Transactions on Parallel Distributed Systems*, 23(3):405–425, 2012.

[44] S. Floyd. TCP and explicit congestion notification. *SIGCOMM Computer Communication Review*, 24(5):8–23, October 1994.

[45] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, August 1993.

[46] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[47] M Galles. Spider: A high-speed network interconnect. *IEEE Micro*, 17(1):34–39, January 1997.

[48] Q. Gao, W. Huang, M. J. Koop, and D. K. Panda. Group-based coordinated checkpointing for mpi: A case study on infiniband. In *Proceedings of the 2007 International Conference on Parallel Processing*, ICPP '07, pages 47–, Washington, DC, USA, 2007. IEEE Computer Society.

[49] R. P. Golberg. Architectural Principles for Virtual Computer Systems. *Ph.D Thesis Div. Engineering and Applied Physics Harvard University*, February 1973.

[50] N. Golmie, Y. Saintillan, and D. Su. ABR switch mechanisms: design issues and performance evaluation. *Computer Networks and ISDN Systems*, 30(19):1749–1761, October 1998.

[51] C Gomez, F. Gilabert, M.E Gomez, P Lopez, and J. Duato. Deterministic versus adaptive routing in fat-trees. In Dhabaleswar K. (DK) Panda, editor, *IEEE International Parallel and Distributed Processing Symposium*, pages 292–300. IEEE Computer Society, 2007.

[52] 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, 2006.

[53] 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 Cynthia Phillips, editor, *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010.

[54] E. G. Gran and S.-A. Reinemo. Infiniband congestion control: modelling and validation. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools, pages 390–397, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[55] E. G. Gran, S.-A. Reinemo, O. Lysne, T. Skeie, E. Zahavi, and G. Shainer. Exploring the scope of the infiniband congestion control mechanism. In Bob Werner, editor, *2012 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE Computer Society, 2012.

[56] E. G. Gran, E. Zahavi, S.-A. Reinemo, T. Skeie, G. Shainer, and O. Lysne. On the relation between congestion control, switch arbitration and fairness. In Rajkumar Buyya, editor, *11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2011)*, pages 342 – 351. IEEE, May 2011.

[57] W. L Guay, B. Bogdanski, S.-A. Reinemo, O. Lysne, and T. Skeie. vftree - a fat-tree routing algorithm using virtual lanes to alleviate congestion. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 197–208, Washington, DC, USA, 2011. IEEE Computer Society.

[58] W. L. Guay and S.-A. Reinemo. A scalable method for signalling dynamic reconfiguration events with opensm. In Rajkumar Buyya, editor, *11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, pages 332 – 341. IEEE Computer Society Press, 2011.

[59] W. L. Guay, S.-A. Reinemo, B. D. Johnsen, and T. Skeie. A scalable signalling mechanism for VM migration with SR-IOV over infiniband. In *18th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 384–291. IEEE Computer Society, 2012.

[60] W. L. Guay, S.-A. Reinemo, O. Lysne, and T. Skeie. dftree: a fat-tree routing algorithm using dynamic allocation of virtual lanes to alleviate congestion in infiniband networks. In *Proceedings of the first international workshop on Network-aware data management*, NDM '11, pages 1–10, New York, NY, USA, 2011. ACM.

[61] 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 (CLUSTER)*, pages 126–135, 2010.

[62] I. Habib. Virtualization with kvm. *Linux Journal*, 2008(166), February 2008.

[63] C-T. Ho and L. J. Stockmeyer. A new approach to fault-tolerant wormhole routing for mesh-connected parallel computers. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 20–, Washington, DC, USA, 2002. IEEE Computer Society.

[64] H. J. Holz, A. Applin, B. Haberman, D. Joyce, H. Purchase, and C. Reed. Research methods in computing: what are they, and how should we teach them? *SIGCSE Bulletin*, 38(4):96–114, June 2006.

[65] W. Huang, J. Liu, M. Koop, B. Abali, and D.K Panda. Nomad: migrating OS-bypass networks in virtual machines. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 158–168, New York, NY, USA, 2007. ACM.

[66] X-Y. Lin; Y-C. Chung; T-Y. Huang. A multiple lid routing scheme for fat-tree-based infiniband networks. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 11–, 26-30 April 2004.

[67] J. Hursey, T. I. Mattox, and A. Lumsdaine. Interconnect agnostic checkpoint/restart in open mpi. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 49–58, New York, NY, USA, 2009. ACM.

[68] IBM. 20 Petaflop Sequoia Supercomputer. Press release, February 2009. `http://www-304.ibm.com/jct03004c/press/us/en/pressrelease/26599.wss`.

[69] IEEE 802 LAN/MAN Standards Committee. *Data Center Bridging standard*, IEEE 802.1Qau edition.

[70] Cisco Visual Networking Index(VNI). Hyperconnectivity and the Approaching Zettabyte Era. In *White Paper*, Jun 2010.

[71] InfiniBand Trade Association. *Infiniband architecture specification*, 1.2.1 edition, November 2007.

[72] V. Jacobson. Congestion avoidance and control. In *Symposium proceedings on Communications architectures and protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM.

[73] J. Jann, L. M. Browning, and R. S. Burugula. Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers. *IBM System Journal*, 42(1):29–37, January 2003.

[74] A. Kadav and M. M. Swift. Live migration of direct-access devices. *SIGOPS Operating Systems Review*, 43:95–104, July 2009.

[75] P. Kermani and L. Kleinrock. Virtual cut-through: a new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[76] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 126–137, New York, NY, USA, 2007. ACM.

[77] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. *SIGARCH Computer Architecture News*, 36(3):77–88, June 2008.

[78] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. ExaScale Computing Study: Technology Challenges in Achieving ExaScale Systems. Technical report, DARPA IPTO, Air Force Research Labs, September 2008.

[79] C. P. Kruskal and M. Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Transactions on Computers*, 32(12):1091–1098, December 1983.

[80] Oak Ridge National Laboratory. Introducing Titan: Advancing the Era of Accelerated Computing. Press release, November 2012. `http://www.olcf.ornl.gov/titan/`.

[81] C. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong-chan, Shaw wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Journal of Parallel and Distributed Computing*, pages 272–285, 1992.

[82] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, C-34:892–901, 1985.

[83] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance VMM-bypass I/O in virtual machines. In *USENIX Annual Technical Conference*, Berkeley, CA, USA, 2006. USENIX Association.

[84] 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, 2006.

[85] O. Lysne and T. Waadel. One-fault tolerance and beyond in wormhole routed meshes. *Microprocessors and Microsystems*, 21:471–481, 1998.

[86] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, The Mckinsey Global Institue, May 2011.

[87] D. May and P. Thompson. Transputers and routers: components for concurrent machines. In *Proceedings of the 13th Occam user group technical meeting on Real-time systems with transputers*, OUG-13, pages 215–231, Amsterdam, The Netherlands, The Netherlands, 1990. IOS Press.

[88] J. Montañana, J. Flich, A. Robles, and J. Duato. *High-Performance Computing*, volume 4759 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[89] Ohio State University Network-Based Computing Laboratory.

[90] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the impact of checkpoints on next-generation systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, MSST '07, pages 30–46, Washington, DC, USA, 2007. IEEE Computer Society.

[91] Christina Parsa and Jj Garcia-Luna-Aceves. Improving tcp congestion control over internets with heterogeneous transmission media. In *Proceedings of the Seventh Annual International Conference on Network Protocols*, ICNP '99, pages 213–, Washington, DC, USA, 1999. IEEE Computer Society.

[92] PCI-SIG. *Single Root I/O Virtualization And Sharing Specification Revision 1.1*, 1.1 edition, January 2010.

[93] Fabrizio Petrini and Marco Vanneschi. K-ary n-trees: High performance networks for massively parallel architectures. Technical report, University of Pisa, 1995.

[94] G. F. Pfister and V. A. Norton. ̈Hot Spot C̈ontention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, 34(10):943–948, 1985.

[95] G. F. Pfister and V. A. Norton. Interconnection networks for high-performance parallel computers. chapter ̈Hot spot c̈ontention and combining in multistage interconnection networks, pages 276–281. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[96] T. Pinkston, R. Pang, and J. Duato. Deadlock-free dynamic reconfiguration schemes for increased network dependeability. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):780–794, August 2003.

[97] D. K. Pradhan, editor. *Fault-tolerant computer system design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[98] E. Pugh. *IBM's 360 and early 370 systems.* MIT Press, Cambridge, Mass, 1991.

[99] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. Technical report, United States, 2001.

[100] K. K. Ramakrishnan and Raj Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Trans. Comput. Syst.*, 8(2):158–181, May 1990.

[101] V. Ramesh, R. L. Glass, and I. Vessey. Research in computer science: an empirical study. *Journal of Systems Software*, 70(1-2):165–176, February 2004.

[102] S.-A. Reinemo, T. Skeie, and M. K. Wadekar. Ethernet for high-performance data centers: On the new ieee datacenter bridging standards. *IEEE Micro*, 30:42–51, 2010.

[103] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1), 2007.

[104] F. O. Sem-Jacobsen and O. Lysne. Topology Agnostic Dynamic Quick Reconfiguration for Large-Scale Interconnection Networks. In *Proceedings of The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 228–235, May 2012.

[105] F. O. Sem-Jacobsen, T. Skeie, and O. Lysne. A dynamic fault-torlerant routing algorithm for fat-trees. In Hamid R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, June 27-30*, pages 318–324. CSREA Press, 2005.

[106] 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.

[107] T. Skeie, O. Lysne, J. Flich, P. Lopez, A. Robles, and J. Duato. LASH-TOR: A Generic Transition-Oriented Routing Algorithm. In *Proceedings*

of the IEEE International Conference on Parallel and Distributed Systems (ICPADS), pages 595–604. IEEE Computer Society Press, 2004.

[108] T. Skeie, O. Lysne, and I. Theiss. Layered shortest path (LASH) routing in irregular system area networks. In *Proceedings of Communication Architecture for Clusters*, 2002.

[109] J. E. Smith and R. Nair. The Architecture of Virtual Machines. *Computer*, 38(5):32–38, May 2005.

[110] H. Subramoni, P. Lai, M. Luo, and D. K. Panda. Rdma over ethernet - a preliminary study. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*, pages 1–9. IEEE, 2009.

[111] R. Vaidyanathan and J.L. Trahan. *Dynamic Reconfiguration: Architectures and Algorithms*. Series in Computer Science Series. Kluwer Academic Pub, 2003.

[112] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[113] A. Velte and T. Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[114] VMware. NSI DoubleTake and VMware ESX Server & GSX Server Virtual Machines. VMware Technical paper, Oct 2005 2007. `http://www.vmware.com/resources/techresources/236`.

[115] J. Wu. A Fault-Tolerant and Deadlock-Free Routing Protocol in 2D Meshes Based on Odd-Even Turn Model. *IEEE Transactions on Computers*, 52(9):1154–1169, 2003.

[116] M. Xie, Y. Lu, K. Wang, L. Liu, H. Cao, and X. Yang. Tianhe-1A Interconnect and Message-Passing Services. *IEEE Micro*, 32(1):8–20, 2012.

[117] E. Zahavi et al. Optimized InfiniBand TM fat-tree routing for shift all-to-all communication patterns. *Concurrency and Computation: Practice and Experience*, 22(2):217–231, 2009.

[118] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with Pass-through Device for Linux VM. In *Ottawa Linux Symposium (OLS)*, pages 261–268, 2008.

List of appendices

Paper I: W. L. Guay, Sven-A. Reinemo, O. Lysne, T. Skeie, B. D. Johnsen, and L. Holen. Host Side Dynamic Reconfiguration with InfiniBand. In Proc. 2010 IEEE International Conference on Cluster Computing (CLUSTER 2010), pages. 126-135, IEEE Computer Society, 2010.

Paper II: W. L. Guay and Sven-A. Reinemo. A Scalable Method for Signalling Dynamic Reconfiguration Events with OpenSM. In Proc. 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011). 332-341, IEEE Computer Society, 2011.

Paper III: W. L. Guay, B. Bogdanski, Sven-A. Reinemo, O. Lysne, and T. Skeie. vFtree - A Fat-tree Routing Algorithm using Virtual Lanes to Alleviate Congestion. In Proc. 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011). 197-208, IEEE Computer Society, 2011.

Paper IV: W. L. Guay, Sven-A. Reinemo, O. Lysne, and T. Skeie. dFtree - A Fat-tree Routing Algorithm using Dynamic Allocation of Virtual Lanes to Alleviate Congestion in InfiniBand Networks. In Proc. 1st ACM/IEEE Network-Aware Data Management (NDM) Workshop in conjunction with SC'11, 1-10, ACM, 2011 (*Best Paper Award*).

Paper V: W. L. Guay, Sven-A. Reinemo, B. D. Johnsen, Chien-H. Yen, T. Skeie, O. Lysne and O. Torudbakken. Early Experiences with Live Migration of SR-IOV enabled InfiniBand. Submitted to International journal.

Paper VI: W. L. Guay, Sven-A. Reinemo, B. D. Johnsen, T. Skeie and O. Torudbakken. A Scalable Signalling Mechanism for VM Migration with SR-IOV over InfiniBand. In Proc. 18th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2012.

Paper I

Host Side Dynamic Reconfiguration with InfiniBand

Wei Lin Guay, Sven-Arne Reinemo, Olav Lysne, Tor Skeie,
Bjørn Dag Johnsen and Line Holen

2010 IEEE International Conference on Cluster Computing

Host Side Dynamic Reconfiguration with InfiniBand

Wei Lin Guay, Sven-Arne Reinemo *Member, IEEE*, Olav Lysne *Member, IEEE*, Tor Skeie
weilin, svenar, olavly, tskeie @simula.no
Simula Research Laboratory,
Martin Linges vei 17, Fornebu,
N-1325 Lysaker, Norway
Phone: +4767828200 Fax: +4767828201
and
Bjørn Dag Johnsen, Line Holen
bjorn-dag.johnsen, line.holen @sun.com
Sun Microsystems

***Abstract*—Rerouting around faulty components and migration of jobs both require reconfiguration of data structures in the Queue Pairs residing in the hosts on an InfiniBand cluster. In this paper we report an implementation of dynamic reconfiguration of such host side data-structures. Our implementation preserves the Queue Pairs, and lets the application run without being interrupted. With this implementation, we demonstrate a complete solution to fault tolerance in an InfiniBand network, where dynamic network reconfiguration to a topology-agnostic routing function is used to avoid malfunctioning components. This solution is in principle able to let applications run uninterruptedly on the cluster, as long as the topology is physically connected. Through measurements on our test-cluster we show that the increased cost of our method in setup latency is negligible, and that there is only a minor reduction in throughput during reconfiguration.**

I. Introduction

The quest for ever increasing computing power drives the development of compute clusters to larger and larger scale. At the time of writing, the biggest InfiniBand installation has over 100K processors, and there are almost twenty sites with more than 10K processors reported on the Top 500 list [1]. This has challenged the interconnection networks of the systems with regards to mean time between component failure. It has to be an operating assumption for machines of these sizes that components in the interconnect fail while applications are running.

Although the importance of fault tolerant interconnection networks has only recently been acknowledged by industry, it has been a pet subject for researchers in academia for many years. The proposed solutions have generally had the aims of saving the application that was running at the instance of time when the component failed, and to increase the total uptime of the system.

One body of work has assumed that there is a checkpointing mechanism in the application. This means that when there is a fault-situation, the application can be halted, a new routing structure that avoids the faulty component can be implemented in the network while it is empty of traffic, and finally the application can be restarted from the last checkpoint. In this category we find [2] where faulty components are avoided through indirect routing. A number of routing algorithms that are not limited to any specific topology, and thereby is able to handle any topology change resulting from faults are also motivated from this mode of operation [3][4][5][6].

This work is in part financed by Sun Microsystems, Inc.

The techniques described and implemented in this paper do not assume any checkpointing mechanism. Rather, it aims at reacting so fast that the applications running on the cluster can continue uninterrupted. We call such methods *application transparent*. Application transparency has also been the assumption in most of the academic work in the field. An example is Boppana and Chalasani who defined a protocol for deadlock free rerouting around faulty regions in meshes [7][8], another is the proposal by Montañana et al. for torus networks [9]. Other examples focus on fault tolerance in variants of multistage networks [10], a topic that has become the focus of a renewed interest in the context of Ethernet-based data centre networks [11][12][13]. A solution that is independent of topology and routing functions is described in [14].

The main weakness of the above approaches for application transparent fault-tolerance is that they are highly inflexible. They either work for a limited set of topologies, or for a limited set of fault situations. This led to some efforts on *dynamic reconfiguration*, where the idea is to move from one routing function to another while the system is up and running [15][16][17][18][19]. The difficulty has been to do this in a way that does not create deadlocks in the transition phase [20].

Industrial installations of large computers have so far mainly resorted to the checkpoint/restart approach. This is, however, only a viable solution as long as the mean time between failures is high enough for the application to be expected to make significant progress - at least to the next checkpoint - before the next fault occurs. Due to the size of recent installations, we have seen some implementations of application-preserving fault tolerance mechanisms in InfiniBand clusters [21][22].

In designing solutions for application transparent fault tolerance, there are two fundamental decisions that have to be made. One is whether the change of routing paths should be triggered in the network interface cards, or in the network switches. Another fundamental question is whether the alternative paths should be set up a priori, or whether a *network manager* should be involved in finding new paths

978-0-7695-4220-1/10 $26.00 © 2010 IEEE
DOI 10.1109/CLUSTER.2010.21

once the fault has been discovered. In [21] Vishnu et al reports an implementation that uses an InfiniBand feature called *Automatic Path Migration*. When a network interface card detects loss of connection, it automatically swaps to a backup path that can still be connected. This backup path is set up a-priori by using a duplicate set of addresses. The drawback of this approach is that it is not always possible to define two disjoint paths between every two points in the network. This problem is particularly severe for implementations based on -ary -cubes, like the recent Red Sky installation in Sandia National Lab [23], but it is also true for oversubscribed fat-trees [22].

In this paper we report a reconfiguration implementation that is based on a network (subnet) manager, that dynamically reroutes the network. Furthermore it is based on changing routing tables in the switches. In order to efficiently route the topology containing the fault, we use a topology agnostic routing algorithm called LASH in the experiments [24]. Our method is, however, independent of topology and routing algorithm. Combined with a topology agnostic routing algorithm, the method can in principle sustain connectivity of all connections, and thereby keep applications alive, as long as the topology is physically connected.

We provide details of the implementation of the method in an InfiniBand cluster. The fault detection, and the reconfiguration of the routing algorithm in the network is done through mechanisms in OpenSM. Leveraging this, we develop novel mechanisms at the host side, to allow the application to migrate its connections from the old routing structure to the new one. Furthermore we present measurements that demonstrate the performance of the method, as well as the deadlock problem that results from rerouting in a path set up a-priori.

The paper is structured as follows: In section II we give some background of the InfiniBand technology, before we give some details on the implementation of the reconfiguration method in section III. Thereafter we present the setup of the experiments on an InfiniBand cluster in section IV and the results and analysis in section V. Finally we conclude in section VI.

II. The InfiniBand Architecture

The InfiniBand Architecture was first standardised in October 2000 [25], as a merge of two older technologies called Future I/O and Next Generation I/O. As with most other recent interconnection networks, InfiniBand (IB) is a serial point-to-point full-duplex technology. It is scalable beyond ten-thousand nodes with multiple CPU cores per node and efficient utilisation of host side processing resources. The current trend is that IB is replacing proprietary or low-performance solutions in the high performance computing domain, where high bandwidth and low latency are key requirements. The IB specification is an evolving standard that was last updated in 2007. Our proposal is in part based on features added in the latest version of the standard. The following sections give a brief overview of the relevant management and communication properties in IB.

A. Subnet Management

InfiniBand networks are referred to as subnets, where a subnet consists of a set of hosts interconnected using switches and point to point links. A single subnet is scalable to more than ten-thousand nodes and two or more subnets can be interconnected using an IB router. Hosts and switches within a subnet are addressed using local identifiers (LIDs). The LID is a 16 bit value where the first 48151 values are reserved for unicast addresses and the rest is reserved for multicast.

An IB subnet requires at least one subnet manager (SM), which is responsible for initialising and bringing up the network, including the configuration of all the switches, routers and host channel adaptors (HCAs) in the subnet. At the time of initialisation the SM starts in the *discovering state* where it does a heavy sweep of the network in order to discover all switches and hosts. During this phase it will also discover any other SMs present and negotiate who should be the master SM. When this phase is complete the SM enters the *master state*. In this state it proceeds with LID assignment, switch configuration, routing table calculations, and port configuration. If successful it enters the *subnet up state* and the subnet is ready for use.

During normal operation the SM performs periodic *light sweeps* of the network to check for topology changes. If a change is discovered during a light sweep or if a message (trap) signalling a network change is received by the SM it will reconfigure the network according to the changes discovered. The reconfiguration includes the steps used during initialisation. Whenever the network changes (e.g. a link goes down, a device is added, or a link is removed) the SM must reconfigure the network accordingly. A major part of the reconfiguration process is the rerouting of the network which must be performed in order to guarantee full connectivity, deadlock freedom, and proper load balancing between all source and destination pairs.

B. Transport Services and Queue Pairs

InfiniBand supports a rich set of *transport services* in order to provide both Remote Direct Memory Access (RDMA) and traditional Send/Receive semantics. The available transport services are Reliable Connection (RC), Unreliable Connection, Reliable Datagram, Unreliable Datagram, and Raw Datagram. The reliable services guarantee in order delivery, correctness, and acknowledgements. All our results in this paper are based on the RC service, but is valid for the other services as well. For more details about the various services please refer to the IB specification [25].

The available transport services support various *transport functions*. For a RC the most common are the RDMA Read/Write and the Send/Receive operations. The RDMA Read/Write operation allows a node to Read/Write the virtual address space of a remote note without involving the host CPU. The Send/Receive operations is much like Channel I/O where data can be sent to a remote host, but it is up to the receiver to handle the data properly. In our experiments we have used the Send/Receive operations over a RC.

Independent of the transport service used all InfiniBand HCAs communicate using Queue Pairs (QPs). A QP is created during the communication setup, and a set of initial attributes such as QP number, HCA port, destination LID, queue sizes, and transport service are supplied. When communication is over the QP is destroyed. An HCA can handle many QPs, each QP consists of a pair of queues, a Send Queue (SQ) and a Receive Queue (RQ), and there is one such pair present at each end-node participating in the communication. The send queue holds work requests to be transferred to the remote node, while the receive queue holds information on what to do with the data received from the remote node.

In addition to the QPs each HCA has one or more Completion Queues (CQs) that are associated to a set of send and receive queues. The CQ holds completion notifications for the work requests posted to the send and receive queue.

C. Traps, Notices, and Event Forwarding

In InfiniBand the SM is responsible for monitoring the network for changes. This is done using Subnet Management Agents (SMAs) that are present in every switch and every HCA. The SMAs communicate any changes, e.g. new connections, disconnections, and port state changes, to the SM using *traps* and *notices*. A trap is a message sent to alert about a certain event, and it contains a notice attribute with the details about the event. Different traps are defined for different events, e.g. the *port state changed* trap (trap number 128) is sent from a switch to the SM whenever there is a change in the port state on one of the switch ports. I.e. a port has been connected or disconnected. If the SM receives trap number 128 with a notice indicating that a switch port has lost the connection to an end-node, the SM will generate an *out of service* trap (trap number 65) which indicates that the given end-node is unavailable. This trap is sent from the SM to all end-nodes that has subscribed to this trap.

In order to reduce unnecessary distribution of traps IB applies an *event forwarding* mechanism where end-nodes are required to explicitly subscribe to the traps they want to be informed about. E.g. the trap in the previous example would only be sent if the end-node had subscribed to trap number 65.

III. Reconfiguration Method

Our application transparent reconfiguration method consists of a *network reconfiguration* phase, a *signalling* phase, and a *host reconfiguration* phase as illustrated in Figure 1. After the SM has initialised and brought up the subnet, hosts will subscribe for any events that they would like to be notified about and the subnet will enter normal operation. If a change occurs in the network this is discovered by the SM and triggers the network reconfiguration phase. When the network configuration is complete and new routing tables have been distributed, it enters the *signalling* phase where the *repath* trap is forwarded to the hosts that are subscribing to this event. At the host, the reception of the *repath* trap triggers the *host reconfiguration* phase.

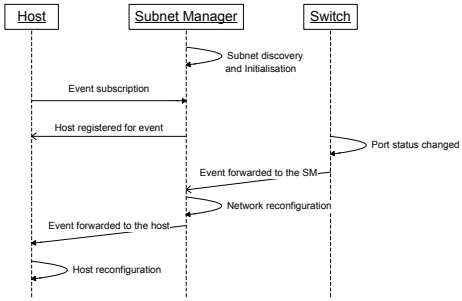


Fig. 1. A sequence diagram showing the interaction between the host and the SM relevant for reconfiguration.

Our proposal is based on the established methods for detecting and signalling network changes in IB, but we have added support for the necessary traps and notices required for our reconfiguration method. In the following section we give more details about each phase and its implementation.

A. Network Reconfiguration

Before a host can be reconfigured two things must happen, the change must be detected and the network must be reconfigured. Only when we have a new valid configuration for the network is it possible to reconfigure the hosts. We will not address the network reconfiguration in this paper, but our implementation is based on the existing reconfiguration mechanism in OpenSM.

During normal operation OpenSM performs periodic *light sweeps* of the network to check for topology changes as described in section II-A. If a change is discovered either during a light sweep or through the reception of trap 128 (port state changed) or trap 144 (change capability mask) a network reconfiguration is triggered. A major part of the reconfiguration process is the rerouting of the network which must be performed in order to guarantee full connectivity, deadlock freedom, and proper load balancing between all source and destination pairs. When the network reconfiguration is complete OpenSM enters the signalling phase described in the next section.

The current implementation in OpenSM is very coarse in the way that it can only tell when rerouting is started and when it is finished. OpenSM does not maintain information about what paths have been changed. The current functionality is, however, adequate to demonstrate the concept of host side dynamic reconfiguration.

B. Signalling

When OpenSM has reconfigured the network the hosts must be notified about the changes, but this is not supported by the current version of OpenSM. Our proposed way of doing this

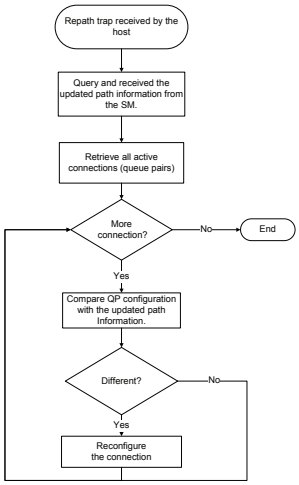


Fig. 2. The flow chart of the host reconfiguration.

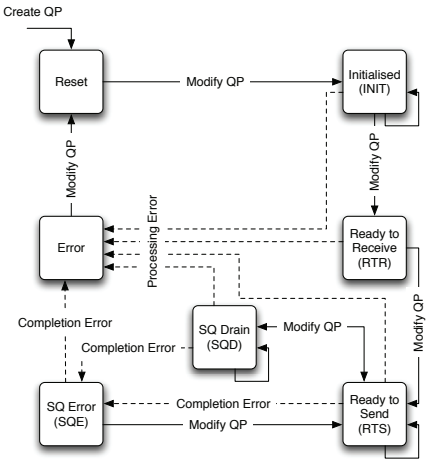


Fig. 3. The QP state diagram.

follows the general trapping concept in IB, using traps, notices, and event forwarding as described in section II-C. The basic idea is that OpenSM will generate an *unpath* trap whenever it invalidates a path and generate a *repath* trap when the path is recreated. This requires that the end nodes subscribe to these traps at network initialisation time.

Owing to the current implementation of OpenSM lacking functionality for identifing individual path changes, our implementation is further reduced to only sending the *repath* trap whenever the SM has completed a rerouting phase. I.e. the *repath* trap tells the hosts that the network has been reconfigured and the host should check if it needs to update any of its QPs.

For the actual implementation we have used the *repath* trap, introduced in the 1.2.1 release of the IB specification, to signal the nodes. Support for generating and subscribing to this trap has been added to OpenSM and to the IB stack at the host.

C. Host Reconfiguration

For host side dynamic reconfiguration the QP and the address handle is the host-side entity that must be reconfigured whenever the subnet configuration changes. The series of events that is part of the host reconfiguration process is illustrated in Figure 2.

Whenever the host receives a *repath* trap it knows that the network is reconfigured and will query the SM for the updated path record. When the host receives the updated path record it must cycle through all its active QPs and compare the updated path record with the current path properties. If there are any differences the QP must be reconfigured. There are many QP attributes that can be changed during host reconfiguration such as link width, maximum transfer unit, and service level. What attributes to update depends on the properties of the routing algorithm and on the usage scenario, in our prototype we will update the service level assigned to a given source,destination pair by the routing algorithm in order to avoid deadlock after rerouting. But it could be used to update any QP attributes.

For the actual implementation we have added trap handling code for the *repath* trap and a mechanism to keep track of the active QPs within a HCA. In our proposal, all active QPs are stored in a linked list when they are created and removed from this list when they are destroyed.

When it comes to the actual reconfiguration of the QP path information (remote address vector) there are several ways to achieve this with various limitations and impact on upper layer protocols. How we are allowed to do this in a compliant manner is limited by the QP state diagram show in Figure 3. In addition to the transitions shown all states are allowed to move directly to the reset state.

1) Reset Queue Pair: When a QP is created it goes through the RESET-INIT-RTR-RTS state sequence show in Figure 3. It is only when the QP has reached the RTS state that it is considered fully operational, and it cannot be changed when in this state (except when using APM as discussed later). If an error occurs the QP is moved to the SQE or ERROR state and communication is disrupted. Considering this, one obvious way to reconfigure the QP would be to recreate the QP, i.e. repeat the full reset cycle RESET-INIT-RTR-RTS which can be done at any time by doing the RTS-RESET transition. This approach has two drawbacks: First, all existing QPs need to be stored before the QP enters the RESET state. Only then can the remote address vector of a QP be updated. Second, when a QP is moved to the RESET state the SQ and CQ will be cleared and loss of data might happen. As this loss is not handled by the QP it is up to the upper layer protocols to

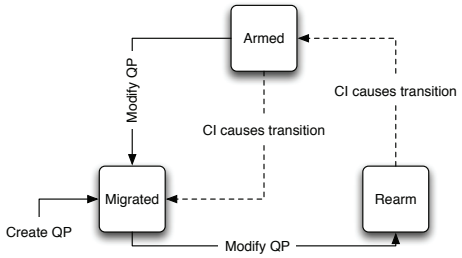


Fig. 4. The APM state diagram.

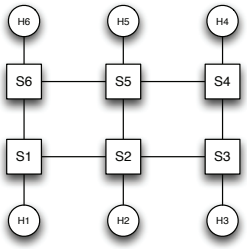


Fig. 5. The test-bed with a 3x2 mesh that is reduced to a 6 node ring when the link between switch S2 and S5 fails.

handle it. Thus, it is not possible to perform an application transparent host reconfiguration using this approach.

2) Send Queue Drain: A second alternative is to move the QP into the SQD state, with the assumption that the remaining elements in the SQ are able to be executed successfully before the state transition. Then, when in the SQD state, the host can be reconfigured with the updated path information before returning to the RTS state (see Figure 3). This approach has two drawbacks: First, this mechanism only works if the transition from RTS to SQD is a success, i.e. that we are able to drain the SQ. The SQD may not succeed if there are outstanding operations and the current path is not operational, hence this would not be a generic solution in the first place and would need a mode to avoid the QP going to the full error state when unsuccessful. Second, the SQD state is currently not supported by the Mellanox ConnectX firmware, so this is not a viable solution with current hardware.

3) Automatic Path Migration: A third alternative, and the one that we have used in this paper, is to use the automatic path migration (APM) mechanism provided by IB. Automatic path migration will automatically switch from a primary path to an alternative path when a fault happens without destroying the established connection. It is designed to support fast switching to a precalculated backup path in the case of failure. For dynamic host reconfiguration we will use APM purely as a mechanism to switch the remote address vector manually, i.e. we will not use a precalculated path but reconfigure a path based on network reconfiguration.

The APM state diagram in shown in Figure 4 and it must be seen in relation to the QP state diagram in Figure 3. For our proposal the most important property is that when the QP is in the RTS state it is also allowed to be in any of the APM states. I.e. the QP can make any of the transitions in the APM state diagram without disrupting the connection. This makes it possible to modify the QP while in the RTS state.

In order to modify the QP we first make a manual transition from the MIGRATED to the REARMED state (Figure 4). Then the address vector of the alternative path is updated with the new path information. Finally, the alternative path is switched over to become the primary path and the QP state is manually switched back to the MIGRATED state (REARMED-ARMED-MIGRATED). With this approach we are able to dynamically reconfigure the host in an application transparent manner without disrupting the connection.

IV. Experiment Setup

In this section, we present the hardware and software used in our test bed and we describe the network configuration and the imitated fault that is used in our test case.

A. Experimental Test Bed

Our test bed consists of six nodes and six switches. Each node consists of a Sun Fire X2200 M2 server [26] that has a dual port Mellanox ConnectX DDR HCA with an 8x PCIe 1.1 interface, one dual core AMD Opteron 2210 CPU, and 2GB of RAM. The switches consist of four 24-port Infiniscale-III [27] switches and two 36-port Infiniscale-IV [28] switches that is used to construct the topology illustrated in Figure 5. All the hosts are installed with Ubuntu Linux 8.04 x86_64 kernel version 2.6.24-24-generic and a modified version of OFED 1.4.1 that contains the dynamic reconfiguration prototype. Some changes have been made to *Perftest* [29] in order to support regular bandwidth reporting and continuously sending traffic at full link capacity. The modified *Perftest* is used to generate the ping pong traffic pattern shown in Table I. A minor modification to the *MVAPICH2* [30] library has been made as well, where we commented out their APM handler as it conflicts with our implementation.

B. Network Configuration

With respect to -ary -cube topologies, it is very difficult to predict and predefine an alternative path for every fault. We have therefore selected a simple 3x2 mesh together with the LASH routing algorithm in our experiment to study this scenario.

The LASH [24] routing algorithm is a topology agnostic routing engine that is available in OpenSM. It uses virtual lanes as deadlock avoidance mechanism and a one-to-one mapping between the service level and virtual lane. For each source, destination pair, LASH assigns the shortest path together with a service level to be used that guarantee deadlock freedom. The LASH routing engine is responsible

Src	Dst
H6	H3
H5	H2
H4	H1
H3	H6
H2	H5
H1	H4

TABLE I
THE SOURCE, DESTINATION PAIRS USED IN OUR SYNTHETIC TRAFFIC PATTERN.

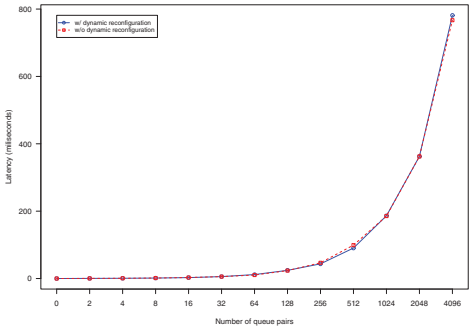


Fig. 6. The observed latency of ib_create_qp with and without the reconfiguration code.

for generating a deadlock free routing table whenever there is a change in the subnet. However, the number of virtual lanes required for deadlock avoidance may vary after the rerouting since the required number of virtual lanes depends on the underlying topology.

Using regular APM it is possible to predefine a deadlock free alternative path for the link between S1 and S6 or S3 and S4 (Figure 5), but a backup for the path between S2 and S5, is hard to define in advance because the subnet has turned into a ring topology that requires two virtual lanes for deadlock avoidance. Neither the route through S1-S6 nor S3-S4 with VL 0 only are valid as the shortest path alternative because both will form a cyclic channel dependency. This problem will increase in complexity when the topology size and number of faults increase.

C. Imitated Fault

The majority of faults in an IB subnet is caused by resource failures such as a link or switch malfunction. In our experiments we have imitated link faults by manually disconnecting the link using *ibportstate*. When a fault happens, the worst case effect of rerouting *without* host reconfiguration is the occurrence of a deadlock in the subnet due to the use of old paths. The deadlock happens because packets are unable to make progress since they are mutually waiting for the release of resources, such as buffers and channels. Without host side reconfiguration, terminating and restarting the running

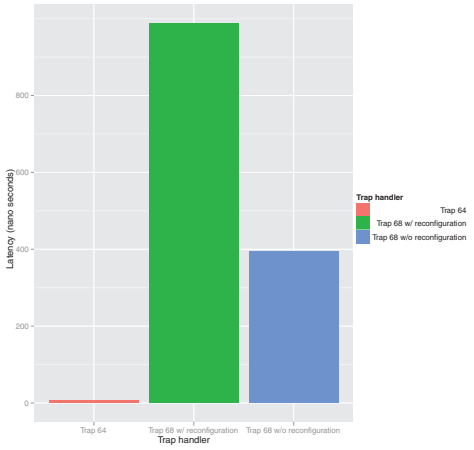


Fig. 7. The observed latency for trap 64 and trap 68 with and without the reconfiguration code.

application is the only solution to update the QP and break the cyclic dependencies.

V. Performance Evaluation

We have carried out three types of experiments to evaluate the performance of our host side dynamic reconfiguration implementation. First, we use a set of micro benchmarks to measure the impact our implementation has on the latency introduced during the setup phase and the trap handling phase. Second, we study the application transparency of our solution using synthetic traffic patterns by comparing the results from our proposed solution and the conventional implementation. Third, we analyse the application level impact of our solution using the *HPC Challenge* (HPCC) benchmark [31] and NAS Parallel Benchmark [32].

A. Queue Pair Setup Latency

With this micro benchmark we measure the additional latency that has been introduced during the *ib_create_qp*. In our implementation all the active queue pairs are stored in a linked list when it is created and removed from the linked list when it is destroyed. To simulate a large clusters where there might be many active QPs at each host we have designed a test to create an increasing number of QPs and calculate the time consumed. The results are presented in Figure 6 and shows that our implementation has no measurable impact on the initialisation time.

B. Reconfiguration Latency

In Figure 7 we compare the total latency between the simple *in service* trap handler and *repath* trap handler. The latency

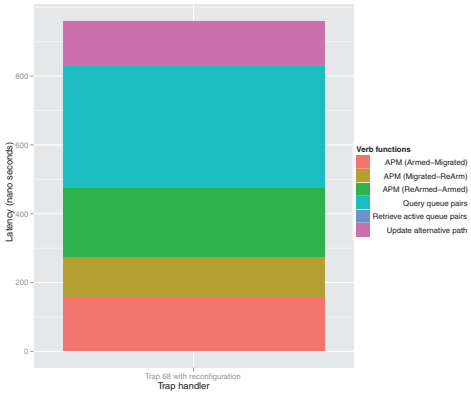


Fig. 8. The host side reconfiguration latency broken down into the processes that contribute to the increased latency.

measurements are based on time stamp dumps added in the kernel and it is only based on one QP per HCA.

For the *repath* trap we show both a scenario where reconfiguration is unnecessary and a scenario where reconfiguration is required. There are two reasons why the *repath* trap handler is so time consuming compared to the *in service* trap handler: First, the *repath* trap handler has to retrieve all the active QPs within a HCA. Second, it must query all the QPs to determine whether it must be reconfigured. The *in service* trap handler, however, only have to update the service state of the newly connected node when an *in service* trap is received.

Figure 8 shows the *repath* trap handling broken down into the different functions that contribute to the latency of the reconfiguration process. It shows that the majority of time is spent in the *ib_query_qp* function and on APM state migration. The *ib_query_qp* retrieves all the relevant info from the active queue pairs that are in the RTS state, which consists of all the data structures required for APM state transitions. The APM state migration occurs in the HCA firmware so the details about what happens there is not available.

In total a majority of the reconfiguration time is spent on the interaction between HCA firmware and hardware (e.g *ib_query_qp* function and APM state transitions). With only one QP this time consumption is negligible, but with many active QPs this might be a scalability problem.

C. Synthetic Traffic

In this experiment we measure the impact that host reconfiguration has on network performance. The experiment also demonstrate the application transparency of our solution and the negative effect of deadlock on network performance.

The experiment was performed by running the traffic pattern given in Table I between the hosts in the 3x2 mesh topology given in Figure 5. While the communication is active, the link between S2 and S5 in Figure 5 is disconnected manually to imitate a link failure. This causes the topology to change from a 3x2 mesh into a ring. When this happens, the switches will detect the failed link and trigger a *port state changed* trap that is sent to the SM. In response to this trap the SM performs a network reconfiguration. When the SM has completed the reconfiguration new deadlock free routing tables have been distributed to all the switches. A major change between the old and new routing tables is that the number of virtual lanes required for deadlock avoidance has increased from 1 to 2. Unfortunately, the QP connections that was established before the rerouting will not be aware of these changes and continue to use only virtual lane 0. If dynamic host reconfiguration is not enabled this will eventually lead to deadlock and a reduction of throughput to less than 1% of the available bandwidth as shown in the purple plot in Figure 9. If dynamic reconfiguration is enabled the loss of throughput is avoided as show in the blue in Figure 9. This shows that our proposal is able to reconfigure the host in an application transparent manner and with a limited loss of throughput. From the figure we observe that during reconfiguration there is a brief dip in throughput, but the hosts are quickly able to regain most of their bandwidth. They are not able to regain all throughput because now there is one less link active and less physical bandwidth in the network. The difference in bandwidth between the two topologies equals the difference between the orange and green plot in Figure 9. The orange line shows the bandwidth achieved for the 3x2 mesh using a single virtual lane, while the green line represents the total throughput in a ring topology with two virtual lanes.

D. Application Traffic

In the previous section, we presented the results from measurement based on the synthetic predefined traffic patterns. In this section we repeat the experiment with traffic generated by the HPCC *b_eff* benchmark with 12 processes. Even though the fault is still synthetically created by disconnecting the link manually, it resembles the network environment that the application would experience during a fault.

The results from the HPCC *b_eff* benchmark are shown in Table II. The second column contains the results for a run without faults and without any reconfiguration happening. The third column contains the results for a run with one fault (*link fails between S2 and S5 in Figure 5*) and one consecutive reconfiguration, and the fourth column contains the results for a run with two faults (*link fails between S2 and S5, and between S3 and S4 in Figure 5*) and two consecutive reconfigurations. If the benchmark is run without reconfiguration active the benchmark will not complete once the first fault happens due to deadlock as demonstrated in the previous section. From the results it is not the numbers themselves that are interesting, but the fact that with our host side dynamic reconfiguration the QPs always survive the reconfiguration even with the deadlock scenario. Moreover, it demonstrates that the reconfiguration is application transparent.

We have also performed some quick measurements with the NAS Parallel Benchmark to study the overhead of our reconfiguration mechanism with one fault. The results of the

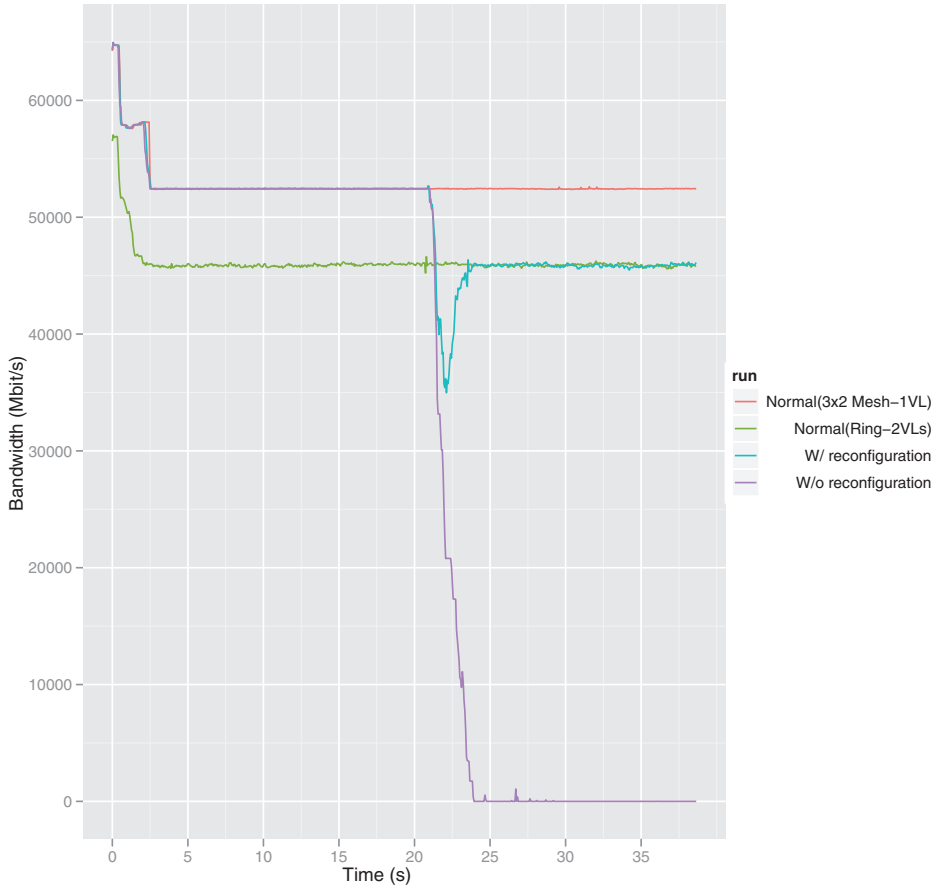


Fig. 9. The observed throughput with and without reconfiguration.

NAS Parallel Benchmark, IS, class B with 8 and 16 processes respectively and EP, class B with 12 processes are shown in Table III. The results show that the execution time increases by 79% with IS, Class B with 8 processes and by 36% with IS, Class B with 16 processes. However, with EP, Class B, 12 processes, the execution time overhead is just barely 0.57% compared with the original execution time. Hence, the results demonstrate that for an application running for a reasonably long period of time, the overhead caused by our host side reconfiguration is almost negligible. In the future, we plan to have a more thorough evaluation and performance analysis of our solution using the NAS Parallel Benchmark.

VI. Conclusions and Future Work

In this paper we have presented a novel method for host side dynamic reconfiguration for IB. Our proposed fault tolerance mechanism dynamically reconfigures the host side QP attributes on active connections based on feedback from the SM. We have evaluated a prototype of this mechanism on a small IB cluster and it is evident from our experiments with synthetic and application traffic that our approach is entirely transparent from the upper layer protocols. Furthermore, the hosts are able to retain network performance and deadlocks are avoided.

Future work includes studying this method in larger clusters in order to better evaluate performance and scalability. One

Network latency and throughput	a) Without faults	b) With one fault	c) With two faults
Min Ping Pong Lat. (ms)	0.001331	0.001431	0.001371
Avg Ping Pong Lat. (ms)	0.002119	0.002184	0.002273
Max Ping Pong Lat. (ms)	0.002454	0.002503	0.002742
Naturally Ordered Ring Lat. (ms)	0.002214	0.002384	0.002503
Randomly Ordered Ring Lat. (ms)	0.002309	0.002440	0.002514
Min Ping Pong BW (MB/s)	871.046	870.142	876.232
Avg Ping Pong BW (MB/s)	1522.620	1522.301	1522.655
Max Ping Pong BW (MB/s)	1591.162	1591.766	1591.162
Naturally Ordered Ring BW (MB/s)	455.061734	462.909141	476.388614
Randomly Ordered Ring BW (MB/s)	537.962774	555.397093	424.144639

TABLE II
RESULTS FROM THE HPC CHALLENGE BENCHMARK WITH ZERO, ONE AND TWO FAULTS/RECONFIGURATIONS.

Execution time of selected tests in NPB	a) Normal	b) One fault w/ reconf.
IS, Class B, 8 processes (s)	1.36	2.44
IS, Class B, 16 processes (s)	6.435	8.798
EP, Class B, 12 processes (s)	25.803	25.95

TABLE III
RESULTS FROM THE SELECTED NAS PARALLEL BENCHMARK TEST SUITE, WITH NORMAL AND ONE FAULT WITH RECONFIGURATION.

possible issue we foresee is the scalability problem when the subnet size grows and the number of active QPs increase, this will increase the latency of the host reconfiguration. In the current implementation the reconfiguration is based on the topology changes. However, a topology change does not necessarily mean path information change. Thus, we would like to further improve the granularity in which the SM reports network changes and in the way the routing algorithms reroutes the network in case of changes. We believe that this will improve the speed of the network reconfiguration in the SM, reduce the frequency of the event signalling, and reduce the time spent looking for unnecessary reconfiguration at the host side.

In the long term, we plan to further extend our work to support the capability of job migration where QPs can be dynamically reconfigured and moved between hosts without having to tear down and restart the running application. This is applicable and beneficial during clusters maintenance and for network virtualisation in large data centres.

REFERENCES

[1] Top 500 supercomputer sites. http://top500.org/, November 2009.

[2] María Engracia Gómez, Nils Agne Nordbotten, Jose Flich, Pedro López, Antonio Robles, José Duato, Tor Skeie, and Olav Lysne. A routing methodology for achieving fault tolerance in direct networks. *IEEE Trans. Computers*, 55(4):400–415, 2006.

[3] 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, 2006.

[4] T. Skeie, O. Lysne, J. Flich, P. Lopez, A. Robles, and J. Duato. Lashtor: A generic transition-oriented routing algorithm. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 595–604. IEEE Computer Society Press, 2004.

[5] 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.

[6] M.Koibuchi, A.Funahashi, A.Jourakua, , and H.Amano. L-turn routing: an adaptive routing in irregular networks. In *Proceedings of the 2001 International Conference on Parallel Processing*, pages 383–392. IEEE Computer Society, September 2001.

[7] R. V. Boppana and S. Chalasani. Fault-tolerant wormhole routing algorithms for mesh networks. *IEEE Transactions on Computers*, 44(7):848–864, 1995.

[8] Suresh Chalasani and Rajendra V. Boppana. Communication in multicomputers with nonconvex faults. *IEEE Transactions on Computers*, 46(5):616–622, May 1997.

[9] José Montañana, José Flich, Antonio Robles, and José Duato. *High-Performance Computing*, volume 4759 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] Youngsong Mun and Hee Yong Youn. On performance evaluation of fault-tolerant multistage interconnection networks. In *SAC '92: Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing*, pages 1–10. ACM Press, 1992.

[11] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. *SIGCOMM Comput. Commun. Rev.*, 39(4):51–62, 2009.

[12] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.*, 39(4):63–74, 2009.

[13] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. *SIGCOMM Comput. Commun. Rev.*, 39(4):39–50, 2009.

[14] Ingebjørg Theiss and Olav Lysne. Froots: A fault tolerant and topology-flexible routing technique. *IEEE Trans. Parallel Distrib. Syst.*, 17(10):1136–1150, 2006.

[15] N. Natchev, D. Avresky, and V. Shurbanov. Dynamic reconfiguration in high-speed computer clusters. In *Proceedings of the International Conference on Cluster Computing*, pages 380–387, Los Alamitos, 2001. IEEE Computer Society.

[16] 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.

[17] T. Pinkston, R. Pang, and J. Duato. Deadlock-free dynamic reconfiguration schemes for increased network dependeability. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):780–794, August 2003.

[18] Olav Lysne, Jose Miguel Montanana, Jose Flich, Jose Duato, Timothy Mark Pinkston, and Tor Skeie. An efficient and deadlock-free network reconfiguration protocol. *IEEE Transactions on Computers*, 57:762–779, 2008.

[19] J. M. Montanana, J. Flich, and J. Duato. Epoch-based reconfiguration: Fast, simple, and effective dynamic network reconfiguration. *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, April 2008.

[20] Jose Duato, Olav Lysne, Ruoming Pang, and Timothy M. Pinkston. Part

I: A theory for deadlock-free dynamic network reconfiguration. *IEEE Transactions on Parallel and Distributed Systems*, 16:412–427, 2005.

[21] Abhinav Vishnu, A.R. Mamidala, Sundeep Narravula, and D.K. Panda. Automatic path migration over infiniband: Early experiences. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8. IEEE, 2007.

[22] Abhinav Vishnu, Manojkumar Krishnan, and Dhabaleswar K. Panda. An efficient hardware-software approach to network fault tolerance with infiniband. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 1–9. IEEE, 2009.

[23] Sandia National Laboratories. Red sky. http://www.top500.org/system/10188.

[24] T. Skeie, O. Lysne, and I. Theiss. Layered shortest path (LASH) routing in irregular system area networks. In *Proceedings of Communication Architecture for Clusters*, 2002.

[25] InfiniBand Trade Association. *Infiniband architecture specification*, 1.2.1 edition, November 2007.

[26] Oracle. Sun fire x2200 m2 server. Press release, November 2008. http://www.sun.com/servers/x64/x2200/.

[27] Mellanox Technologies. Mellanox technologies announces 480 gb/sec single chip 3rd generation infiniband switch device. Press release, November 2003. http://www.mellanox.com/pdf/press_releases/pr_111003a.pdf.

[28] Mellanox Technologies. Mellanox infiniscale iv switch architecture provides massively scaleable 40gb/s server and storage connectivity. Press release, November 2007. http://www.mellanox.com/content/pages.php?pg=press_release_item&rec_id=34.

[29] Perftest - performance testing framework. http://perftest.sourceforge.net/, September 2009.

[30] Mvapich2 - mpi-2 over openfabrics-ib, openfabrics-iwarp, psm, udapl and tcp/ipe. http://http://mvapich.cse.ohio-state.edu///, February 2010.

[31] Hpc challenge benchmark. http://icl.cs.utk.edu/hpcc/.

[32] Nas parallel benchmark. http://www.nas.nasa.gov/Resources/Software/npb.html.

Paper II

A Scalable Method for Signalling Dynamic Reconfiguration Events with OpenSM

Wei Lin Guay and Sven-Arne Reinemo

2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

A Scalable Method for Signalling Dynamic Reconfiguration Events with OpenSM

Wei Lin Guay, Sven-Arne Reinemo
Simula Research Laboratory,
Lysaker, Norway
{weilin, svenar}@simula.no

***Abstract*—Rerouting around faulty components, on-the-fly policy changes, and migration of jobs all require reconfiguration of data structures in the Queue Pairs residing in the hosts on an InfiniBand cluster. In addition to a proper implementation at the host, the subnet manager needs to implement a scalable method for signaling reconfiguration events to the hosts. In this paper we propose and evaluate three different implementations for signalling dynamic reconfiguration events with OpenSM. Through our evaluation we demonstrate a scalable solution for signalling host-side reconfiguration events in an InfiniBand network based on an example where dynamic network reconfiguration combined with a topology-agnostic routing function is used to avoid malfunctioning components.**

Through measurements on our test-cluster and an analytical study we show that our best proposal reduces reconfiguration latency by more than 90% and in certain situations eliminates it completely. Furthermore, the processing overhead in the subnet manager is shown to be minimal.

I. Introduction

The interconnection network plays a critical role in the next generation of super computers, clusters and data centres. Major efforts are put into improving classical performance metrics such as latency and bandwidth, and more recently evolved metrics for utilisation and power efficiency. Furthermore, the importance of fault tolerant interconnection networks has been acknowledged by industry when moving towards exascale systems.

Fault tolerance has been a research topic in academia for many years. The proposed solutions have generally had the aims of saving the application that was running at the instance of time when the component failed, and to increase the total uptime of the system. One body of work has assumed that there is a checkpointing mechanism in the application. This means that when there is a fault-situation, the application can be halted, a new routing structure that avoids the faulty component can be implemented in the network while it is empty of traffic, and finally the application can be restarted from the last checkpoint. In this category we find [1] where faulty components are avoided through indirect routing. A number of routing algorithms that are not limited to any specific topology, and thereby is able to handle any topology change resulting from faults are also motivated from this mode of operation [2][3][4][5].

Other techniques do not assume any checkpointing mechanism, rather, they aim at reacting so fast that the applications running on the cluster can continue uninterrupted. We call such methods *application transparent*. Application transparency has also been the assumption in most of the academic work in this field. An example is Boppana and Chalasani who defined a protocol for deadlock free rerouting around faulty regions in meshes [6][7] while another is the proposal by Montañana et al. for torus networks [8]. Other examples focus on fault tolerance in variants of multistage networks [9], a topic that has become the focus of a renewed interest in the context of Ethernet-based data centre networks [10][11][12].

The main weakness of the above approaches for application transparent fault-tolerance is that they are highly inflexible. They either work for a limited set of topologies, or for a limited set of fault situations. This led to some efforts on *dynamic reconfiguration*, where the idea is to move from one routing function to another while the system is up and running [13][14][15][16][17]. The difficulty has been to do this in a way that does not create deadlocks in the transition phase [18].

Industrial installations of large computers have so far mainly resorted to the checkpoint/restart approach. This is, however, only a viable solution as long as the mean time between failures is high enough for the application to be expected to make significant progress - at least to the next checkpoint - before the next fault occurs. Due to the size of recent installations, we have seen some implementations of application-preserving fault tolerance mechanisms in InfiniBand clusters [19][20].

In [21] we proposed a host-side dynamic reconfiguration method for InfiniBand that is based on a network (subnet) manager, that dynamically reroutes the network. Furthermore it is based on changing routing tables in the switches and is independent of topology and routing algorithm. Combined with a topology agnostic routing algorithm, the method can in principle sustain connectivity of all connections, and thereby keep applications alive, as long as the topology is physically connected. Unfortunately, the solution was not practical due to the amount of messages generated by the subnet manager during reconfiguration.

In this paper we extend our previous work [21] with a scalable method for signalling reconfiguration events. When defining a scalable mechanism, there are two fundamental decisions that we need to make. The first is whether the

978-0-7695-4395-6/11 $26.00 © 2011 IEEE
DOI 10.1109/CCGrid.2011.48

event updates should be delivered in-bulk or one-by-one. The second is whether the mechanism to filter the event updates should be part of the subnet manager or offloaded to the hosts. In order to investigate this, we propose, implement, and evaluate three different event signalling mechanisms. In order to efficiently route the topology containing the fault, we use a topology agnostic routing algorithm called LASH in the experiments [2]. Our concept is, however, independent of topology and routing algorithm. Through measurements in our InfiniBand cluster and analytical studies we show that our best candidate reduces reconfiguration latency by more than 90% and in certain situations eliminates it completely. Furthermore, the processing overhead in the subnet manager is shown to be minimal.

The rest of this paper is organised as follows: We introduce the InfiniBand Architecture in Section II followed by a detailed motivation of our work in Section III. The three different signalling methods and algorithms are described in Section IV. Then, we describe the experimental setup in Section V followed by the performance analysis of the experimental and analytical studies in Section VI. Finally, we conclude in Section VII.

II. The InfiniBand Architecture

The InfiniBand (IB) architecture is a serial point-to-point technology that was first standardised in October 2000 [22] as a merge of two older technologies called Future I/O and Next Generation I/O. Due to its low latency, high bandwidth, and efficient utilization of host-side processing resources, it has been gaining acceptance within the High Performance Computing (HPC) community as a solution to build large and scalable computer clusters [23].

The de facto system software for IB is OFED which is developed by dedicated professionals and maintained by the OpenFabrics Alliance [24]. OFED is open source and is available for both GNU/Linux and Microsoft Windows. The improved signalling methods that we have proposed in this paper were implemented and evaluated in a development version of OpenSM, a subnet manager which is bundled together with OFED.

A. Subnet Management

IB networks are referred to as subnets, where a subnet consists of a set of hosts interconnected using switches and point-to-point links. A single subnet is scalable to more than ten-thousand nodes and two or more subnets can be interconnected using an IB router. The hosts and switches within a subnet are addressed using local identifiers (LIDs) and a single subnet is limited to 48151 unicast addresses.

An IB subnet requires at least one subnet manager (SM) which is responsible for initialising and bringing up the subnet, including the configuration of all the IB ports residing on switches, routers and host channel adapters (HCAs) in the subnet. At the time of initialisation, the SM starts in the *discovering state* where it does a sweep of the network in order to discover all switches and hosts. During this phase, it will also discover any other SMs present and negotiate who should be the master SM. When this phase is complete, the SM enters the *master state*. In this state, it proceeds with LID assignment, switch configuration, routing table calculations and deployment, and port configuration. At this point the subnet is up and ready to use.

After the subnet has been configured, the SM is responsible for monitoring the network for changes (e.g. a link goes down, a device is added, or a link is removed). If a change is detected during the monitoring process, a message (trap) is forwarded to the SM and it will reconfigure the network. A major part of the reconfiguration process (also known as ”heavy sweep”) is the rerouting of the network which must be performed in order to guarantee full connectivity, deadlock freedom, and proper load balancing between all source and destination pairs.

B. Queue Pairs

For InfiniBand, all HCAs communicate using Queue Pairs (QPs). A QP is created during the communication setup, and a set of initial attributes such as QP number, HCA port, destination LID, queue sizes, and transport service are supplied. When the communication is over, the QP is destroyed. An HCA can handle many QPs, each QP consists of a pair of queues, a Send Queue (SQ) and a Receive Queue (RQ), and there is one such pair present at each end-node that is participating in the communication. The send queue holds work requests to be transferred to the remote node, while the receive queue holds information on what to do with the data received from the remote node.

In addition to the QPs, each HCA has one or more Completion Queues (CQs) that are associated with a set of send and receive queues. The CQ holds completion notifications for the work requests posted to the send and receive queue.

C. Subnet Administration

The Subnet Administrator (SA) is a subnet database built by the master SM to store different information about a subnet. The communication with the SA is often needed by the end-node to establish a QP, and this is accomplished by sending a general service management datagram (MAD) through QP1. Both sender and receiver require information such as source/destination LIDs, service level (SL), MTU, etc. to establish a QP and this information can be retrieved from a data structure known as a path record that is provided by the SA. In order to obtain a path record, the end-node can use the *SubnAdmGet/SubnAdmGetTable* operation to perform a *path record query* to the SA. Then, the SA will return the requested path records to the end-node.

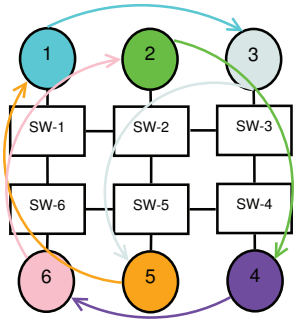


Figure 1. The 3x2 mesh topology with LASH routing. The arrows of the drawing indicate the direction of communication between the source and the destination node.

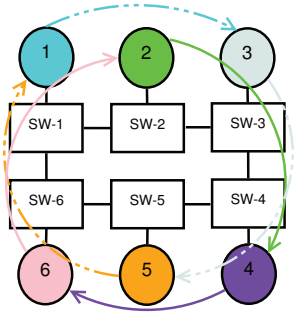


Figure 2. The ring topology with LASH routing. The dotted lines represent communication over VL1 and the solid lines represent VL0.

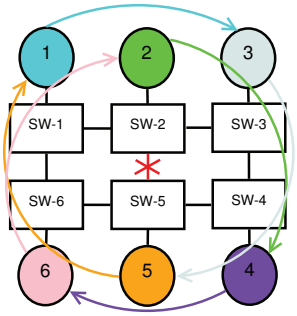


Figure 3. The routing of 3x2 mesh with LASH routing after link failure (red line in between of SW-2 and SW-5) without host-side reconfiguration.

D. Traps, Notices, and Event Forwarding

In InfiniBand, the SM is responsible for monitoring the network for changes. This is done using Subnet Management Agents (SMAs) that are present in every switch and every HCA. The SMAs communicate any changes, e.g. new connections, disconnections, and port state changes, to the SM using *traps* and *notices*. A trap is a message sent to alert end-nodes about a certain event, and it contains a notice attribute with the details about the event. Different traps are defined for different events, e.g. the *port state changed* trap (trap number 128) is sent from a switch to the SM whenever there is a change in the port state on one of the switch ports. I.e. a port has been connected or disconnected. If the SM receives trap number 128 with a notice indicating that a switch port has lost the connection to an end-node, the SM will reconfigure the subnet. In addition, the SM will generate an *out of service* trap (trap number 65) which indicates that the given end-node is unavailable. This trap is redirected from the SM to the SA in order to forward it out to all end-nodes that have subscribed to this trap.

In order to reduce the unnecessary distribution of traps, IB applies an *event forwarding* mechanism where end-nodes are required to explicitly subscribe to the traps they want to be informed about. E.g. the trap in the previous example would only be forwarded out by the SA if the end-node had subscribed to trap number 65.

III. Motivation

Our main goal in this paper is to find a scalable method for signalling reconfiguration events based on our earlier experience in [21]. The scalability that we are focusing on in this paper is the time domain rather than the space domain. In order to investigate this, we propose, implement, and evaluate three different event signalling mechanisms and we apply these mechanisms to the LASH algorithm in a fault tolerance context. Our concept is, however, independent of topology and routing algorithm. Moreover, it can be further extended to cover host-side reconfiguration for virtualisation, QoS, traffic aware routing, etc.

For InfiniBand, the LASH routing algorithm [2] is one of the deterministic and topology-agnostic routing engines that is offered by OpenSM. It uses virtual lanes (VLs) as a deadlock avoidance mechanism. Each virtual lane has a direct one-to-one mapping with the service levels (SLs). The applications used with the LASH routing algorithm are required to perform a path record query to get the path SL for each QP. Otherwise, the default SL will be used and a deadlock might happen. A major challenge with the LASH routing algorithm is that it requires host-side reconfiguration support in order to be able to handle failure, because after a failure the assigend SL for a given path might change. Host-side reconfiguration, as explained in [21], is used to update the path record attribute on a QP whenever a fault happens. In order to illustrate this problem, we use three simple scenarios as shown in Fig. 1, 2 and 3 where each of them uses the following synthetic communication pattern:1-

3, 2-4, 3-5, 4-6, 5-1 and 6-2. Fig. 1 shows the first scenario with the routing of a 3x2 mesh using the LASH routing algorithm. It only requires a single VL to avoid deadlock while maintaining shortest-path routing in the subnet. Fig. 2 shows the second scenario where 2 VLs are needed in order to have deadlock-free and shortest-path routing in a 6-node ring. Fig. 3 shows the last scenario where a 3x2 mesh has changed into a 6-node ring after the link between SW-2 and SW-5 has failed. The initial path record query that is performed by the application are based on a 3x2 mesh where each node only uses VL0. After the link failure, the SM is aware that the topology has changed into a 6-node ring and triggers LASH to regenerate a new routing table. After rerouting, LASH will require 2 VLs to avoid deadlock but there is no mechanism available to notify the nodes to modify their existing QPs with the updated SL. Therefore, all nodes continue to send packets on VL0, and eventually each of them ends up waiting for credits, which leads to deadlock.

In our previous work [21], we proposed and implemented a host-side dynamic reconfiguration mechanism that fixes the above mentioned issue. It is part of the host stack and dynamically reconfigures the QPs whenever the topology changes based on feedback from the SM. Moreover, the implementation preserves the QPs without interrupting the running applications. However, one issue with the previous solution is that it was not scalable. An increase in the number of switches and hosts, and the number of active QPs, lead to a huge increase in the host reconfiguration latency. This has been identified to be due to a simple signalling method in the SM where the reconfiguration is based on topology changes, but a topology change does not necessarily reflect path record changes. In this paper, we propose and evaluate three signalling methods with the aim of reducing the reconfiguration latency.

IV. Signalling Method

In this section, we propose three different signalling methods that are implemented in OpenSM and notify the host stack about network events. These events will help the host stack to decide whether it is required to reconfigure the path SLs in any of its existing QPs.

Fig. 4 shows the first approach named *3-way wildcard handshake*, a simple 3-way handshaking mechanism that only requires minimum modification to the SM. This is the solution that was used in our previous work. As illustrated in fig 4, there are 3 sequential steps in this approach. Firstly, the SM forwards out a re-path trap to notify the host that there is a change in the subnet topology. Secondly, the host constructs a message to query the SA for the latest path records. Finally, the SM returns the path records in-bulk using the Reliable Multi-Packet Transaction Protocol(RMPP) [22]. One of the major drawbacks of this approach is that the SM is unable to identify the updated paths when the subnet has changed. This disadvantage will cause the SM to always trigger the re-path trap for all hosts when a fault happens and creates unnecessary overhead on the host stack if no updates are necessary.

Algorithm 1 Trigger re-path trap with path record

```
1:  if never_been_routed == TRUE then
2:    for sw_src = 0 to max_switches do
3:      for sw_dst = 0 to max_switches do
4:        OLD_VL[sw_src][sw_dst] = VL[sw_src][sw_dst]
5:      end for
6:    end for
7:  else
8:    for sw_src = 0 to max_switches do
9:      for sw_dst = 0 to max_switches do
10:       if sw_src != sw_dst AND OLD_VL[sw_src][sw_dst]
          != VL[sw_src][sw_dst] then
11:         construct_repath_trap(sw_src, sw_dst)
12:         trigger(repath_trap)
13:         OLD_VL[sw_src][sw_dst] = VL[sw_src][sw_dst]
14:       end if
15:     end for
16:   end for
17: end if
```

Fig. 5 shows the second approach named *repath-only* where we have fully utilised the features of the re-path mechanism [22]. This is different from the first approach where a re-path trap is only used as a method to notify the host stack whenever there is a change in the subnet topology. In this method, every re-path trap includes an updated path record. This requires a method that can differentiate between the new and the old paths. Therefore, we have developed a new algorithm (Algo. 1) as the path record distinguisher (PRD) to identify the path changes and to trigger a re-path trap with an updated path record. The $never\_been\_routed$ flag determines whether the *OLD_VL* array has been assigned with values. If they are assigned, the algorithm compares the path changes between the *OLD_VL* array and the *VL* array. If the path record for a source/destination address pair has changed, the algorithm constructs a dedicated re-path trap which includes an updated path record and forwards it to the affected source and destination pair. Then, it updates the *OLD_VL* array accordingly. This solution is straightforward and does not involve any handshaking, but its scalability depends on the total number of path record changes, m.

Fig. 6 shows the third signalling method that we have proposed named *3-way hybrid handshake*. Due to the fact that the second approach is vulnerable to path changes, we need a mechanism that forwards the updated path records in-bulk. Hence, the third approach is a combination of the first and the second signalling methods where the 3-way handshaking is combined with a PRD in the SM to minimise unnecessary overhead. The algorithm is separated

Algorithm 2 Identify path record changes

```
1: if never_been_routed == TRUE then
2:   for sw_src = 0 to max_switches do
3:     for sw_dst = 0 to max_switches do
4:       OLD_VL[sw_src][sw_dst] = VL[sw_src][sw_dst]
5:     end for
6:   end for
7: else
8:   for sw_src = 0 to max_switches do
9:     for sw_dst = 0 to max_switches do
10:      if sw_src != sw_dst AND OLD_VL[sw_src][sw_dst]
         != VL[sw_src][sw_dst] then
11:        increment(path_changes)
12:        OLD_VL[sw_src][sw_dst] = VL[sw_src][sw_dst]
13:      end if
14:    end for
15:  end for
16: end if
```

Algorithm 3 Trigger wildcard re-path trap

```
1: if path_changes > 0 then
2:   trigger_wildcard(repath_trap)
3:   clear(path_changes)
4: end if
```

into 2 parts as illustrated in Algo. 2 and Algo. 3. Instead of sending out a dedicated re-path trap as is done by the second approach, it compares all the path record changes for every source/destination address pair and increments the *path_changes* variable if there is a difference. Subsequently, the *path_changes* variable that is assigned in Algo. 2 is used as a flag in Algo. 3 to determine whether a wildcard re-path trap will be forwarded to every host. Afterwards, each host requests the updated path records and reconfigures the QPs accordingly.

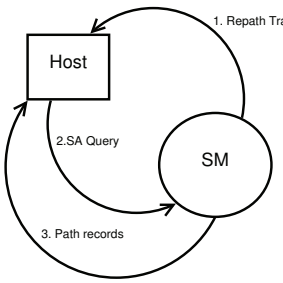


Figure 4. The 3-way wildcard handshake approach.

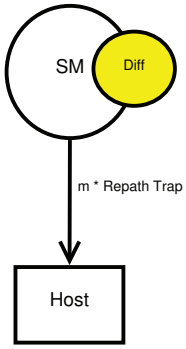


Figure 5. The repath-only approach, the "diff" represents the PRD in the SM that is used to identify the differences between an old and new path.

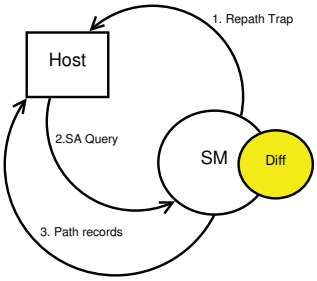


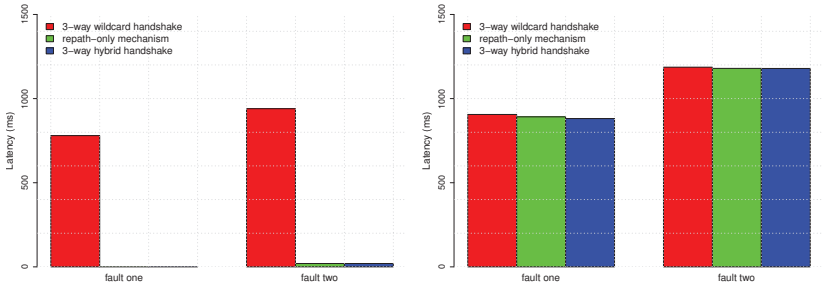
Figure 6. The 3-way hybrid handshake approach, the "diff" represents the PRD in the SM that is used to identify the differences between an old and new path.

V. Experiment Setup

Our test bed consists of six nodes and six switches. Each node is a Sun Fire X2200 M2 server that has a dual port Mellanox ConnectX DDR HCA with an 8x PCIe 1.1 interface, one dual core AMD Opteron 2210 CPU, and 2GB of RAM. The switches are four 24-port Infiniscale-III DDR switches and two 36-port Infiniscale-IV QDR switches which we used to construct the topologies illustrated in Fig. 1 and Fig. 2. Each host has Ubuntu Linux 8.04 x86_64 installed with kernel version 2.6.24-24-generic and a modified version of OFED 1.4.1 that contains the dynamic reconfiguration prototype. The subnet is managed by a modified version of OpenSM 3.2.5 that contains the proposed signalling methods.

VI. Performance Evaluation

Our performance evaluation consists of measurements on an experimental cluster and an analysis of scalability. For the cluster measurements, we use *latency* as our main metric to



(a) Host stack reconfiguration latency.

(b) OpenSM rerouting time.

Figure 7. OpenSM and host stack reconfiguration latency for two different faults.

compare the performance of our three signalling methods. For the mathematical analysis, we use *total management message overhead* and *latency* as the metrics to measure the scalability of each solution.

A. Experimental Results

We have carried out two different experiments to evaluate our signalling methods. Both experiments were performed using Integer Sort (IS), one of the application kernels in the NAS Parallel Benchmarks Suite [25]. The measured host stack latency was captured using kernel debug messages.

1) First Experiment: A synthetic fault is generated to emulate a link failure in the subnet while the IS benchmark is running. This experiment is repeated with 2 different faults. *Fault one* (link between SW-4 and SW-5 in Fig. 1) is a fault that will not cause path record changes. The same VLs are required to maintain a deadlock-free subnet as before the fault happened. *Fault two* (link between SW-2 and SW-5 in Fig. 1) is a fault that will generate a new set of path records because an additional VL is required to ensure that the subnet is deadlock-free. The main purpose of this experiment is to compare the latency added to the host stack during the host-side reconfiguration, and the total rerouting time in the SM for each of our signalling methods.

For *fault one*, Fig. 7a shows that the 3-way wildcard handshake requires approximately 780ms on each host to perform the reconfiguration. This is because there is no mechanism available in the SM to distinguish any path record changes. Consequently, a re-path trap is always triggered once there is a topology change. From a host stack perspective, it always queries the SA in order to obtain the latest path records and compares them with the existing path records in the active QPs list once a re-path trap is received. These operations are time-consuming and add unnecessary latency if there are no changes in the path records. For the other methods, neither the repath-only approach nor the 3-way hybrid handshake approach requires any cycles in the host stack when *fault one* happens. This is because the PRD in the SM knows that there are no changes in term of path records and will not trigger a re-path trap. Fig. 7b also illustrates that a simple PRD embedded in the SM does not increase the rerouting time of OpenSM.

For *fault two*, Fig. 7a also shows that the 3-way wildcard handshake still has the highest latency at around 930ms due to the unavailability of the PRD. When the link between SW-2 and SW-5 failed as shown in Fig. 3, both switches generate a *port state changed* trap (refer to section II-D) to notify the SM that the status of one of the ports has changed. As a result, the SM performs a ”heavy sweep” twice and therefore two continuous re-path traps are forwarded to the host. The data from the kernel debug messages show that the interval between the first and the second re-path trap contributes the highest latency. This is because the second re-path trap can only be triggered after the SM performed the second ”heavy sweep”. With the PRD, the SM knows that the second *port state changed* trap does not modify any path records and will not trigger a re-path trap. So, both repath-only and 3-way hybrid handshake are only taking around 20ms for the reconfiguration process.

In summary, the simple PRD that is available in both the repath-only and the 3-way hybrid handshake has completely removed the host stack latency of the 3-way wildcard handshake in a 3x2 mesh topology for *fault one* and reduces the host stack latency by 97.87% for *fault two*.

2) Second Experiment: The objective of this experiment is to assess the scalability of these signalling methods with an increased number of active QPs. We have only used *fault two* to emulate a link failure and it was repeated with 8, 16, 32, and 64 processes, because more processes means a higher number of QPs.

Fig. 8 shows that the 3-way wildcard handshake only requires an additional 70ms, whereas the 3-way hybrid

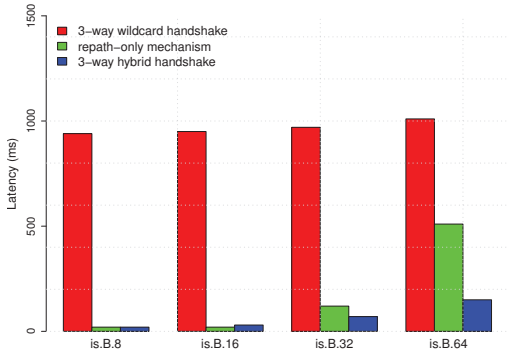


Figure 8. Host stack reconfiguration latency for *IS class B* using 8, 16, 32 and 64 processes.

handshake requires an additional 130ms when we increase the number of processes from 8 to 64. The repath-only mechanism requires an additional 490ms. Even though both the repath-only approach and the 3-way hybrid handshake have equivalent host reconfiguration latency with 8 and 16 processes, the host reconfiguration latency of the repath-only mechanism increases faster, starting from 32 processes. With 64 processes, the repath-only mechanism requires approximately 500ms, whereas the 3-way hybrid handshake only requires 150ms for the host reconfiguration. This shows that the latency of the repath-only approach increases non-linearly with the number of active QPs. Furthermore, this implies that the repath-only approach does not scale with the number of active QPs. This is caused by the fact that the repath-only mechanism can only include a single updated path record in a re-path trap but there are usually more than a single path record change when a fault happens. Consequently, the host needs to compare each path record encapsulated in a re-path trap with its active QPs and this comparison process is repeated for all the received re-path traps.

The 3-way wildcard handshake approach is scaling well when we increase the number of processes, but its initial latency with 8 processes is much higher than the other two approaches. As a result, its total host reconfiguration latency with 64 processes is double the latency of the repath-only approach and approximately seven times higher than the 3-way hybrid handshake. To conclude the second experiment, the 3-way hybrid handshake is the best approach because it scales well with a higher number of active QPs and have the lowest host reconfiguration latency with 64 processes.

Table I
TABLE OF NOTATION

Symbol	Meaning
n	Number of switches in the subnet
m	Number of path record changes
$ohd_{3way-ideal}$	Total message overhead of the ideal 3-way handshake mechanism
msg_{repath}	Message size of a re-path trap
msg_{sa_query}	Message size of a SA query
msg_{gmp}	Total message size of a general service management packet
ohd_{wc}	Total message overhead of the 3-way wildcard handshake mechanism
ohd_{repath}	Total message overhead of the repath-only mechanism
$ohd_{3way-hyb}$	Total message overhead of the 3-way hybrid handshake mechanism
msg_{pr}	Message size of a path record
msg_{rmpp}	Message size of a collection of path records in RMPP
lat_{wc}	Latency for the 3-way wildcarded handshake mechanism
lat_{repath}	Latency for the repath-only mechanism
$lat_{3way-hyb}$	Latency for 3-way hybrid handshake mechanism
pr/n	Average path record changes per node
t_1	Time required to construct a SA query message
t_2	Waiting time for getting the updated path records
t_3	Time required for ib_query_qp
t_4	Time required for APM mechanism
t_5	Waiting time before receiving the 2^{nd} re-path trap

B. Scalability

In this section, we have derived nine equations to analyse the scalability of our proposed signalling methods. The notations used in this paper are shown in table 1. Furthermore, we have also made an assumption that each switch has only one node connected to it and there are $n-1$ established QPs in each node, where n represents the number of switches.

Before the host attempts to reconfigure the active QPs, the host must be notified that there is a change in the subnet topology. This is performed by the re-path trap. Once the host has been notified by the re-path trap, the host must initiate a SA query message to obtain the latest path records. Then, the SA returns the path records to the requesting host. Therefore, the total *management message overhead* (MMO) that is created by an *ideal* 3-way handshake mechanism in a subnet can be represented by eq. 1 below. Where both msg_{repath_trap} and msg_{sa_query} are equivalent to a msg_{gmp} message that is 256 bytes in size, whereas the msg_{rmpp} message is a RMPP management packet.

$$ohd_{3way-ideal} = msg_{repath\_trap} * n \ + msg_{sa\_query} * n + msg_{rmpp} * n \quad (1)$$

The msg_{rmpp} message contains two msg_{gmp} messages and multiple data messages that will encapsulate the requested path records. The first msg_{gmp} is for the RMPP

initialisation while the second msg_{gmp} is for the acknowledgement after all the path records have been received. Eq. 2 represents the msg_{rmpp} message where the msg_{pr} message is a path record data structure with 64 bytes each (or 1/4 of the msg_{gmp} message).

$$\begin{aligned} msg_{rmpp} &= n * msg_{pr} + 2 * msg_{gmp} \\ &= n * msg_{gmp}/4 + 2 * msg_{gmp} \end{aligned} \quad (2)$$

As a result, the total MMO of an *ideal* 3-way handshake becomes:

$$ohd_{3way-ideal} = (4n + n^2/4) * msg_{gmp} \quad (3)$$

For the 3-way hybrid handshake, the total MMO is similar to the ideal 3-way handshake approach. This is because the PRD in the SM can identify the path record changes before initiating a re-path trap. Therefore, the equation for the 3-way hybrid handshake becomes:

$$\begin{aligned} ohd_{3way-hyb} &= ohd_{3way-ideal} \\ &= (4n + n^2/4) * msg_{gmp} \end{aligned} \quad (4)$$

The total MMO of the 3-way wildcard handshake is twice of the *ideal* 3-way handshake, because it always receives two re-path traps back-to-back after a single fault. Eq. 5 defines the total MMO of the 3-way wildcard handshake:

$$\begin{aligned} ohd_{wc} &= 2 * ohd_{3way-ideal} \\ &= 2 * (4n + n^2/4) * msg_{gmp} \end{aligned} \quad (5)$$

For the repath-only mechanism, the host does not query the SM for the latest path records as the updated path record is already encapsulated in the re-path trap. Thus, the total MMO of this approach can be represented by eq. 6 where m is the total number of path record changes.

$$\begin{aligned} ohd_{repath} &= msg_{repath\_trap} * m \\ &= msg_{gmp} * m \end{aligned} \quad (6)$$

Eq. 7 defines the latency for the 3-way wildcard handshake. Firstly, it requires t_1 to construct a SA query message. Secondly, it spends t_2 to wait for the updated path records from the SA. Thirdly, it spends t_3 to query all the active QPs, $(n-1)$. Finally, it spends t_4 to reconfigure the QPs accordingly if they need to be updated. Due to the fact that the SM cannot identify the path record changes, the second re-path trap is triggered and this requires $t5$, before it repeats the first two steps of the 3-way handshaking process again.

$$lat_{wc} = 2[t_1 + t_2 + (n-1) * t_3] + (pr/n) * t_4 + t_5 \quad (7)$$

The representation of the latency for the 3-way hybrid handshake is similar to lat_{wc} where t_1 and t_2 are required in order to obtain the latest path records. However, with the PRD in the SM, the second re-path trap can be avoided and t_5 is not required. So the equation for the 3-way hybrid handshake latency becomes:

$$lat_{3way-hyb} = t_1 + t_2 + (n-1) * t_3 + (pr/n) * t_4 \quad (8)$$

Eq. 9 represents the latency of the repath-only approach where the querying of all the active QPs, $(n-1)$ is repeated by a host depending on (pr/n), which is the average number of path changes per host. It also clearly illustrates that a higher number for $(n-1)$ or (pr/n) will increase the repath-only latency.

$$lat_{repath} = (n-1) * t_3 * (pr/n) + (pr/n) * t_4 \quad (9)$$

Based on the above equations, we can determine the scalability in terms of the total MMO and reconfiguration latency for each proposal. Other than the variable *n*, which is the number of switches within a subnet, the rest of the variables of the equations are based on measurements from our InfiniBand cluster or the *IbMgtSim* [26] simulator. The t_1, t_2, t_3, t_4 and t_5 are captured from kernel debug messages in the cluster. The pr, which is the total path record changes, is collected from a simple test case that is created using *IbMgtSim* to simulate a link failure in different N x M mesh topologies. The result of this test case is shown in Fig. 9. The most important observation is that a single fault in a subnet with more than 144 switches results in many path record changes when using LASH. E.g. for a subnet with 400 switches, a total of 79798 path records have changed after a single fault.

The plot in Fig. 10 shows the total MMO for each method. The total MMO for both repath-only and 3-way wildcard handshake have a comparable trend where they increase tremendously after 144 switches. Both approaches require 20 Mbytes of MMO for a single fault in a 400 switches subnet. On the contrary, the 3-way hybrid handshake only requires half the message at approximately 10 Mbytes. Even though the repath-only approach does not require handshaking, the overhead of the high frequency of re-path traps is larger than the overhead in the 3-way handshaking. Consequently, the repath-only mechanism has a higher MMO during the host reconfiguration. The MMO for the 3-way wildcard handshake is always higher than the MMO for the 3-way hybrid handshake due to the absence of the PRD. It always triggers two continuous re-path traps with a single fault, therefore, generates unnecessary management message overhead.

In terms of latency, Fig.11 shows the host reconfiguration latency for the three signalling methods based on eq. 7, 8 and 9. Both the 3-way handshake mechanisms scale well as there are only a minor increment in latency when the subnet

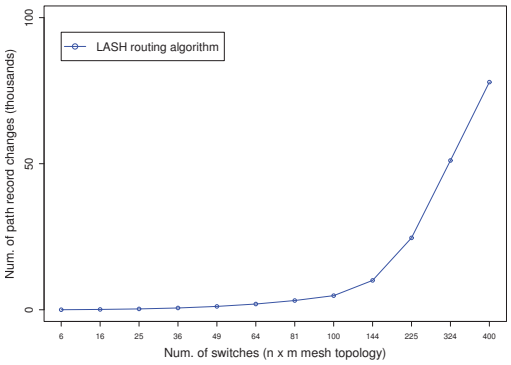


Figure 9. Number of path record changes for LASH routing as a function of network size (single fault).

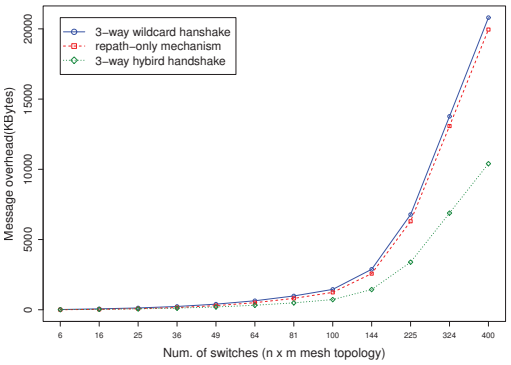


Figure 10. Total management message overhead as a function of network size during reconfiguration.

size increases from 6 to 400 switches. The repath-only mechanism starts with a very low latency, but it increases tremendously and goes beyond the total latency of the 3-way wildcard handshake after 100 switches. The major factor is that the updated path records are not delivered to the host in-bulk but they are enclosed one at a time in a single re-path trap. Consequently, a combination of a high number of QPs and high frequency of re-path traps in a large-scale cluster have amplified the latency of the repath-only mechanism. Fig. 11 shows that a single fault in a subnet with 400 switches needs 9s to reconfigure the QPs on each host when the repath-only mechanism is used. For the same subnet the 3-way wildcard handshake and the 3-way hybrid handshake only needs 1.12s and 0.15s, respectively.

To summarise, the *latency* and the *management message overhead* results from Fig. 10 and 11, respectively, shows a similar observation to our experimental results. It has clearly shown that the 3-way hybrid handshake is the best among these three signalling methods. It also needs to be mentioned that the repath-only mechanism is highly dependent on the total path record changes where both results in Fig. 10 and 11 have a trend similar to the total path record changes in Fig. 9.

VII. Conclusion and Future Work

Rerouting around faulty components, on-the-fly policy changes, and migration of jobs all require reconfiguration of data structures in the Queue Pairs residing in the hosts on an InfiniBand cluster. A part of this problem is related to how to signal reconfiguration events from the subnet manager to the hosts. In this paper we have proposed three methods for signalling such reconfiguration events using a subnet manager.

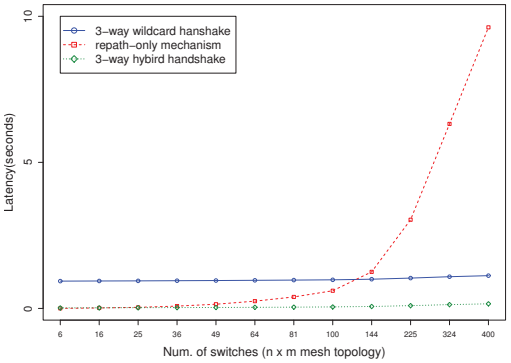


Figure 11. Host stack latency as a function of network size during reconfiguration.

Through measurements on our InfiniBand cluster and an analytical study, we show that our best proposal reduces reconfiguration latency by more than 90% and in certain situations eliminates it completely compared to previous efforts. Furthermore, the processing overhead at the subnet manager is minimal.

This contribution combined with our previous work in [21] is a complete prototype for host-side dynamic reconfiguration in InfiniBand. So far we have demonstrated this in the context of dynamic rerouting with faults. In the future, we plan to combine this with virtualisation, live migration, and traffic aware routing.

REFERENCES

[1] 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 Trans. Computers*, vol. 55, no. 4, pp. 400–415, 2006.

[2] O. Lysne, T. Skeie, S.-A. Reinemo, and I. r. Theiss, "Layered Routing in Irregular Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, pp. 51–65, 2006.

[3] T. Skeie, O. Lysne, J. Flich, P. Lopez, A. Robles, and J. Duato, "LASH-TOR: A Generic Transition-Oriented Routing Algorithm," in *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society Press, 2004, pp. 595–604.

[4] F. Silla and J. Duato, "High-performance routing in networks of workstations with irregular topology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 7, pp. 699–719, Jul. 2000.

[5] M.Koibuchi, A.Funahashi, A.Jourakua, , and H.Amano, "L-turn routing: an adaptive routing in irregular networks," in *Proceedings of the 2001 International Conference on Parallel Processing*. IEEE Computer Society, Sep. 2001, pp. 383–392.

[6] R. V. Boppana and S. Chalasani, "Fault-tolerant wormhole routing algorithms for mesh networks," *IEEE Transactions on Computers*, vol. 44, no. 7, pp. 848–864, 1995.

[7] S. Chalasani and R. V. Boppana, "Communication in Multicomputers with Nonconvex Faults," *IEEE Transactions on Computers*, vol. 46, pp. 616–622, May 1997.

[8] J. Montañana, J. Flich, A. Robles, and J. Duato, *A Scalable Methodology for Computing Fault-Free Paths in InfiniBand Torus Networks*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 4759. [Online]. Available: http://www.springerlink.com/content/j5132531q261r734

[9] Y. Mun and H. Y. Youn, "On performance evaluation of fault-tolerant multistage interconnection networks," in *SAC '92: Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing*. ACM Press, 1992, pp. 1–10.

[10] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, 2009.

[11] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 63–74, 2009.

[12] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 39–50, 2009.

[13] N. Natchev, D. Avresky, and V. Shurbanov, "Dynamic reconfiguration in high-speed computer clusters," in *Proceedings of the International Conference on Cluster Computing*. Los Alamitos: IEEE Computer Society, 2001, pp. 380–387.

[14] 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*, vol. 12, no. 2, pp. 115–132, Feb. 2001.

[15] T. Pinkston, R. Pang, and J. Duato, "Deadlock-free dynamic reconfiguration schemes for increased network dependeability," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 8, pp. 780–794, Aug. 2003.

[16] O. Lysne, J. M. Montanana, J. Flich, J. Duato, T. M. Pinkston, and T. Skeie, "An efficient and deadlock-free network reconfiguration protocol," *IEEE Transactions on Computers*, vol. 57, pp. 762–779, 2008.

[17] J. M. Montanana, J. Flich, and J. Duato, "Epoch-based reconfiguration: Fast, simple, and effective dynamic network reconfiguration," *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12, Apr. 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4536298

[18] 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*, vol. 16, pp. 412–427, 2005.

[19] A. Vishnu, A. Mamidala, S. Narravula, and D. Panda, "Automatic path migration over infiniband: Early experiences," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2007, pp. 1–8.

[20] A. Vishnu, M. Krishnan, and D. K. Panda, "An efficient hardware-software approach to network fault tolerance with infiniband," in *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 2009, pp. 1–9.

[21] 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*, 2010, pp. 126–135.

[22] *InfiniBand architecture specification*, 1st ed., InfiniBand Trade Association, November 2007.

[23] "Top 500 supercomputer sites," http://www.top500.org/, Nov. 2010.

[24] "The OpenFabrics Alliance," http://openfabrics.org/, Sep. 2010.

[25] "The nas parallel benchmarks," *The International Journal Of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, 1991. [Online]. Available: citeseer.ist.psu.edu/bailey95nas.html

[26] "Infiniband management simulator," http://www.openfabrics.org/archives/spring2006sonoma/OpenSM_Tools--OpenIB_OpenSM_v1.pdf.

Paper III

vFtree - A Fat-tree Routing Algorithm using Virtual Lanes to Alleviate Congestion

Wei Lin Guay, Bartosz Bogdanski, Sven-Arne Reinemo, Olav Lysne, and Tor Skeie

2011 IEEE International Parallel & Distributed Processing Symposium

vFtree - A Fat-tree Routing Algorithm using Virtual Lanes to Alleviate Congestion

Wei Lin Guay, Bartosz Bogdanski, Sven-Arne Reinemo, Olav Lysne, Tor Skeie
Simula Research Laboratory
P.O. Box 134,
NO-1325, Lysaker, Norway
E-mail:{weilin, bartoszb, svenar, olavly, tskeie}@simula.no

***Abstract*—It is a well known fact that multiple virtual lanes can improve performance in interconnection networks, but this knowledge has had little impact on real clusters. Currently, a large number of clusters using InfiniBand is based on fat-tree topologies that can be routed deadlock-free using only one virtual lane. Consequently, all the remaining virtual lanes are left unused.**

In this paper we suggest an enhancement to the fat-tree algorithm that utilizes virtual lanes to improve performance when hot-spots are present. Even though the bisection bandwidth in a fat-tree is constant, hot-spots are still possible and they will degrade performance for flows not contributing to them due to head-of-line blocking. Such a situation may be alleviated through adaptive routing or congestion control, however, these methods are not yet readily available in InfiniBand technology. To remedy this problem, we have implemented an enhanced fat-tree algorithm in OpenSM that distributes traffic across all available virtual lanes without any configuration needed. We evaluated the performance of the algorithm on a small cluster and did a large-scale evaluation through simulations. In a congested environment, results show that we are able to achieve throughput increases up to 38% on a small cluster and from 221% to 757% depending on the hot-spot scenario for a 648-port simulated cluster.

I. Introduction

The fat-tree topology is one of the most common topologies for high performance computing clusters today, and for clusters based on InfiniBand (IB) technology the fat-tree is the dominating topology. This includes large installations such as the Roadrunner, Ranger, and JuRoPa [1]. There are three properties that make fat-trees the topology of choice for high performance interconnects: deadlock freedom, the use of a tree structure makes it possible to route fat-trees without using virtual lanes for deadlock avoidance; inherent fault-tolerance, the existence of multiple paths between individual source destination pairs makes it easier to handle network faults; full bisection bandwidth, the network can sustain full speed communication between the two halves of the network.

For fat-trees, as with most other topologies, the routing algorithm is crucial for efficient use of the underlying topology. The popularity of fat-trees in the last decade led to many efforts trying to improve the routing performance. This includes the current approach that the OpenFabrics Enterprise Distribution (OFED) [2], the de facto standard for IB system software, is based on [3], [4]. These proposals, however, have several limitations when it comes to flexibility and scalability. One problem is the static routing used by IB technology that limits the exploitation of the path diversity in fat-trees as pointed out by Hoefler et al. in [5]. Another problem with the current routing are its shortcomings when routing oversubscribed fat-trees as addressed by Rodriguez et al. in [6]. A third problem is that performance is reduced when the number of compute nodes connected to the tree is reduced as addressed by Bogdanski et al. in [7]. And finally we have the problem of reducing the negative impact of congestion due to head-of-line blocking (HOL) [8]. This is not a routing problem per se as this should be handled by a congestion control mechanism, e.g. the mechanism found in IB [9], [10]. This mechanism, however, has its own set of challenges; one being that it is not generally available for existing IB hardware, another being that it is not yet understood how to configure congestion control for large networks [11]. Therefore, it is important to minimize the problem by other means. We suggest to do this using a combination of efficient routing and virtual lanes in an implementation that can be directly applied to IB or other technologies supporting multiple virtual channels.

Virtual lanes (or channels) were first introduced by Dally in the late eighties [12]. The intention at the time was to alleviate the restriction on routing flexibility that was imposed by deadlock considerations. In 1992 he published an analysis on the effect that virtual channels could have on network performance [13]. In spite of his positive findings, the usage of virtual channels has been confined to flexible routing and service differentiation, both in academia and in the industry. This is partly due to the fact that the analysis in the 1992 paper was based on assumptions that were not true for real technologies - in particular that a source was free to decide virtual lanes at the packet level, and not at the stream level. More recent works have addressed the congestion issue in several other ways. A recent proposal by Rodriguez et al. [14] also addresses this from a routing perspective, but in an application-specific manner and without using virtual lanes. Another approach using a combination of multipath routing and bandwidth estimation was proposed by Vishnu et al. in [15], but this is significantly more complex to implement than our proposal. A third proposal by Escudero-

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.28

Sahuquillo et al. [16] uses multiple queues at the input ports in the switches to avoid HOL, but this is not compatible with any existing network technology and requires new hardware to be built.

In this paper, we present the first results that indicate the gain of adding virtual lanes on a real commercial technology. These results deviate from Dally's in two respects. Firstly, they indicate that the performance gain is significantly bigger than he reported. Secondly,

that most of the gain can be realized by only 2 VLs, making it an obvious, readily available and potent improvement for all existing InfiniBand clusters. To be specific, we analyze the performance of the fat-tree routing algorithm in OpenSM in a hot-spot scenario and suggest a new routing algorithm, vFtree, that improves performance when hot-spots are present by using virtual lanes. Through a prototype implementation in OpenSM we demonstrate, using a small cluster, how virtual lanes can be used to achieve the same effect as IB congestion control. Then we generalize this into a new fat-tree routing algorithm that we evaluate for performance on a small cluster and for performance and scalability through simulations.

The rest of this paper is organized as follows: we introduce the InfiniBand Architecture in Section II followed by a description of fat-tree topologies and routing in Section III and a motivation for our proposal in Section IV. The algorithm is described in Section V. Then we describe the experimental setup in Section VI followed by the performance analysis of the experimental and simulated results in Section VII. Finally, we conclude in Section VIII.

II. The InfiniBand Architecture

InfiniBand is a serial point-to-point full-duplex technology, and the InfiniBand Architecture was first standardized in October 2000 [9]. Due to efficient utilization of host side processing resources, IB is scalable beyond ten thousand nodes with each having multiple CPU cores. The current trend is that IB is replacing proprietary or low-performance solutions in the high performance computing domain [1], where high bandwidth and low latency are the key requirements.

The de facto system software for IB is OFED developed by dedicated professionals and maintained by the OpenFabrics Alliance [2]. OFED is open source and is available for both GNU/Linux and Microsoft Windows. The improved vFtree algorithm that we propose in this paper was implemented and evaluated in a development version of OpenSM, which is a subnet manager distributed together with OFED.

A. Subnet Management

InfiniBand networks are referred to as subnets, where a subnet consists of a set of hosts interconnected using switches and point-to-point links. An IB fabric constitutes of one or more subnets, which can be interconnected together using routers. Hosts and switches within a subnet are addressed using local identifiers (LIDs) and a single subnet is limited to 48151 LIDs.

An IB subnet requires at least one subnet manager (SM), which is responsible for initializing and bringing up the network, including the configuration of all the IB ports residing on switches, routers and host channel adapters (HCAs) in the subnet. At the time of initialization the SM starts in the *discovering state* where it does a sweep of the network in order to discover all switches and hosts. During this phase it will also discover any other SMs present and negotiate who should be the master SM. When this phase is complete the SM enters the *master state*. In this state, it proceeds with LID assignment, switch configuration, routing table calculations and deployment, and port configuration. At this point the subnet is up and ready for use. After the subnet has been configured, the SM is responsible for monitoring the network for changes.

A major part of the SMs responsibility are routing table calculations. Routing of the network aims at obtaining full connectivity, deadlock freedom, and proper load balancing between all source and destination pairs. Routing tables must be calculated at network initialization time and this process must be repeated whenever the topology changes in order to update the routing tables and ensure optimal performance.

B. Virtual Lanes

InfiniBand is a lossless networking technology, where flow-control is performed per *virtual lane* (VL) [13]. The concept of virtual lanes is shown in Fig. 1. VLs are logical channels on the same physical link, but with separate buffering, flow-control, and congestion management resources. Fig. 2 shows an example of per VL credit-based flow-control where VL 0 runs out of credits after cycle 1 (depicted by a bold D) and is unable to transmit until credit arrives in cycle 9 (depicted by a bold C). As the other lanes have sufficient credit, they are unaffected and are able to use the slot that VL 0 would otherwise use. Transmission resumes for VL 0 when credit arrives.

The concept of virtual lanes makes it possible to build virtual networks on top of a physical topology. These virtual networks, or layers, can be used for various purposes such as efficient routing, deadlock avoidance, fault-tolerance and service differentiation. Our proposal exploits VLs for improved routing and network performance.

The VL architecture in IB consists of four mechanisms: *service levels*, *virtual lanes*, *virtual lane weighting*, and *virtual lane priorities*. A service level (SL) is a 4-bit field in the packet header that denotes what type of service a packet shall receive as it travels toward its destination. This is supplemented by up to sixteen virtual lanes. A minimum of two VLs must be supported: VL 0 as the default data lane and VL 15 as the subnet management traffic lane. By

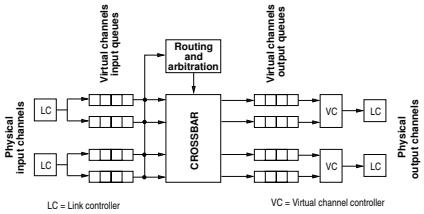


Figure 1. The canonical virtual channel architecture.

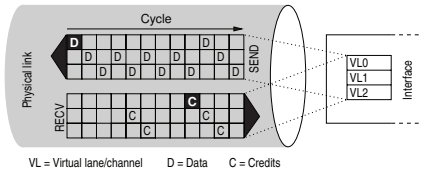


Figure 2. Virtual lane flow control in InfiniBand.

default, the sixteen SLs are mapped to the corresponding VL by the SL number, i.e. SL_i is mapped to VL_i. If a direct SL to VL mapping is not possible, the SL will be degraded according to a SL to VL mapping table. In the worst case only one data VL is supported and all SLs will be mapped to VL 0. In current IB hardware it is common to support nine VLs: one for management and eight for data.

Each VL can be configured with a weight and a priority, where the weight is the proportion of link bandwidth a given VL is allowed to use and the priority is either high or low. Our contribution in this paper will not make use of the weight and priority features in IB, we will use a direct SL to VL mapping and equal priority for all VLs. For more details about the weight and priority mechanisms consult [9], [17].

III. Fat-Trees

The fat-tree topology was introduced by C. Leiserson in 1985 [18], and has since then become a common topology in high performance computing (HPC). The fat-tree is a layered network topology with link capacity equal at every tier (applies for balanced fat-trees), and is commonly implemented by building a tree with multiple roots, often following the m-port n-tree definition [19] or the k-ary n-tree definition [20].

With the introduction of IB the fat-tree became the topology of choice due to its inherent deadlock freedom, fault tolerance, and full bisection bandwidth properties. It is used in many of the IB installations in the Top500 List, including supercomputers such as Los Alamos National Laboratory's Roadrunner, Texas Advanced Computing Center's Ranger, and Forschungszentrum Juelich's JuRoPa [1]. The Roadrunner differs from the two other examples in that it uses an oversubscribed fat-tree [21]. By carefully designing an oversubscribed fabric the implementation costs of an HPC cluster can be significantly reduced with only a limited loss in the overall application performance [22].

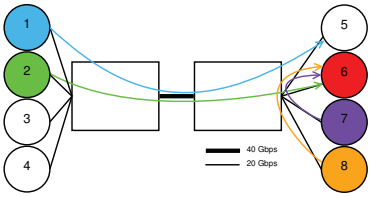


Figure 3. A simple congestion control experiment.

For fat-trees, as for most other network topologies, the routing algorithm is essential in order to exploit the available network resources. In fat-trees the routing algorithm consists of two distinct stages: the upward stage in which the packet is forwarded from the source, and the downward phase when the packet is forwarded toward the destination. The transition between those two stages occurs at the least common ancestor, which is a switch that can reach both the source and the destination through its downward ports. The algorithm ensures deadlock-freedom, and the IB implementation available in OpenSM also ensures that every path toward the same destination converges at the same root node, which causes all packets toward that destination to follow a single dedicated path in the downward direction [4]. By having dedicated downward paths for every destination, contention in the downward stage is effectively removed (moved to the upward stage), so that packets for different destinations have to contend for output ports in only half of the switches on their paths. In oversubscribed fat-trees, the downward path is not dedicated and is shared by several destinations.

Since the fat-tree routing algorithm only requires a single VL, the remaining virtual lanes are available for other purposes such as quality of service or for reducing the negative impact of congestion induced by the head-of-line blocking as we will do in this paper.

IV. Motivation

Algorithmic predictability of network traffic patterns is reduced with the introduction of virtualization and many-cores systems. When multiple virtualized clients reside on the same physical hardware, the network traffic becomes an overlay of multiple traffic patterns that might lead to hot-spots in the network. A *hot-spot* occurs if multiple flows are destined toward a single endpoint. Common sources for hot-spots include complex traffic patterns due to virtualization, migration of virtual machine images, checkpoint and restore mechanisms for fault tolerance, and storage and I/O traffic.

When a hot-spot exists in a network, the flows designated for the hot-spot might reduce the performance for other flows, called *victim flows*, not designated to the hot-spot. This is due to the head-of-line (HOL) blocking phenomena created by the congested hot-spot [8]. One way to avoid this problem is to use a congestion control mechanism such

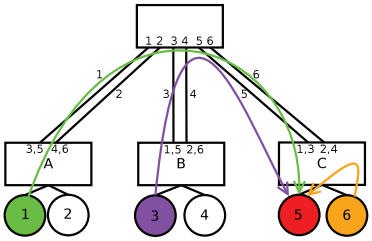
as the one recently specified and implemented for IB [9]. This mechanism was evaluated in hardware and shown to be working by Gran et al. [11], however this solution is not yet generally available. Furthermore, the selection of the appropriate CC parameters highly depends on the topology, is poorly understood, and incorrect parameters might lead to performance degradation.

It is a well known fact that using virtual lanes is one of the ways to alleviate network congestion [12], [13]. To demonstrate this as a plausible solution to alleviate congestion, we performed three simple hardware experiments on the topology presented in Fig. 3. In all the three experiments we used the following synthetic communication pattern: 1-5, 2-6, 8-6 and 7-6 which creates a hot-spot on endpoint 6. In Fig. 5a we show the per flow throughput for the first experiment where we observe that the victim flow (1-5) is affected by the congestion to the same degree as the flows contributing to congestion, even though the flow is not destined to the congested endpoint. This is caused by two main reasons. Firstly, the victim flow (1-5) is sharing the 32 Gb/s (effective bandwidth) link with the contributors to the congestion. Secondly, the HOL blocking reduced its bandwidth to the same share of switch-to-switch link bandwidth as the contributor, which is approximately 4.5 Gb/s. In Fig. 5b we present the result from the second experiment where the IB congestion control mechanism was turned on and configured using the parameters given by Gran et al. [11]. Now we observe that the victim flow is not deteriorated by HOL blocking and it manages to get about 13 Gb/s independent of other traffic flows.

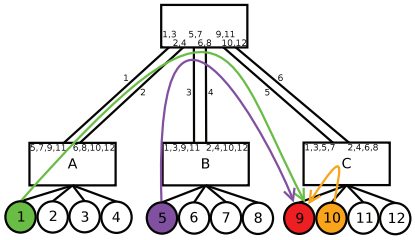
However, some oscillations occur among the flows due to the fact that the congestion control mechanism is dynamically adjusting the injection rate of the senders. In Fig. 5c we show the result of the third experiment where the victim flow (1-5) was manually assigned to a separate virtual lane. The measured result is similar to the experiment with IB CC turned on. In fact, the total network throughput shown in Fig. 6 is slightly better because we do not see the oscillations of IB CC mechanism. However, the traffic patterns used in this scenario are artificial and the victim flow is known in advance. Since we are unable to adapt to the congestion with this approach, we must rely on the statistical probability of improvements occurring when we distribute the traffic across a set of virtual lanes.

V. Design of Congestion Avoidance Mechanism Based on Routing

In this section, we propose a generic and simple routing-based congestion avoidance mechanism, which may easily be applied to high-performance network technologies supporting VLs. Our proposed routing algorithm uses the available virtual lanes to reduce the hot-spot problem by distributing *source destination* pairs across all available VLs. The approach is generic and can be applied to any



(a) A hot-spot scenario in a non-oversubscribed fat-tree.



(b) A hot-spot scenario in a oversubscribed fat-tree.

Figure 4. Experiment scenarios for the hardware testbed.

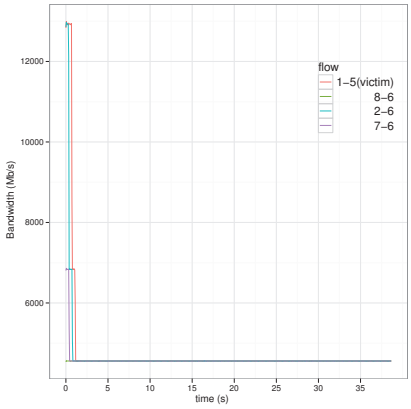
topology, but in this paper we focus on fat-trees because their regular structure makes it straightforward to distribute the destinations across VLs and they do not require VLs to be routed deadlock-free.

A. vFtree - Fat-tree Routing using Virtual Lanes

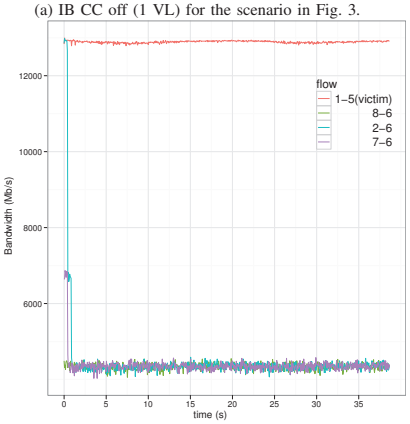
Routing in IB is a table lookup algorithm where each switch holds a forwarding table containing a linear list of LIDs (destination addresses) and the corresponding output ports. Our proposed algorithm is an extension of the current fat-tree routing algorithm that is available in OpenSM 3.2.5. The main feature of the new algorithm is to isolate the possible endpoint congestion flows by distributing *source-destination* pairs across virtual lanes. Our simple experiments in Section IV showed that virtual lanes can be used as a mechanism to alleviate endpoint congestion if the victim flow is identified. Unfortunately, as we have no reliable way of identifying victim flows, we propose to evenly distribute all *source-destination* pairs that share the same link in the upward direction across the available VLs.

The current fat-tree routing algorithm proposed by Zahavi et al. [4] is already optimized to avoid the contention for shift all-to-all communication traffic patterns. In a fully populated and non-oversubscribed fat-tree, this algorithm is equalizing the load on each port and always selects a different upwards link for destinations located on the same leaf switch. For example, in Fig. 4a destination 3 is reached from leaf switch A using link 1, while destination 4 is reached using link 2. It means that the destinations sharing the same upstream link are always located on different leaf switches.

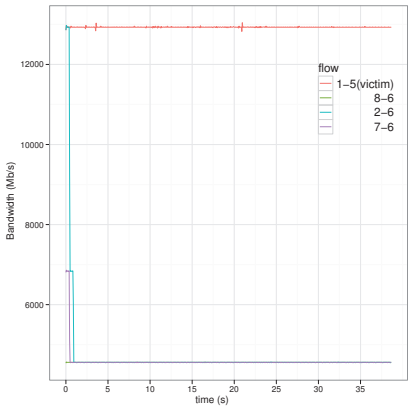
Our algorithm works on top of this and distributes all



(a) IB CC off (1 VL) for the scenario in Fig. 3.



(b) IB CC on for the scenario in Fig. 3.



(c) IB CC off (2 VLs) for the scenario in Fig. 3.

Figure 5. Per flow throughput comparison for IB CC experiment on Fig. 3.

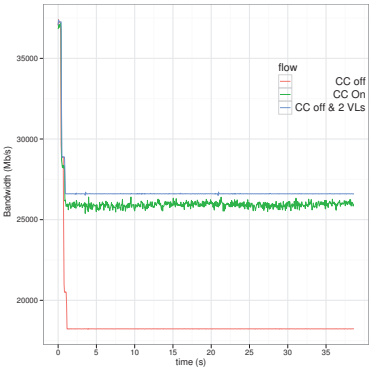


Figure 6. The total network throughput for experiment in Fig. 3.

destinations sharing the same upstream link across different virtual lanes. In detail, this means that from switch A we reach destination 3 using link 1 and VL 0, while destination 5 also uses link 1, but has VL 1 assigned. Consequently, if one of the designated destinations is the contributor to an endpoint hot-spot, the other destination flow (victim flow) sharing the same upstream port is not affected by HOL blocking because it uses a different virtual lane with its own buffering resources.

Ideally, the number of VLs required by our algorithm depends on the number of destinations that share the same upstream link, which is equivalent to the $N-1$ where N is the number of leaf switches. The $N-1$ virtual lanes cover all the traffic routed to destinations connected to other leaf switch except those destinations that are connected to the same leaf switch as the traffic source. In the implementation, however, the number of virtual lanes required is higher due to a requirement in the IB specification. The specification and the current implementation requires that the VL used from A to B is symmetric, which means that the communication from A to B and from B to A must be able to use the same VL.

The core functionality of the vFtree routing is divided into two algorithms as presented in Algo. 1 and Algo. 2. The former one distributes leaf $< sw_{src}, sw_{dst} >$ pairs across the available virtual lanes.

The outer ***for*** loop iterates through all the source leaf switches and the inner ***for*** loop iterates through all the destination leaf switches. In the inner ***for*** loop we check whether a VL has been assigned to a $< sw_{src}, sw_{dst} >$ pair, and, if not, we assign a VL accordingly. The requirement is that the VL assigned to a $< sw_{src}, sw_{dst} >$ must be the same as the VL assigned to $< sw_{dst}, sw_{src} >$ pair. The first ***if...else*** block starting at line 2 determines the initial *vl* value used for the inner ***for*** loop so the overlapping of VLs is minimized. The max_{vl} variable is an input argument for OpenSM that provides flexibility to the cluster administrator

who may wish to reserve some VLs for quality of service (QoS).

When a connection (Queue Pair) is being established in IB, the source node will query the path record and then Algo. 2 is executed. The arguments passed to this function are the source and destination addresses. Using these values, the source node obtains the VL from the array generated in Algo. 1 to be used for communication with the destination node.

Algorithm 1 Assign Virtual Lanes

Require: Routing table has been generated.
Ensure: Symmetrical VL for every <src, dst> pair.

```
for sw_src = 0 to max_leaf_switches do
  if odd(sw_src × 2/max_vl) then
    vl = ((sw_src × 2) % max_vl) + 1
  else
    vl = (sw_src × 2) % max_vl
  end if
  for sw_dst = 0 to max_leaf_switches do
    if sw_dst > sw_src then
      if VL[sw_src][sw_dst].set = FALSE then
        VL[sw_src][sw_dst].vl = vl
        VL[sw_src][sw_dst].set = TRUE
      end if
      if VL[sw_dst][sw_src].set = FALSE then
        VL[sw_dst][sw_src].vl = vl
        VL[sw_dst][sw_src].set = TRUE
      end if
      vl = incr(vl) % max_vl
    else if sw_dst == sw_src then
      VL[sw_dst][sw_src].vl = 0
      VL[sw_dst][sw_src].set = TRUE
    end if
  end for
end for
```

However, in OFED one major difference between vFtree and the conventional fat-tree routing is that for an application to acquire the correct SL in a topology routed using vFtree, a communication manager (CM) needs to be queried. The reason for that is only the CM can return the SL that was set up by the SM, and this SL implies which VL should be used. Otherwise, the default VL would be used, which is VL 0.

Algorithm 2 Get Virtual Lanes (LID_{src},LID_{dst})

```
dst_id = get_leaf_switch_id(LID_dst)
src_id = get_leaf_switch_id(LID_src)
return VL[src_id][dst_id]
```

B. Limitations

The main limitation of our approach is related to the number of VLs used. The IB specification defines 16 VLs, however, the actual implementation in today's hardware is limited to 8 VLs. This is insufficient to cover all the possibilities of endpoint congestion in a large-scale cluster which requires $N - 1$ VLs where N is the number of leaf switches. But as our results in Section VII show, large improvements are still possible with only two VLs. Another limitation is related to the use of VLs for other purposes such as QoS. QoS can be used together with vFtree routing, but then the number of SLs is reduced from 8 to 4 because for each SL two VLs are consumed by vFtree routing. For topologies other than fat-tree, which might use VLs for deadlock-free routing, the number of available SLs may be further reduced.

Furthermore, the assumption made for the vFtree algorithm is that the end node distribution is uniform. This is because the current version of the fat-tree routing algorithm has limitations when it comes to properly balancing the paths when the end node distribution is nonuniform as mentioned in [7].

VI. Experiment Setup

To evaluate our proposal we have used a combination of simulations and measurements on a small IB cluster. In the following subsections, we present the hardware and software configuration used in our experiments.

A. Experimental Test Bed

Our test bed consists of twelve nodes and four switches. Each node is a Sun Fire X2200 M2 server that has a dual port Mellanox ConnectX DDR HCA with an 8x PCIe 1.1 interface, one dual core AMD Opteron 2210 CPU, and 2GB of RAM. The switches are two 24-port Infiniscale-III DDR switches and two 36-port Infiniscale-IV QDR switches which we used to construct the topologies illustrated in Fig. 3 and Fig. 4. All the hosts have Ubuntu Linux 8.04 x86_64 installed with kernel version 2.6.24-24-generic and the subnet is managed by a modified version of OpenSM 3.2.5 that contains the implementation of the vFtree routing algorithm. Our *Perftest* [23] was also modified to support regular bandwidth reporting and continuous sending of traffic at full link capacity. The modified *Perftest* is used to generate the hot-spots shown in Fig. 4a and Fig. 4b.

B. Simulation Test Bed

To perform large-scale evaluations and verify the scalability of our proposal, we developed an InfiniBand model for the OMNeT++ simulator. The model contains an implementation of HCAs and switches with routing tables and virtual lanes. The network topology and the routing tables were generated using OpenSM and converted into OMNeT++ readable format in order to simulate real-world

systems. In the simulator, every source-destination pair has a VL assigned according to Algo. 1.

The simulations were performed on a 648-port fat-tree topology, which is the largest possible 2-stage fat-tree topology that can be constructed using 36-port switch elements. When fully populated this topology consists of 18 root switches and 36 leaf switches. Additionally, we performed the simulations of a 648-port topology that had an oversubscription ratio of 2:1. This was achieved by removing half of the root switches from the topology (9 switches). We chose a 648-port fabric because it is a common configuration used by switch vendors in their own 648-port systems [24], [25], [26]. Additionally, such switches are often connected together to form larger installations like JuRoPa. For the simulations we used a nonuniform traffic pattern, where 5% of all packets generated by a computing node was sent to a predefined hot-spot and the rest of the traffic was sent to a randomly chosen node. Additionally, we used multiple localized hot-spots by partitioning the network into three or nine segments, which corresponded to the physical features of the 648-port switch built in such a way that four leaf switch elements are placed on a single modular card.

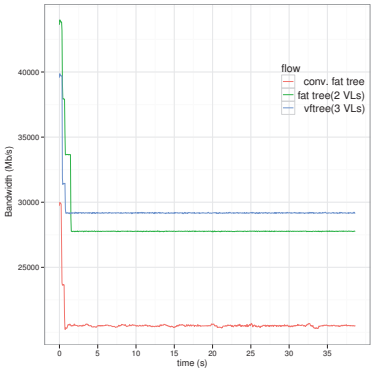
Each simulation run was repeated eight times with different seeds and the average of all simulation runs was taken. The packet size was 2 kB for every simulation. Furthermore, we have tuned the simulator to the hardware so we could observe the same trends when performing the data analysis. The results obtained through simulations exhibited the same trends as the results obtained from the IB hardware, with a maximum difference of 12% between the hardware and simulations.

VII. Performance Evaluation

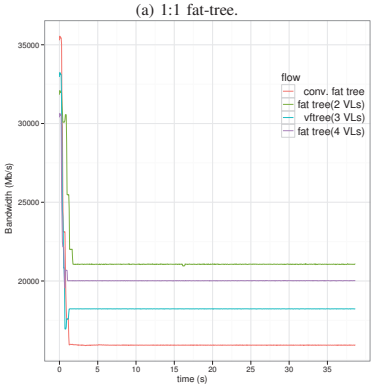
Our performance evaluation consists of measurements on an experimental cluster and simulations of large-scale topologies. For the cluster measurements we use the *per flow throughput* and *the total network throughput* as our main metrics to compare the performance of our proposed vFtree algorithm and the existing fat-tree algorithm. Additionally we use the results from the HPCC benchmark on certain scenarios to show how the algorithm impacts application traffic. For the simulations we use the *achieved average throughput per end node* as the metric for measuring the performance of the vFtree algorithm on the simulated 648-port topology. In both experimental cluster and simulations, all traffic flows are started at the same time and they are based on transport layer of the IB stack.

A. Experimental results

We carried out two different experiments on two different configurations which are a *non-oversubscribed fat-tree* as shown in Fig. 4a and *2:1 oversubscribed fat-tree* as shown in Fig. 4b. In the first experiment for the first configuration, a collection of synthetic traffic patterns ({1-5, 2-3, 3-5, 4-1, 6-5}) was selected to generate a hot-spot. In Fig. 4a node 5 is the hot-spot and the nodes 1, 3 and 6 are the contributors to the hot-spot. The flows 2-3 and 4-1 represent the victim flows. The purpose of the first experiment is to illustrate the negative impact the HOL blocking has on the victim flows. Additionally, it also shows how the vFtree routing algorithm avoids the negative effects of endpoint congestion. In the second experiment, we replaced the victim flows with the *HPC challenge* benchmark [27]. Our HPCC benchmark b_eff test suite was modified to generate 5000 random traffic patterns in order to obtain a more accurate result for randomly ordered ring bandwidth test. Even though the congested flows are still synthetically generated, this scenario resembles the network environment that an application could experience during congestion.



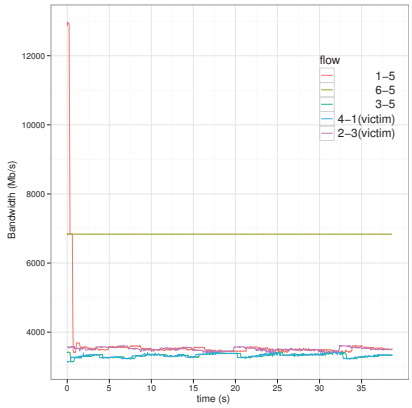
(a) 1:1 fat-tree.



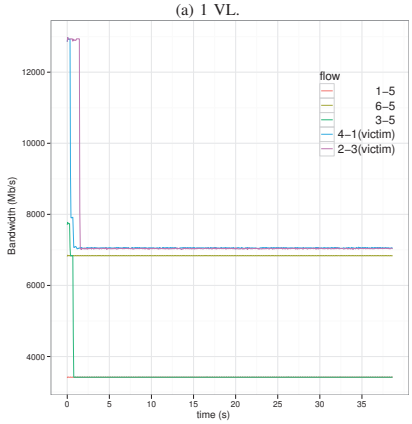
(b) 2:1 oversubscribed fat-tree.

Figure 7. Total network throughput using 1 to 3 VLs for the non-oversubscribed fat-tree and 1 to 4 VLs for the oversubscribed one (Fig. 4).

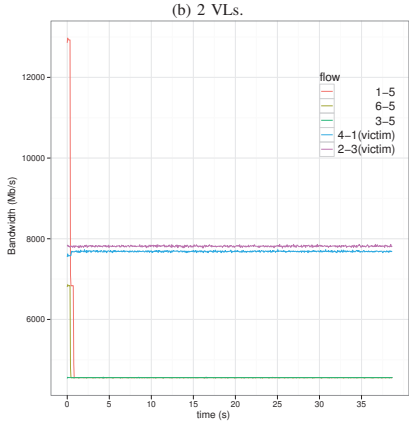
On the second configuration, shown in Fig. 4b, we repeated both of the experiments. A collection of synthetic traffic patterns {1-9, 6-9, 8-11, 10-9} was used for the first experiment. In this case, the hot-spot was at node 9 and the contributors were the nodes 1, 6 and 10. The flow 8-11



(a) 1 VL.



(b) 2 VLs.



(c) 3 VLs.

Figure 8. Per flow throughput for the fat-tree in Fig. 4a using 1 to 3 VLs.

represents the victim flow.

The first experiment (synthetic traffic) is described in Sections VII-A1 for non-oversubscribed fat-tree and VII-A2 for the oversubscribed one, and the second experiment (HPCC benchmark) is described in Section VII-A3 for both fat-tree topologies.

1) Non-oversubscribed fat-tree: Fig. 8 shows the per flow throughput of the first experiment when using 1 to 3 VLs. Fig. 8a shows the per flow throughput for the conventional fat-tree routing algorithm using one VL. The congestion towards node 5 blocks the traffic on physical link 1 and 3, and makes flows 4-1 and 2-3 victim flows. For these flows the throughput is less than half of the bandwidth that is available in the network, but due to the HOL blocking the bandwidth of flow 2-3 is reduced to the bandwidth that the congested flow 1-5 achieves across link 1. For the same reason flow 4-1 is reduced to the bandwidth of the congested flow 3-5. Furthermore, we also observe that, owing to the parking lot problem [28], flow 6-5 gets a higher share of the bandwidth toward destination 5 than flow 1-5 and 3-5.

If we manually assign the victim flows to a different VL, the situation improves as shown in Fig. 8b. The victim flows are able to avoid the HOL blocking, giving each of them an effective throughput of approximately 7 Gb/s, which corresponds to the actual available bandwidth in the network. The parking lot problem, however, is still present and we can see unfairness among the flows toward the endpoint hot-spot.

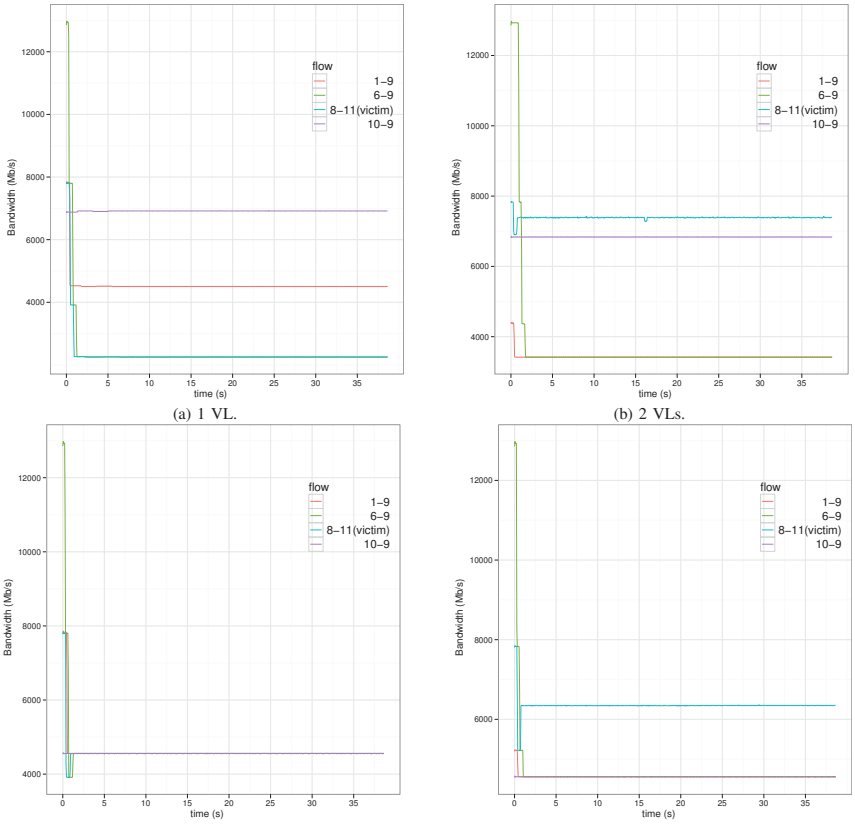
Fig. 8c shows the results of the repeated experiment with 3 VLs where both the HOL blocking of the victim flows and the parking lot problem toward the congested endpoint are solved. The reason for that is that the vFtree algorithm placed the routes for the victim flows in their own separate VLs, which solves the HOL blocking. Additionally, it placed the flows 1-5 and 3-5 on different VLs making the link arbitration between the sources 1,3, and 6 fair at switch C.

To summarize, we showed that the vFtree algorithm reduced both the HOL blocking and the parking lot problem when applied to fat-tree networks. The overall increase in total network throughput is approximately 38% when compared to the original fat-tree routing algorithm as shown in Fig. 7a.

2) 2:1 Oversubscribed network: In an oversubscribed network, the victim flows may suffer from HOL blocking in two different ways.

The first case is similar to the non-oversubscribed network where the performance reduction of the victim flow is due to a shared upstream link with the contributors to congestion.

In the second case, the victim flow shares both the upstream and the downstream link toward the hot-spot with the contributors, even though the victim flow is eventually routed to a different destination. In this section, we focus on the latter case because for the former case the results will be similar to the non-oversubscribed network scenario from Section VII-A1.



(a) 1 VL. (b) 2 VLs.

(c) 3 VLs. (d) 4 VLs.

Figure 9. Per flow throughput for the fat-tree in Fig. 4b using 1 to 4 VLs.

Fig. 9a shows the per flow throughput for the conventional fat-tree routing algorithm where the congestion in the network (Fig. 4b) blocked the victim flow (8-11). Among the congested flows, flow 6-9 obtains the lowest bandwidth because it also shares the upstream/downstream path with the victim flow. As a result, this also affects the victim flow (8-11) because it receives the same bandwidth across link 5 as flow 6-9 due to the HOL blocking, approximately 2.1 Gb/s.

If we manually assign the victim flow to a different VL, the situation improves as Fig. 9b shows. The results show that the victim flow (8-11) is able to avoid the HOL blocking and obtains approximately 7.5 Gb/s, the actual effective bandwidth that is available for this flow. As previously, unfairness exists among the contributing flows (1-9, 6-9 and 10-9) due to the parking lot problem.

As shown in Fig. 9d, the parking lot problem is resolved with 4 VLs where each of the contributors of the congestion manages to obtain 1/3 of the effective link bandwidth at approximately 4.5 Gb/s. Another observation is that the victim flow is getting a slightly lower bandwidth. This is due to the fact that the victim flow shares the downstream link with the rest of the congestion contributors. With a separate VL for each congestion contributor, flows 1-9 and 6-9 have an increased link bandwidth and, consequently, reduce the victim flow's bandwidth as shown in Fig. 9d.

In an oversubscribed fat-tree, utilizing more virtual lanes does not necessarily mean increasing the performance for certain types of traffic patterns. As shown in Fig. 7b, the total network bandwidth is higher with only 2 VLs when compared to the bandwidth obtained with 4 VLs. This is because with separate VL for each congestion contributor, the parking lot problem is resolved but the share of the link bandwidth given to the victim flow is reduced. Furthermore, if we would like to consider both cases of victim flow occurrence, it would require more VLs. Thus, in order to make our algorithm more predictable, we have decided to support only the first case of the victim flow as discussed in

Table I
RESULTS FROM THE HPC CHALLENGE BENCHMARK WITH CONVENTIONAL FAT-TREE ROUTING AND VFTREE.

Network latency and throughput	a) conventional Ftree	b) vFtree	c) Improvement
Min Ping Pong Lat. (ms)	0.002116	0.002116	0.0%
Avg Ping Pong Lat. (ms)	0.022898	0.013477	41.14%
Max Ping Pong Lat. (ms)	0.050500	0.043005	14.84%
Naturally Ordered Ring Lat. (ms)	0.021791	0.014591	33.04%
Randomly Ordered Ring Lat. (ms)	0.024262	0.015826	34.77%
Min Ping Pong BW (MB/s)	94.868	345.993	264.71%
Avg Ping Pong BW (MB/s)	573.993	830.909	44.75%
Max Ping Pong BW (MB/s)	1593.127	1594.338	0.07%
Naturally Ordered Ring BW (MB/s)	388.969246	454.236253	16.78%
Randomly Ordered Ring BW (MB/s)	331.847978	438.604531	32.17%

Table II
RESULTS FROM THE HPC CHALLENGE BENCHMARK WITH CONVENTIONAL FAT-TREE ROUTING AND VFTREE IN AN OVERSUBSCRIBED NETWORK.

Network latency and throughput	a) conventional Ftree	b) vFtree	c) Improvement
Min Ping Pong Lat. (ms)	0.002176	0.002176	0.0%
Avg Ping Pong Lat. (ms)	0.015350	0.009491	38.17%
Max Ping Pong Lat. (ms)	0.050634	0.043496	14.10%
Naturally Ordered Ring Lat. (ms)	0.021601	0.015616	27.70%
Randomly Ordered Ring Lat. (ms)	0.023509	0.016893	28.14%
Min Ping Pong BW (MB/s)	126.135	342.553	171.58%
Avg Ping Pong BW (MB/s)	825.874	1031.098	24.85%
Max Ping Pong BW (MB/s)	1594.186	1594.338	0.01%
Naturally Ordered Ring BW (MB/s)	369.021995	436.588321	18.31%
Randomly Ordered Ring BW (MB/s)	254.737276	355.412454	39.52%

earlier in this section. Nevertheless, we still manage to get about 20% improvement with vFtree routing algorithm that uses 3 VLs when compare with conventional fat-tree routing as illustrated in Fig. 7b. The reason behind this is that the parking lot problem is solved when each of the contributors to the hot-spot has a fair share of link bandwidth, but HOL blocking for the victim flow is not avoided. As observed on Fig. 9c, each of the flows (including the victim flow) obtains approximately 4.5 Gb/s of effective link bandwidth.

3) HPC challenge Benchmark: The second experiment is a combination of the I/O traffic generated by Perftest [23] and the application traffic generated by the HPCC benchmark [27]. The endpoint hot-spot is created using Perftest by running the traffic pattern presented in Fig. 4a for the non-oversubscribed configuration and in Fig. 4b for the 2:1 oversubscribed configuration). Simultaneously, we are running the HPCC benchmark in order to study the impact of congestion on the traffic generated by the HPCC benchmark.

Table I shows the comparison of the HPCC b_eff results between the conventional fat-tree routing and our vFtree routing algorithm in the presence of congestion in a non-oversubscribed network. The most interesting observation is that the randomly ordered ring bandwidth increased by 32.1% with our vFtree routing algorithm which uses only 3 VLs. We can see the improvement for all the latency and bandwidth tests, which is expected, as they correspond to the synthetic traffic patterns experiment that was carried out in the previous section. The results for the oversubscribed network are presented in Table II and the same trends are visible as for the non-oversubscribed network. We also managed to achieve an improvement in most of the latency and bandwidth tests when using our vFtree routing algorithm.

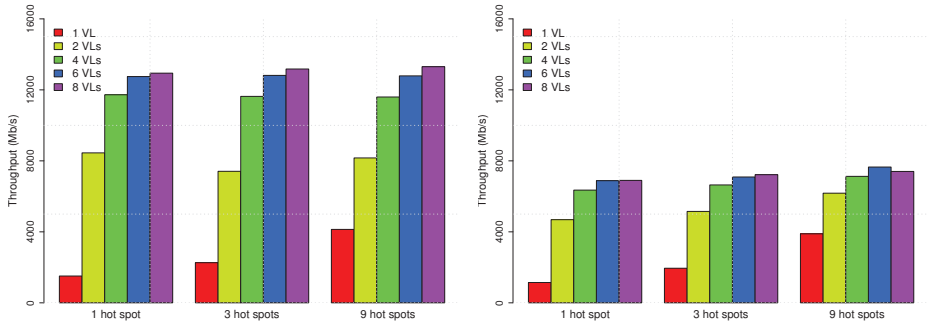
These results clearly illustrate the performance gain with our vFtree routing algorithm from the application traffic pattern's point of view.

B. Simulation results

An important question is how well the presented algorithm scales. Specifically, the purpose of the simulations was to show that the same trends exist when the number of nodes is large and the network topology corresponds to real systems. We performed the simulations on a 648-port switch without oversubscription and with 2:1 oversubscription.

1) Non-oversubscribed network: For a single hot-spot scenario, node 1 connected to switch 1 on modular card 1 (see Section VI-B for modular card definition) was the hot spot, and all the other nodes in the fabric were the contributors to the hot-spot. In case of three hot-spots, nodes 1 (modular card 1, switch 1), 217 (modular card 4, switch 1), and 433 (modular card 7, switch 1) were the hot-spots and the contributors were the nodes connected to modular cards 1-3 for node 1, modular cards 4-6 for node 217, and modular cards 7-9 for node 433. For a nine hot-spot scenario, the hot-spots were nodes 1, 73, 145, 217, 289, 361, 433, 505 and 577 connected to switch 1 at each modular card, and the contributors for each hot-spot were all the the other nodes connected to the same modular card as the hot-spot. In each scenario, the contributors sent 5% of their overall traffic to the hot-spot and 95% of other traffic to any other randomly chosen node in the fabric (it could also be any of the predefined hot-spots).

For the case presented on Fig. 10a, we observe that a single hot-spot dramatically decreases the average throughput per node because of the large number of victim flows. If more hot-spots are added, but the contributor traffic is localized (i.e. less victim flows), we observe that the



(a) Simulation results for 648-port switch with no oversubscription. (b) Simulation results for 648-port switch with 2:1 oversubscription.

Figure 10. Simulation results for 648-port switch.

throughput per node increases. The most important observation is the fact that every additional VL for data also reduces the number of victim flows, therefore increasing the network performance. The largest relative increase in the performance is obtained when adding a second VL. The relative improvement when compared with 1 VL is: 459% for 2 VLs, 676% for 4 VLs, 744% for 6 VLs and 757% for 8 VLs. The improvements when hot-spots are localized are smaller because of the fewer victim flows, which is best illustrated by the example from nine hot-spots case when comparing 1 VL to 8 VLs scenarios where we see an improvement of 221%. It also needs to be mentioned that the difference in average throughput per node between 6 VLs and 8 VLs is in a range of 400 Mb/s, so every additional VL provides a smaller throughput increase. Furthermore, it has to be noted that due to the randomness of the traffic there may exist more hot-spots in the network, and these hot-spots are not necessarily localized, which would explain the drops in average throughput per node for 2 VL scenario (yellow bars).

2) Oversubscribed network: Fig. 10b shows the results of a similar experiment performed on a 648-port network with 2:1 oversubscription. For every scenario, the hot-spots were chosen in the exact same manner as for the non-oversubscribed network and the same traffic patterns were used. We observe that the average throughput per node is generally halved when compared to the previous experiment with a non-oversubscribed topology. This is caused by the fact that the downward paths in the tree are shared by two destinations. The improvements when using 8 VLs compared to 1 VL are 503%, 270% and 90% for one, three or nine hot-spots respectively. Even though the result for nine hot-spots with 6 VLs was better (97% gain when compared with 1 VL) than with 8 VLs, we may assume this was a result of the randomness of the traffic as described in the previous section. This shows that the presented algorithm also reduced the number of the victim flows in an oversubscribed tree scenario, which makes it usable not only for small topologies, but also for real-world fabric examples.

To summarize, the large differences between the hardware results and the simulation results can be attributed to the fact that the simulated topologies are much larger in size than the hardware topologies we were able to construct. In the hardware the 38% improvement is visible for only two contributors sending to a single hot-spot. In the simulation, the worst case is if 5% of all 648 nodes are sending to a single hot-spot (plus any of the 95% of other nodes with a probability of $1/648$). It means we may safely assume that at any point in time at least 32 nodes are the contributors to the hot-spot. Therefore, every additional VL improves the network performance by reducing the number of victim flows, and because there are so many contributors and many more victim flows, the improvement is much larger for large-scale scenarios than for smaller topologies.

VIII. Conclusions and Future Work

In this paper, we demonstrated that by extending the fat-tree routing with VLs, we are able to dramatically improve the network performance in presence of hot-spots. Our solution is not only inexpensive, scalable, and readily available, but also does not require any additional configuration. By implementing the vFtree algorithm in OpenSM, we have shown that it can be used with the current state-of-the-art technology, and that the achieved improvements vary from 38% for small hardware topologies to 757% for large-scale simulated clusters when compared with the conventional fat-tree routing. Furthermore, the ideas from our proposal can be ported to other types of routing algorithms and similar improvements would be expected. Moreover, the solution is not restricted to InfiniBand technology only, and the concept can be applied to any other interconnects that support VLs.

In future, we plan to expand this solution to be able

to dynamically reconfigure the balancing of the network in case of faults, and to contribute our modifications to OpenFabrics community. Looking further ahead, we will also propose extensions to better support oversubscribed fat-trees by distributing the VLs in the downward direction.

REFERENCES

[1] "Top 500 supercomputer sites," http://www.top500.org/, Jun. 2010.

[2] "The OpenFabrics Alliance," http://openfabrics.org/, Sep. 2010.

[3] C. Gómez *et al.*, "Deterministic versus Adaptive Routing in Fat-Trees," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. IEEE CS, 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.5710

[4] E. Zahavi *et al.*, "Optimized InfiniBand TM fat-tree routing for shift all-to-all communication patterns," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 2, pp. 217–231, 2009. [Online]. Available: http://www3.interscience.wiley.com/journal/122677542/abstract

[5] T. Hoefler *et al.*, "Multistage switches are not crossbars: Effects of static routing in high-performance networks," in *Cluster Computing, 2008 IEEE International Conference on*, 2008, pp. 116–125.

[6] G. Rodriguez *et al.*, "Oblivious Routing Schemes in Extended Generalized Fat Tree Networks," *IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER '09.*, pp. 1–8, 2009.

[7] B. Bogdanski *et al.*, "Achieving Predictable High Performance in Imbalanced Fat Trees," in *Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS'10) - to appear*, 2010.

[8] G. F. Pfister and A. Norton, ""Hot Spot" Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 943–948, 1985.

[9] *InfiniBand architecture specification*, 1st ed., InfiniBand Trade Association, November 2007.

[10] G. Pfister *et al.*, "Solving Hot Spot Contention Using InfiniBand Architecture Congestion Control," Jul. 2005. [Online]. Available: http://www.cercs.gatech.edu/hpidc2005/presentations/GregPfister.pdf

[11] E. G. Gran *et al.*, "First Experiences with Congestion Control in InfiniBand Hardware," in *Proceeding of the 24th IEEE International Parallel & Distributed Processing Symposium*, 2010.

[12] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Transactions on Computers*, vol. 36, no. 5, pp. 547–553, May 1987.

[13] W. J. Dally, "Virtual-Channel Flow Control," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 194–205, Mar. 1992.

[14] G. Rodriguez *et al.*, "Exploring pattern-aware routing in generalized fat tree networks," in *Proceedings of the 23rd international conference on Supercomputing*. New York: ACM, 2009, pp. 276–285. [Online]. Available: http://portal.acm.org/citation.cfm?id=1542275.1542316

[15] A. Vishnu *et al.*, "Topology agnostic hot-spot avoidance with InfiniBand," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 3, pp. 301–319, 2009.

[16] J. Escudero-Sahuquillo *et al.*, "An Efficient Strategy for Reducing Head-of-Line Blocking in Fat-Trees," in *Lecture Notes in Computer Science*, D'Ambra, Pasqua And Guarracino, Mario And Talia, Domenico, Ed., vol. 6272. Springer Berlin / Heidelberg, 2010, pp. 413–427.

[17] S.-A. Reinemo *et al.*, "An overview of QoS capabilities in InfiniBand, Advanced Switching Interconnect, and Ethernet," *IEEE Communication Magazine*, vol. 44, Jul. 2006.

[18] C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Transactions on Computers*, vol. C-34, pp. 892–901, 1985.

[19] X.-Y. Lin *et al.*, "A multiple lid routing scheme for fat-tree-based infiniband networks," *Parallel and Distributed Processing Symposium, International*, vol. 1, p. 11a, 2004.

[20] F. Petrini *et al.*, "K-ary N-trees: High Performance Networks for Massively Parallel Architectures," Dipartimento di Informatica, Universita di of Pisa, Tech. Rep., 1995.

[21] K. J. Barker *et al.*, "Entering the petaflop era: the architecture and performance of Roadrunner," *SC Conference*, vol. 0, pp. 1–11, 2008.

[22] "HPC Fabric Analysis - Designed for Reduced Costs and Improved Performance," QLogic Corporation, 2010.

[23] "PerfTest - Performance Tests suite that bundle with OFED," Sep. 2009.

[24] "Sun Datacenter InfiniBand Switch 648," Oracle Corporation, http://www.oracle.com/us/products/servers-storage/networking/infiniband/034537.htm.

[25] "Voltaire QDR InfiniBand Grid Director 4700," Voltaire Inc., http://www.voltaire.com/Products/InfiniBand/Grid_Director_Switches/Voltaire_Grid_Director_4700.

[26] "IS5600 - 648-port InfiniBand Chassis Switch," Mellanox Technologies, http://www.mellanox.com/related-docs/prod_ib_switch_systems/IS5600.pdf.

[27] "HPC Challenge Benchmark," http://icl.cs.utk.edu/hpcc/.

[28] W. J. Dally and B. Towles, *Principles and practices of interconnection networks*. Morgan Kaufmann, 2004, ch. 15.4.1, pp. 294–295.

Paper IV

dFtree - A Fat-tree Routing Algorithm using Dynamic Allocation of Virtual Lanes to Alleviate Congestion

Wei Lin Guay, Sven-Arne Reinemo, Olav Lysne, and Tor Skeie

dFtree - A Fat-tree Routing Algorithm using Dynamic Allocation of Virtual Lanes to Alleviate Congestion in InfiniBand Networks

Wei Lin Guay[1], Sven-Arne Reinemo[1], Olav Lysne[1,2], Tor Skeie[1,2]
{weilin, svenar, olavly, tskeie}@simula.no

[1]Department of NetSys,
Simula Research Laboratory.
N-1364 Fornebu, Norway

[2]Department of Informatics,
University of Oslo.
N-0373 Oslo, Norway

ABSTRACT

End-point hotspots can cause major slowdowns in interconnection networks due to head-of-line blocking and congestion. Therefore, avoiding congestion is important to ensure high performance for the network traffic. It is especially important in situations where permanent congestion, which results in permanent slowdown, can occur. Permanent congestion occurs when traffic has been moved away from a failed link, when multiple jobs run on the same system, and compete for network resources, or when a system is not balanced for the application that runs on it.

In this paper we suggest a mechanism for dynamic allocation of virtual lanes and live optimization of the distribution of flows between the allocated virtual lanes. The purpose is to alleviate the negative effect of permanent congestion by separating network flows into *slow lane* and *fast lane* traffic. Flows destined for a end-point hot-spot is placed in the slow lane and all other flows are placed in the fast lane. Consequently, the flows in the fast lane are unaffected by the head-of-line blocking created by the hot-spot traffic.

We demonstrate the feasibility of this approach using a modified version of OFED and OpenSM with fat-tree routing on a small InfiniBand cluster. Our experiments show an increase in throughput ranging from 150% to 468% compared to the conventional fat-tree algorithm in OFED.

Categories and Subject Descriptors

D.4.4 [**OPERATING SYSTEMS**]: Communications Management—*Network communication*; C.2.5 [**COMPUTER-COMMUNICATION NETWORKS**]: Local and Wide-Area Networks—*High-speed*

General Terms

Design,Performance

NDM'11, November 14, 2011, Seattle, Washington, USA.

Keywords

Dynamic Reconfiguration, High Speed Interconnects, InfiniBand, Fat-tree, Performance Manager, Congestion Control

1. INTRODUCTION

For fat-trees, as with most other topologies, the routing algorithm is crucial for efficient use of the underlying topology. The popularity of fat-trees in the last decade has led to many efforts trying to improve the routing performance in fat-trees. This includes the current approach that the OpenFabrics Enterprise Distribution [2], the de facto standard for InfiniBand (IB) system software, is based on [7, 20]. These proposals, however, have several limitations when it comes to flexibility and scalability. One problem is the static routing used by IB technology that limits the exploitation of the path diversity in fat-trees as pointed out by Hoefler et al. in [13]. Another problem with the current routing is its shortcomings when routing oversubscribed fat-trees as addressed by Rodriguez et al. in [18]. A third problem is that performance is reduced when the number of compute nodes connected to the tree is reduced as addressed by Bogdanski et al. in [4]. And finally we have the problem of reducing the negative impact of congestion due to head-of-line (HOL) blocking [16]. This is not a routing problem per se as this should be handled by a congestion control mechanism, e.g. the mechanism found in IB [14, 15]. This mechanism, however, has its own set of challenges; one being that it is not supported by all IB hardware, another being that it is not yet understood how to configure congestion control for large networks [8]. Therefore, it is important to minimize the problem by other means. A recent proposal by Rodriguez et al. [17] addresses the congestion issue from a routing perspective, but in an application-specific manner and without using virtual lanes (VLs). Another approach using a combination of multipath routing and bandwidth estimation was proposed by Vishnu et al. in [19], but this is significantly more complex to implement than our proposal. A third proposal by Escudero-Sahuquillo et al. [6] uses multiple queues at the input ports in the switches to avoid HOL blocking, but this is not compatible with any existing network technology and requires new hardware to be built. In [10] we suggested the vFtree algorithm that uses a combination of efficient routing and virtual lanes to alleviate congestion. A problem with this approach, however, is that it is based on

a static distribution of source-destination pairs across a set of VLs. The static behaviour of the vFtree algorithm limits the performance whenever there is a mismatch between the current hot-spot and the precalculated distribution of source-destination pairs across VLs.

To rectify this we now propose the dFtree algorithm where the allocation of VLs is performed dynamically during network operation using an optimisation feedback cycle (Fig. 1). We introduce a *performance manager* [14] that monitors the network using hardware port counters to detect congestion and optimises the current VL allocation by classifying flows as either *slow lane* (contributors to congestion) or *fast lane* (victims of congestion). Then the optimisation is applied using the host side dynamic reconfigration method we proposed in [12]. The effect being that all flows contributing to congestion are migrated to a separate VL (slow lane) in order to avoid the negative impact of head-of-line blocking on the flows not contributing to congestion (victim flows). Compared to the vFtree approach we avoid the bottleneck of static allocation of VLs and we reduce the number of VLs required to two. Compared to normal IB congestion control [8] we remove the need for source throttling of the contributors. Furthermore, the current available IB congestion control (CC) parameters cause oscillations among all the flows because IB CC is dynamically adjusting the injection rate of the senders. As a result, this solution might not be suitable for congestion problem of a more persistent nature because the oscillations that can reduce the overall network throughput. In our previous work [12], we are able to obtain a better overall network throughput in certain congestion scenarios by avoiding the oscillations. Such persistent congestion problems occur when traffic has been moved away from a failed link, when multiple jobs run on the same system, and compete for network resources, or when a system is not balanced for the application that runs on it. Our approach handles persistent congestion problems by first detecting them, and thereafter dynamically redistributing the VL resources so as to obtain a balance that will be impossible to achieve statically at system start-up. The rest of this paper is organized as follows: we introduce the InfiniBand Architecture in Section 2 followed by the dFtree design and implementation in Section 3. Then we describe the experimental setup in Section 4 followed by the performance analysis in Section 5. Finally, we conclude in Section 6.

2. THE INFINIBAND ARCHITECTURE

InfiniBand is a serial point-to-point full-duplex technology, that was first standardized in October 2000 [14]. The current trend is that IB is replacing proprietary or low-performance solutions in the high performance computing domain [3], where high bandwidth and low latency are the key requirements.

The de facto system software for IB is Open Fabrics Enterprise Distribution(OFED) developed by dedicated professionals and maintained by the OpenFabrics Alliance. OFED is open source and is available for both GNU/Linux and Microsoft Windows. The dFtree algorithm that we propose in this paper was implemented and evaluated in a development version of OpenSM, which is the subnet manager (SM) distributed together with OFED.

2.1 The Subnet Manager

InfiniBand networks are referred to as subnets, where a subnet consists of a set of hosts interconnected using switches and point-to-point links. An IB fabric is constituted by one or more subnets, which can be interconnected together using routers. Hosts and switches within a subnet are addressed using local identifiers (LIDs) and a single subnet is limited to 48k LIDs.

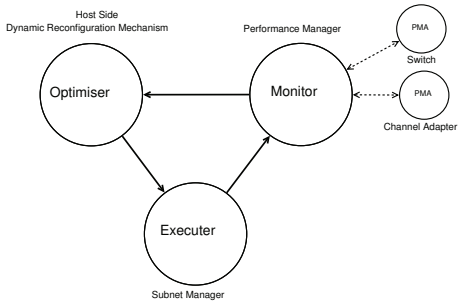


Figure 1: The performance optimisation feedback cycle.

An IB subnet requires at least one *subnet manager* (SM), which is responsible for initializing and bringing up the network, including the configuration of all the IB ports residing on switches, routers and host channel adapters (HCAs) and keeping the subnet operation in the subnet. A major part of the SMs responsibility is routing table calculations and deployment. Routing of the network aims at obtaining full connectivity, deadlock freedom, and load balancing between all source and destination pairs. Routing tables must be calculated at network initialization time and this process must be repeated whenever the topology changes in order to update the routing tables and ensure optimal performance.

2.2 The Performance Manager

Performance management is one of the general management services provided by IB to retrieve performance statistics and error information from IB components. Each IB device is required to implement a performance management agent (PMA) and a minimum set of performance monitoring and error monitoring registers. In addition, the IB specification also defines a set of optional attributes permitting the monitoring of vendor specific and additional performance and error counters.

The task of the *performance manager* (PM) [14] is to retrieve performance and error-related information from these registers. The information is retrieved by issuing a performance management datagram (MAD) to the PMA of a given device. The PMA then executes the retrieval and returns the result to the PM. As a result, the PM can use this information to detect incipient failures and based on this information, the PM can advise the SM about recommended or required path changes and performance optimisations.

3. THE DFTREE DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of the dFtree algorithm. The algorithm is generic and can

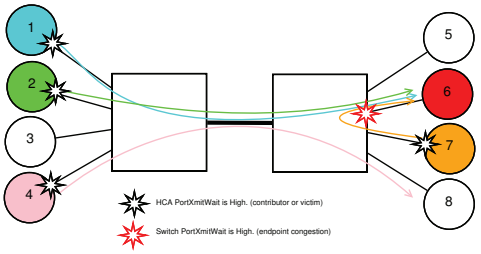


Figure 2: A simple congestion control experiment that illustrates how to use *xmtwait* performance counter to identify an endpoint hot-spot and its contributors.

be applied to any topology and routing algorithm, but in this paper we focus on fat-trees because of the simplicity they provide with respect to freedom from deadlock.

3.1 Overview

Performance tuning is the main activity associated with performance management where tuning consists of finding and eliminating bottlenecks. Hence, we are using the PM as one of the key components to enable dynamic allocation of virtual lanes to alleviate network congestion. Fig. 1 summarises the optimisation feedback cycle that consists of the SM, the PM and our modified host stack with the host side dynamic reconfiguration capability [12, 11]. In a subnet, the SM periodically sweeps the subnet to discover changes and maintain a fully connected subnet. Similarly, the PM periodically collects information from every component in the subnet in order to analyse the network performance. After the analysis, the PM forwards the relevant information to our modified host stack that reconfigures the virtual lanes in order to improve network performance.

3.2 Design

Head-of-Line blocking during traffic peaks or "hot-spot" traffic patterns is one reason for performance degradation in interconnection networks [16]. Common sources for hot-spots include complex traffic patterns due to virtualisation, migration of virtual machine images, checkpoint and restore mechanisms for fault tolerance, and storage and I/O traffic.

One way to avoid this problem is to use a congestion control (CC) mechanism such as the one recently specified and implemented in IB. This mechanism was evaluated in hardware and shown to be working by Gran et al. in [8], however, IB CC is not always available, e.g due to a mixture of old and new equipments in large clusters and this is often the case that will exist for years. In [10], we suggested an enhancement to the routing algorithm in OpenSM [20] that utilises multiple virtual lanes to improve performance during the existence of hot-spots. The virtual lanes are assigned statically during the routing table generation and can avoid the negative impact of the victim flows. However, the assumption is that the topology is a balanced, fully populated and fault-free fat-tree. In order to overcome the shortcomings in our previous work [10], we need a mechanism to identify the hot-spot flows and assign the virtual lanes dynamically.

Table 1: Notation

Counters	Meaning
$XmitWait$	The number of ticks when the port is selected had data to transmit but no data was sent during the entire tick because of insufficient credits or because of lack of arbitration. A tick is the IBA hardware sampling clock interval.
$XmitData$	The total number of data in double words transmitted on all VLs.
$Interval$	The number of seconds between each performance sweep.

Thus, we are using two standard IB performance counters (Table 1) as the metrics to identify endpoint hot-spots and their contributors dynamically during network operation. In addition, we have derived three equations to calculate the *normalised port congestion* and the *port utilisation* that are based on the above mentioned performance counters.

Eq. 1 below defines the normalised port congestion as the number of $XmtWait$s per second. An oversubscribed endnode with a high $Congestion_{port}$ value is either a contributor to the congestion or a victim flow. E.g the contributors at endnode 1,2,4 and the victim at endnode 7 in Fig. 2 have a high value for $Congestion_{port}$. On the other hand, an endnode that has a high $Congestion_{port}$ value for its *remote switch port* indicates that it is an endpoint hot-spot. E.g the switch port that is connected to node 6 in Fig. 2 has a high value for $Congestion_{port}$.

$$Congestion_{port} = \triangle XmitWait/Interval \quad (1)$$

Eq. 2 measures the sender port bandwidth for each port. This formula is derived from the $XmtData$ performance counter that represents the number of bytes transmitted between the performance sweeps. We multiply $XmtData$ by 4 in Eq. 2 because the $XmtData$ counter is measured in the unit of 32-bit words.

$$Bandwidth_{port} = \triangle XmitData * 4/Interval \quad (2)$$

Eq. 3 defines the port utilisation as the ratio between the actual bandwidth, $Bandwidth_{port}$ and the maximum supported link bandwidth.

$$Utilisation_{port} = Bandwidth_{port}/Max\ Bandwidth_{port} \quad (3)$$

These three equations are used in Algo. 1 to identify hot-spot flows as discussed in section 3.3.

3.3 Implementation

The dFtree implementation consists of two algorithms. Algo. 1 is used to identify the hot-spot flows, whereas Algo. 2 is used to reassign a hot-spot flow to a virtual lane classified as *'slow lane'*.

Algo. 1 is executed after every iteration of the performance sweep. The algorithm checks if the remote switch port of an endnode has a $Congestion_{port}$ value exceeding the *threshold*,

if this is true the conclusion is that the endnode is a hot-spot and the remote switch port is marked as a $hotspot_{port}$. After discovering an endpoint hot-spot, Algo. 1 triggers the forwarding a repath to all potential contributors. This trap encapsulates the LID, $hotspot_{LID}$, of the congested node.

The *threshold* value for $Congestion_{port}$ that is use to determine congestion is 100000 *XmtWait* ticks per second. The *XmtWait* counter is calculated on a per port basis, so the *threshold* value to determine congestion is applicable even if the network size increases.

In order to identify a potential contributor, we depend on both Eq. 1 and 3. An endnode where $Congestion_{port}$ exceeds the *threshold* indicates that it is either a hot-spot contributor or a victim flow, whereas $Utilisation_{port}$ is used to differentiate between a fair share link and a congested link. E.g if node A and node B are sending simultaneously toward node C. Even though both node A and B have a $Congestion_{port}$ value that exceeds the *threshold*, they receive a fair share of the link bandwidth toward node C. Thus, our algorithm will only mark an endnode as a potential contributor for $hotspot_{LID}$ and forward a repath trap if the $Congestion_{port}$ value is above the *threshold* and $Utilisation_{port}$ is less than 50%. In addition, if a new flow is directed to an existing $hotspot_{port}$, the new flow will still be moved to the *'slow lane'*. On the opposite, if an endnode is no longer a hot-spot, all flows that are directed to that endnode will be moved back to a virtual lane classified as *'fast lane'*. When a repath trap is received by a potential contributor, Algo. 2 is executed. The host will retrieve all the active QPs and compare them with the DLID in the repath trap. If a matching DLID is found in one of the QPs, the QP is reconfigured to use a *'slow lane'*. Initially, all QPs are initalised using a *'fast lane'*.

Algorithm 1 Detect endpoint hot-spot and its contributors

Ensure: Subnet is up and running and PM is constantly sweeping

for $sw_{src} = 0$ to sw_{max} **do**
  for $port_{sw} = 0$ to $port_{max}$ **do**
    if $remote\_port(port_{sw})$ == HCA **then**
      if $congestion_{port} >$ Threshold **then**
        if $port_{sw} \neq$ hot-spot **then**
          Mark $port_{sw}$ as $hotspot_{port}$
        end if
        Encapsulate $hotspot_{LID}$in a repath trap
        Encapsulate *slow lane* as $SL_{repath\ trap}$
        for $hca_{src} = 0$ to hca_{max} **do**
          if $congestion_{port} >$ Threshold **then**
            if $hca \neq hotspot_{LID}$ contributor **then**
              if $Utilisation_{port} < 0.5$ **then**
                Mark *hca* as $hotspot_{LID}$ contributor
                Forward repath trap to HCA
              end if
            end if
          end if
        end for
      else if $congestion_{port} <$ Threshold **then**
        if $port_{sw}$ == hot-spot **then**
          Clear $port_{sw}$ as $hotspot_{port}$
          Encapsulate $hotspot_{LID}$ in a unpath trap
          Encapsulate *fast lane* as $SL_{repath\ trap}$
          for $hca_{src} = 0$ to hca_{max} **do**
            if *hca* is $hotspot_{LID}$ contributor **then**
              Clear *hca* as $hotspot_{LID}$
              Forward unpath trap to HCA
            end if
          end for
        end if
      end if
    end if
  end for
end for

3.4 Limitations

In general, running OpenSM with the PM enabled adds an overhead because the PM periodically queries the performance counters in each component within the subnet. These queries, however, have minimal impact on data traffic as long as OpenSM is running on a dedicated node.

Another concern is that the detection of the hot-spot flows depends on the interval of the performance sweeps. If a hot-spot appeared just after iteration n, the hot-spot detection and the *'slow lane'* assignment can only be performed at iteration $n + 1$, i.e. t seconds later.

4. EXPERIMENT SETUP

To evaluate our proposal we have used both simulations and measurements on a small IB cluster. In the following subsections, we present the hardware and software configuration used in our experiments.

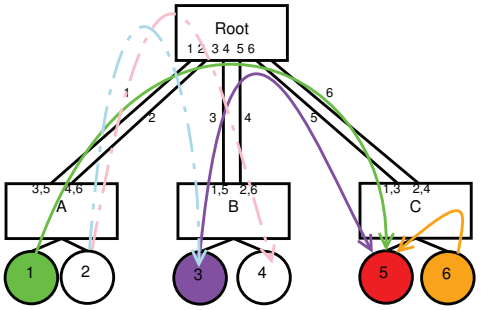
4.1 Experimental Test Bed

Our test bed consists of twelve nodes and four switches. Each node is a Sun Fire X2200 M2 server that has a dual port Mellanox ConnectX DDR HCA with an 8x PCIe 1.1 interface, one dual core AMD Opteron 2210 CPU, and 2GB of RAM. The switches are one 24-port Infiniscale-III DDR switches and three 36-port Infiniscale-IV QDR switches which we used to construct the topologies illustrated in Fig. 3. All the hosts have CentOS 5.3 installed with a customised IB host stack and the subnet is managed by a modified version of OpenSM 3.3.5 that includes the PM. The *Perftest* tool

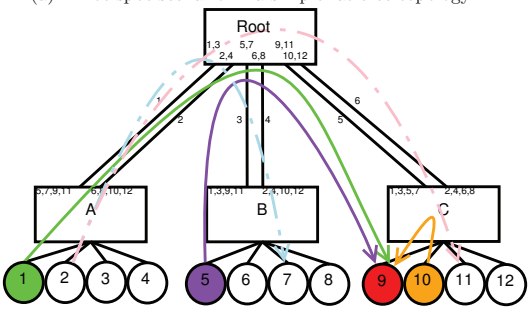
Algorithm 2 Reconfigure QP to slow/fast lane

Ensure: Host receives repath trap

for $QP_i = 0$ to QP_{max} **do**
  if $DLID_{QP} == DLID_{repath\ trap}$ **then**
    Reconfigure SL_{QP} according to $SL_{repath\ trap}$
  end if
end for



(a) A hot-spot scenario in a simple fat-tree topology.



(b) A hot-spot scenario in an over-subscribed fat-tree topology. In order to illustrate that network bandwidth is not a problem, link 1-6 are QDR links whereas the links between the leaf switches and the end nodes are DDR links.

Figure 3: Experiment scenarios for the hardware testbed. The solid lines represent the hot-spot flows and the dotted lines represent the victim flows. The numbers stated in the leaf switch A,B,C and root switch represent the routing table.

was also modified to support regular bandwidth reporting and continuous sending of traffic at full link capacity. The modified *Perftest* is used to generate the hot-spots shown in Fig. 3a and Fig. 3b.

4.2 Simulation Test Bed

To perform large-scale evaluations and verify the scalability of our proposal, we developed an InfiniBand model for the OMNeT++ simulator [9]. The simulations were performed on a 648-port fat-tree topology as shown in Fig. 4 with a nonuniform traffic pattern, where 5% of all packets generated by a compute node was sent to a predefined hot-spot and the rest of the traffic was sent to a randomly chosen node. Additionally, we used multiple localised hot-spots by partitioning the 648-port switched network into one, three or nine segments as described in section 5.4.

5. PERFORMANCE EVALUATION

Our performance evaluation consists of measurements on an experimental cluster and simulations of large-scale topologies. For the cluster measurements we use the *per flow throughput* and the *worst case throughput during congestion* as the main metrics to compare the performance between the dFtree algorithm and the existing fat-tree algorithm. Additionally, we use the results from the HPCC benchmark to show how the algorithm impacts application traffic. For the simulations we use the *achieved average throughput per end node* as the metric for measuring the performance of the dFtree algorithm on the simulated 648-port topology.

5.1 Synthetic Traffic Patterns - Non oversubscribed fat-tree

We carried out two different experiments on a *non oversubscribed fat-tree* as shown in Fig. 3a. For both experiments, a collection of synthetic traffic flows ({1-5, 3-5, 6-5}) is used to generate a hot-spot as shown in Fig. 3a. Node 5 is the hot-spot, nodes 1, 3 and 6 are the contributors to the hot-spot.

5.1.1 Experiment I

In this experiment, the victim flow (2-3) is started first and then the contributors ({1-5, 3-5, 6-5}) are added after 13s. This experiment illustrates the negative impact of HOL blocking on the victim flow. It also shows how our dFtree algorithm avoids it.

Fig. 5 shows the per flow throughput with and without the dFtree algorithm. In Fig. 5a, the victim (2-3) is running at 12.9 Gbps before the congested flows are introduced. Starting from 13s, the congestion towards node 5 blocks the traffic on link 1 and 3. Consequently, the bandwidth of flow 2-3 is reduced to 3 Gbps, the same bandwidth that the congested flow 1-5 achieved across link 1 due to the HOL blocking. Fig. 5b shows the per flow throughput with the dFtree algorithm. Now, the victim flow achieves a throughput of 7.5 Gbps and it is not affected by the congestion. We have also summarised the worst case per flow throughput with and without the dFtree algorithm during the congestion in Fig. 6. With dFtree, the victim flow has improved approximately 150% from 3 Gbps to 7.5 Gbps without impacting the contributors.

The reason that the dFtree algorithm can avoid HOL blocking is that the PM detects that node 5 is the hot-spot when the congested flows are introduced. After the analysis, a repath trap that encapsulates node 5 as a hot-spot LID is forwarded to the source node of the contributors and the victim flows. When a sender (hot-spot contributor or a victim flow) receives the repath trap, it retrieves all the active QPs and compares the destination LID with the repath trap LID. If a QP has a matching destination LID it will be reconfigured to the *'slow lane'*. As you can see from Fig. 5b, there is a slight glitch for flow 1-5, 3-5 and 6-5 between 14s and 16s because the QPs are reconfiguring to the *'slow lane'*. After the reconfiguration, the victim flow regains its throughput to 7.5 Gbps because the dFtree algorithm placed the congested flows in a separated VL (*'slow lane'*) that resolves the HOL blocking.

Another observation is that flow 6-5 has a higher share of the bandwidth at 6.8 Gbps toward the hot-spot than flow 1-5 and 3-5 because of the parking lot problem [5]. In order to resolve the parking lot problem, we would need to use additional VLs.

5.1.2 Experiment II

In this experiment, the victim flow is now flow 2-4. It has to share the upstream link with flow 1-5, one of the contributors, if a fault happens on link 1 or 2. Thus, in this

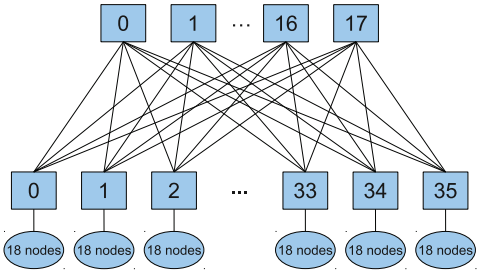


Figure 4: A 648-port switch fat-tree topology.

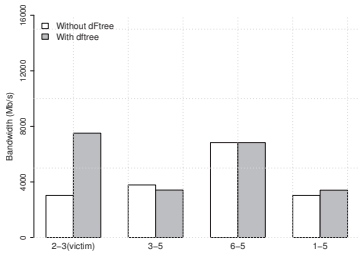
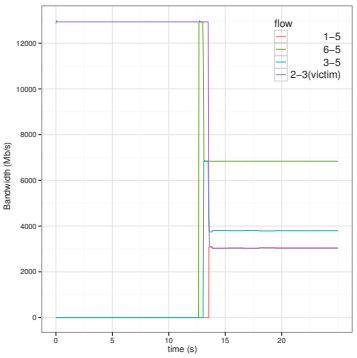
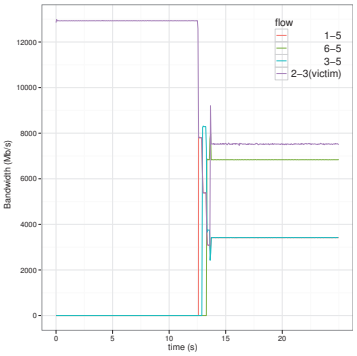


Figure 6: Per flow worst case bandwidth during congestion in Experiment I.



(a) Per flow throughput without dFtree.



(b) Per flow throughput with dFtree.

Figure 5: Experiment I using scenario in Fig. 3a.

experiment all flows are started at the same time, but after 13s we disconnect link 2 to emulate a link failure.

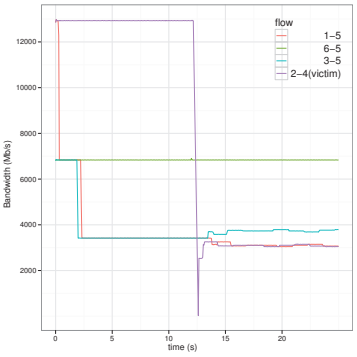
Fig. 7 shows the per flow throughput with and without the dFtree algorithm in a faulty link scenario. Fig 7a shows that the victim flow 2-4 achieves its maximum bandwidth at 12.9 Gbps in the presence of congested flows (before 13s). The victim (flow 2-4) is not impacted by the congested flows because it uses link 2 whereas flow 1-5, one of the contributors, uses link 1 as the upstream link. After 13s, a fault happens at link 2 that triggers the SM to generate a new set of routing tables that causes both flow 1-5 and flow 2-4 to share the same upstream link. Consequently, without the dFtree algorithm the throughput of flow 2-4 drops to 3 Gbps due to the HOL blocking caused by the congested flow 1-5. On the other hand, with the dFtree algorithm as shown in Fig. 7b, flow 2-4 instantly regains its link bandwidth at 7.5 Gbps after the link failure because the congested flows were separated from the normal traffic flow and placed into the *'slow lane'* before the fault happened. The dip that causes the throughput drop at 13s is due to OpenSM rerouting the network after the fault happened. Furthermore, there is also a glitch in each of the congested flows (flow 1-5, 3-5 and 6-5) in between 3-5s because the host is reconfiguring the hot-spot contributors QP to the *'slow lane'*.

In summary, Fig. 8 shows that the dFtree algorithm achieves approximately a 150% improvement in throughput from 3 Gbps to 7.5 Gbps for the victim flow worst case throughput without affecting the congested flows after the fault happened.

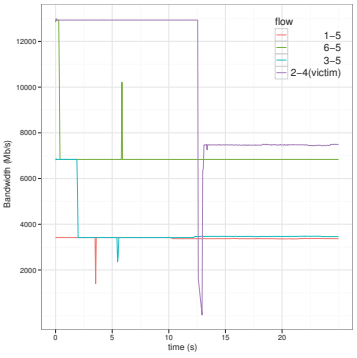
5.2 Synthetic Traffic Patterns - 2:1 oversubscribed fat-tree

We have also carried out two different experiments on a *2:1 oversubscribed fat-tree* as shown in Fig. 3b. In an *oversubscribed fat-tree*, the downward path is not dedicated to a single destination, but it is shared by several destinations. The term *2:1* means that a downward path is shared by two destinations. Furthermore, in order to show that lack of network bandwidth is not the cause of the problem when fat-trees are *oversubscribed*, we used quad data rate (QDR) for link 1 to 6 in Fig. 3b (the links connecting switch A, B, and C with the upper root switch).

In a 2:1 oversubscribed fat-tree, there are two situations where the victim flows may suffer from HOL blocking be-



(a) Per flow throughput without dFtree.



(b) Per flow throughput with dFtree.

Figure 7: Experiment II with a faulty link using scenario in Fig. 3a.

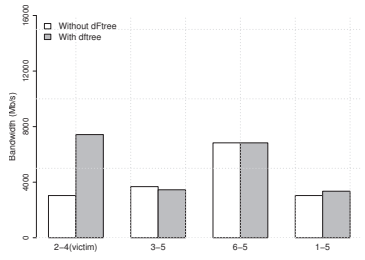


Figure 8: Per flow worst case bandwidth during congestion in Experiment II.

cause the links are oversubscribed. The first case is similar to section 5.1.1 where the performance reduction is due to the upstream link being shared with the congestion contributors. The second case is when both the upstream and downstream links are shared as discussed in section 5.2.2. For both experiments, we use the synthetic traffic flows {1-9, 5-9, 10-9} to evaluate the negative impact of HOL blocking in the *2:1 oversubscribed fat-tree*.The hot-spot is at node 9, and node 1, 5 and 10 are the contributors. The victim flow is started first and the contributors are added after 13s.

5.2.1 Experiment III

This experiment is similar to the Experiment I except that it is performed on a 2:1 oversubscribed fat-tree. Flow 2-7 is selected as the victim flow. Fig. 9a shows the per flow throughput without the dFtree algorithm where the victim (flow 2-7) drops from 12.9 Gbps to 3.4 Gbps. On the opposite, as shown in Fig. 9b, the dFtree algorithm managed to recover the throughput for the victim flow 2-7 to 12.9 Gbps during congestion.

However, the recovery takes approximately 2s because the hot-spot happens right after the performance sweeping and it needs to wait for the next sweep to detect and reallocate the hot-spot flows to the *'slow lane'*.

Fig. 10 shows the comparison of the per flow worst case bandwidth during the congestion with and without the dFtree algorithm. It is obviously illustrated in Fig. 10 that the victim flow worst case throughput with the dFtree algorithm has improved approximately 278% from 3.4 Gbps to 12.9 Gbps compared to not using the dFtree algorithm.

5.2.2 Experiment IV

In this experiment, we change the victim to flow 2-11. This flow is selected because it shares the same upstream and downstream link with the congested flow 1-9.

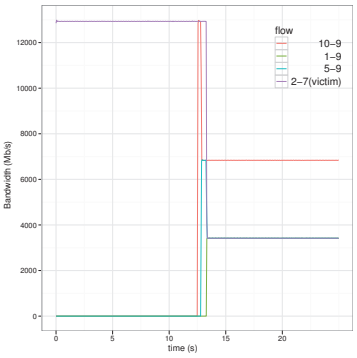
Without the dFtree algorithm, as shown in Fig. 11a, both victim flow 2-11 and congested flow 1-9 have a throughput of approximately 2.2 Gbps because they share both the upstream (link 1) and downstream link (link 5) during the congestion after 13s. The victim flow 2-11 suffers severely by the HOL blocking even though it is not communicating with the hot-spot. Moreover, link 1 and 5 are QDR links where victim flow should be able to achieve a bandwidth of 12.9 Gbps.

With dFtree, the victim flow (2-11) recovers to 12.9 Gbps after the congested flows are reassigned to the *'slow lanes'*. Furthermore, both congested flows 1-9 and 5-9 are transmitting at 3.4 Gbps because flow 2-11 is no longer sharing resources with the congested flow 1-9 after the *'slow lane'* assignment.

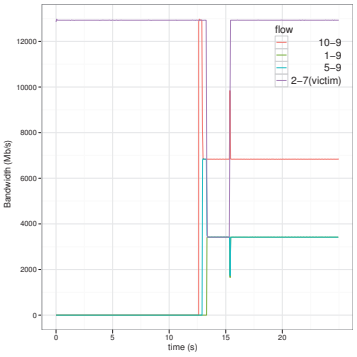
To summarise, the dFtree algorithm reduces the negative effect of HOL blocking when applied to an *oversubscribed fat-tree*. Fig. 12 shows that dFtree increases 468% from 2.2 Gbps to 12.9 Gbps for the victim flow in the worst case scenario during the congestion.

5.3 HPC Challenge Benchmark (HPCC)

In this experiment, we replaced the victim flows with the *HPC challenge* benchmark b_eff test suite [1]. The endpoint hot-spot is created using Perftest by running the traffic pattern presented in Fig. 3a for the non-oversubscribed topology and in Fig. 3b for the 2:1 oversubscribed topology. Simultaneously, we are running the HPCC benchmark in order



(a) Per flow throughput without dFtree.



(b) Per flow throughput with dFtree.

Figure 9: Experiment III using scenario in Fig. 3b.

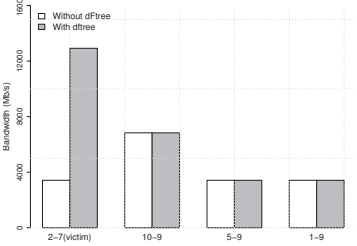
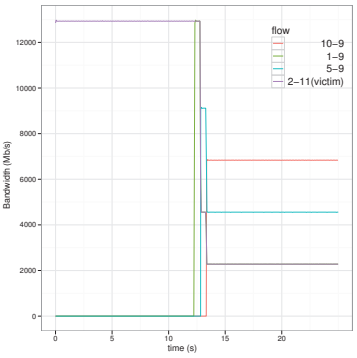
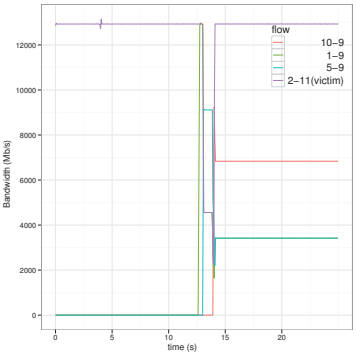


Figure 10: Per flow worst case bandwidth during congestion in Experiment III.



(a) Per flow throughput without dFtree.



(b) Per flow throughput with dFtree.

Figure 11: Experiment IV using scenario in Fig. 3b.

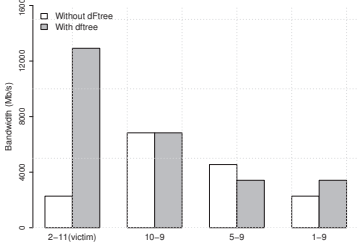


Figure 12: Per flow worst case bandwidth during congestion in Experiment IV.

to study the impact of congestion on the traffic generated by the HPCC benchmark. Even though the congested flows are still synthetically generated, this scenario resembles the network environment that an application could experience during congestion.

Table 2 shows the comparison of the HPCC b_eff results with and without the dFtree algorithm in the presence of congestion in a non-oversubscribed network. The most interesting observation is that the randomly ordered ring bandwidth increased by 49.34% with dFtree using only 2 VLs. We can see the improvement for all the latency and bandwidth tests, which is expected, as they correspond to the synthetic traffic patterns experiment that was carried out in the previous section. The results for the oversubscribed network are presented in Table 3 and the same trends are visible as for the non-oversubscribed network. These results clearly illustrate the performance gain with dFtree from the application traffic pattern's point of view.

5.4 Simulation Results

The main purpose of the simulation study is to show that the dFtree algorithm scales, and that the trends correspond to our cluster experiments. Another purpose of the simulation is to show that if the congested flows are separated from the normal traffic flows in a large network, this would increase the network performance by avoiding the negative impact of HOL blocking. We performed the simulation on a fully populated 648-port fat-tree as shown in Fig. 4 with 1, 3 and 9 hot-spots.

In our simulator, we modified the packet generator to always allocate a different VL (*'slow lane'*) for the packets that are directed to the hot-spot. As a result, the simulator isolates the hot-spot flows from the regular flows and reduce the possibilities of HOL blocking. Our simulation shows the best case results because it does not simulate the additional management overhead in a real cluster required to identify hot-spots and to migrate congested flows to a different virtual lane.

For a single hot-spot scenario, node 1 was the hot spot, and all the other nodes in the subnet were the contributors to this hot-spot. In case of three hot-spots, nodes 1 , 217, and 433 were the hot-spots and the contributors were the node group 1-216, 217-432, and 433-648 respectively. For a nine hot-spot scenario, the hot-spots were nodes 1, 73, 145, 217, 289, 361, 433, 505 and 577 whereas the contributors consists of node group 1-72, 73-144, 145-216, 217-288, 289-360, 361-432, 433-504, 505-576 and 577-648 respectively. For the abovementioned scenarios, the contributors sent 5% of their traffic to the hot-spot and remaining 95% to a randomly chosen node in the subnet.

In Fig. 13, we observe that a single hot-spot dramatically decreases the average throughput per node because of the large number of victim flows. There are a least 32 nodes (5%) contributing to the same hot-spot at any point in time. If more hot-spots are added, the contributor traffic is localised. Consequently, the impact on victim flows are reduced and the throughput per node increases. For the same reason, the relative improvement of the dFtree algorithm is reduced when the number of hot-spots increases. The relative improvement with the dFtree algorithm is 480.25% for 1 hot-spot, 345.32% for 3 hot-spots, and 169.17% for 9 hot-spots. If the number of hot-spots are increased in the same trend until it reaches 36 hot-spots, the improvement will be

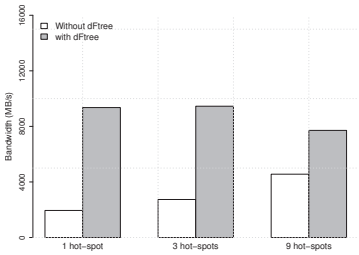


Figure 13: Simulation results for 648-switch.

minimal. This is due to the fact that the hot-spots are becoming increasingly more localised until each leaf switch has a hot-spot. Thus, the traffic pattern naturally splits the congested flows from the uncongested flows and increase overall network performance.

6. CONCLUSION AND FUTURE WORK

In this paper, we demonstrated the basic concept of dynamic networking in InfinBand by combining the performance manager and the subnet manager. By applying this concept to several congestion scenarios in a fat-tree topology, we are able to improve the performance using only 2 VLs, a *fast lane* for normal flows and a *slow lane* for congested flows. During the congestion, the performance manager is responsible for identifying the hot-spot flows and our host side dynamic reconfiguration mechanism is used to dynamically reassign flows into a seperate VL (*slow lane*). Our implementation in OpenSM achieved a 52.60% improvement in throughput on a cluster experiment and 480.25% improvement in a large-scale simulated environment when compared with the conventional fat-tree routing.

In the future, we plan to merge this work with the InfiniBand congestion control mechanism that uses the forward explicit congestion notification (FECN) to determine the hot-spot and backward explicit congestion notification (BECN) to identify its contributors. This combination can detect the hot-spot flows faster and it is independent of the performance sweeping interval. Furthermore, we can also remove the need for source throttling of the contributors in the IB congestion control mechanism.

7. REFERENCES

[1] HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/.

[2] The OpenFabrics Alliance. http://openfabrics.org/, Sept. 2010.

[3] Top 500 supercomputer sites. http://www.top500.org/, Nov. 2010.

[4] B. Bogdanski et al. Achieving Predictable High Performance in Imbalanced Fat Trees. In *Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS'10) - to appear*, 2010.

[5] W. J. Dally and B. Towles. *Principles and practices of interconnection networks*, chapter 15.4.1, pages 294–295. Morgan Kaufmann, 2004.

Table 2: Results from the HPC Challenge benchmark with and without dFtree for experiment in Fig. 3a.

Network latency and throughput	a) without dFtree	b) dFtree	c) Improvement
Min Ping Pong Lat. (ms)	0.002131	0.001878	11.87%
Avg Ping Pong Lat. (ms)	0.026146	0.005665	78.33%
Max Ping Pong Lat. (ms)	0.055388	0.012994	76.54%
Naturally Ordered Ring Lat. (ms)	0.023699	0.007200	69.62%
Randomly Ordered Ring Lat. (ms)	0.027727	0.007469	73.06%
Min Ping Pong BW (MB/s)	336.331	539.374	60.37%
Avg Ping Pong BW (MB/s)	592.416	970.024	63.74%
Max Ping Pong BW (MB/s)	1589.203	1589.203	0.00%
Naturally Ordered Ring BW (MB/s)	383.843326	527.593704	37.45%
Randomly Ordered Ring BW (MB/s)	338.329033	505.272445	49.34%

Table 3: Results from the HPC Challenge benchmark with and without dFtree for experiment in Fig. 3b.

Network latency and throughput	a) without dFtree	b) dFtree	c) Improvement
Min Ping Pong Lat. (ms)	0.001997	0.001997	0.00%
Avg Ping Pong Lat. (ms)	0.010495	0.003995	61.93%
Max Ping Pong Lat. (ms)	0.041634	0.012934	68.93%
Naturally Ordered Ring Lat. (ms)	0.028419	0.007796	72.57%
Randomly Ordered Ring Lat. (ms)	0.031403	0.007721	75.41%
Min Ping Pong BW (MB/s)	358.235	554.179	54.70%
Avg Ping Pong BW (MB/s)	1088.153	1170.939	7.61%
Max Ping Pong BW (MB/s)	1590.408	1590.559	0.01%
Naturally Ordered Ring BW (MB/s)	413.114906	511.079782	23.71%
Randomly Ordered Ring BW (MB/s)	338.930349	517.198255	52.60%

[6] J. Escudero-Sahuquillo et al. An Efficient Strategy for Reducing Head-of-Line Blocking in Fat-Trees. In D'Ambra, Pasqua And Guarracino, Mario And Talia, Domenico, editor, *Lecture Notes in Computer Science*, volume 6272, pages 413–427. Springer Berlin / Heidelberg, 2010.

[7] C. Gómez et al. Deterministic versus Adaptive Routing in Fat-Trees. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. IEEE CS, 2007.

[8] E. G. Gran et al. First Experiences with Congestion Control in InfiniBand Hardware. In *Proceeding of the 24th IEEE International Parallel & Distributed Processing Symposium*, 2010.

[9] E. G. Gran and S.-A. Reinemo. Infiniband congestion control, modelling and validation. In *4th International ICST Conference on Simulation Tools and Techniques (SIMUTools2011, OMNeT++ 2011 Workshop)*, 2011.

[10] W. L. Guay, B. Bogdanski, S.-A. Reinemo, O. Lysne, and T. Skeie. vftree - a fat-tree routing algorithm using virtual lanes to alleviate congestion. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium*, 2011.

[11] W. L. Guay and S.-A. Reinemo. A scalable method for signalling dynamic reconfiguration events with opensm. In R. Buyya, editor, *11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2011)*, pages 332 – 341. IEEE Computer Society Press, 2011.

[12] 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.

[13] T. Hoefler et al. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Cluster Computing, 2008 IEEE International Conference on*, pages 116–125, 2008.

[14] InfiniBand Trade Association. *InfiniBand architecture specification*, 1.2.1 edition, November 2007.

[15] G. Pfister et al. Solving Hot Spot Contention Using InfiniBand Architecture Congestion Control, July 2005.

[16] G. F. Pfister and A. Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, 1985.

[17] G. Rodriguez et al. Exploring pattern-aware routing in generalized fat tree networks. In *Proceedings of the 23rd international conference on Supercomputing*, pages 276–285, New York, 2009. ACM.

[18] G. Rodriguez et al. Oblivious Routing Schemes in Extended Generalized Fat Tree Networks. *IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER '09.*, pages 1–8, 2009.

[19] A. Vishnu, M. Koop, and A. Moody. Topology agnostic hot-spot avoidance with InfiniBand. *Concurrency and Computation: Practice and Experience*, 21(3):301–319, 2009.

[20] E. Zahavi et al. Optimized InfiniBand TM fat-tree routing for shift all-to-all communication patterns. *Concurrency and Computation: Practice and Experience*, 22(2):217–231, 2009.

Paper V

Early Experiences with Live Migration of SR-IOV enabled InfiniBand

Wei Lin Guay, Sven-Arne Reinemo, Bjørn Dag Johnsen, Chien-Hua Yen, Tor Skeie, Olav Lysne and Ola Torudbakken

Early Experiences with Live Migration of SR-IOV enabled InfiniBand

Wei Lin Guay[a,b,1,*], Sven-Arne Reinemo[b], Bjørn Dag Johnsen[a], Chien-Hua Yen[c], Tor Skeie[b], Olav Lysne[b], Ola Torudbakken[b]

[a] *Oracle Corporation, Oslo, Norway*
[b] *Simula Research Laboratory, Lysaker, Norway*
[c] *Oracle Corporation, California, United States*

Abstract

Virtualization is the key to efficient resource utilization and elastic resource allocation in cloud computing. It enables consolidation, the on-demand provisioning of resources, and elasticity through live migration. Live migration makes it possible to optimize resource usage by moving virtual machines (VMs) between physical servers in an application transparent manner. It does, however, require a flexible, high-performance, scalable virtualized I/O architecture to reach its full potential. This is challenging to achieve with high-speed networks such as InfiniBand and remote direct memory access enhanced Ethernet, because these devices usually maintain their connection state in the network device hardware. Fortunately, the single root IO virtualization (SR-IOV) specification addresses the performance and scalability issues. With SR-IOV, each VM has direct access to a hardware assisted virtual device without the overhead introduced by emulation or para-virtualization. However, SR-IOV does not address the migration of the network device state. In this paper we present and evaluate the first available prototype implementation of live migration over SR-IOV enabled InfiniBand devices.

Keywords: IO virtualization, VM migration, SR-IOV, Architecture

*Corresponding author

Email addresses: wei.lin.guay@oracle.com (Wei Lin Guay), svenar@simula.no (Sven-Arne Reinemo), bjorn-dag.johnsen@oracle.com (Bjørn Dag Johnsen), chien.yen@oracle.com (Chien-Hua Yen), tskeie@simula.no (Tor Skeie), olavly@simula.no (Olav Lysne), ola.torudbakken@oracle.com (Ola Torudbakken)

1. Introduction

Cloud computing provides the illusion of an infinite number of computing and storage resources without the need to worry about their over- or under-subscription of resources [1]. This is enabled by virtualization, which plays an essential role in modern data centres, providing elastic resource allocation, efficient resource utilization, and elastic scalability.

Virtualization has been rapidly adopted by the industry during the last decade and is now used in all sorts of computing devices, from laptops to desktops to enterprise servers. The initial reluctance to adopt virtualization had its roots in poor performance for CPU, memory and I/O resources. The performance for CPU and memory resources has, however, improved significantly [2, 3] and is now in such an advanced state that even the high performance computing community is interested in exploiting virtualization [4, 5, 6]. The final performance bottleneck for virtualization is the virtualization of I/O resources. Recent progress has improved the performance of *I/O virtualization* (IOV) through the introduction of *direct assignment* [7] and *single root I/O virtualization* (SR-IOV) [8]. Direct assignment allows a *virtual machine* (VM) to bypass the *VM monitor* (VMM) and directly access the device hardware, which gives close to native performance [9, 10, 11, 12]. Unfortunately, it has one major drawback: its lack of scalability. Only one VM can be attached to a device at a time; thus multiple VMs require multiple physical devices. This is expensive with regards to the number of devices and internal bandwidth. To rectify this situation, SR-IOV was developed to allow multiple VMs to share a single device by presenting a single physical interface as multiple virtual interfaces. Recent experience with SR-IOV devices has shown that they are able to provide high–performance I/O in virtualized environments [13, 14, 15].

A key feature provided by virtualization is the *live migration* of VMs, that is, moving running VMs from one machine to another without disrupting the applications running on them [16, 17, 18]. Live migration is a powerful tool that extends the management of VMs from a single physical host to an entire cluster. It provides for the flexible provisioning of resources and increased efficiency through on–demand server consolidation, improved load balancing, and simpler maintenance. Live migration has been demonstrated for both IOV with emulation over Ethernet [16] and IOV with direct assignment over InfiniBand (IB) [17, 18]. Compared to conventional Ethernet devices or migration based on emulation, high–performance and lossless technologies such

as IB have more resources managed by the network interface hardware, which makes live migration more complex.

In this paper we propose a design for the transparent live migration of VMs over high-speed and lossless SR-IOV devices. This design is prototyped and evaluated using Mellanox–based IB hardware and the Xen-based Oracle Virtual Machine virtualization platform. We describe the implementation and measure the service downtime of an application during migration. Through a detailed breakdown of the different contributors to service downtime, we pinpoint the fraction of the cost of migration that can be mitigated by architectural changes in future hardware. Based on this insight, we propose a new design and argue that, with these changes, the service downtime of live migration with IB SR-IOV devices can be reduced to less than a second, as demonstrated with conventional Ethernet SR-IOV devices [19].

This paper is organized as follows: Section 2 describes the necessary background on IOV and related technologies. Related works are discussed in Section 3. This is followed by a description of the challenges of live migrating a VM over an SR-IOV–enabled IB device in Section 4. Then, Section 5 proposes a design addressing the challenges presented in Section 4 using the current SR-IOV and IB architecture. Section 6 presents a performance evaluation based on measurements from our design prototype. Section 7 discusses the remaining challenges and proposes an architecture for providing seamless live migration over IB. Finally, Section 8 concludes the paper.

2. Background

This section gives an overview of virtualization, IOV technologies, and the IB architecture.

2.1. Virtual Machine Monitors

A VMM, also known as a hypervisor, is a software abstraction layer that allows multiple VMs, that is, operating system instances, to run concurrently on a physical host. A VMM can be categorized into type I and type II [20]: A type I VMM runs directly on the host hardware, has exclusive control over the hardware resources, and is the first software to run after the boot loader. The VMs run in a less privileged mode on top of the VMM. Well–known type I VMMs include the original CP/CMS hypervisor, VMware ESXi, Microsoft Hyper-V, and Xen. A type II VMM runs as a privileged process on top of a

conventional operating system and the VMs run on top of this privileged process. The type II VMM controls and schedules access to hardware resources for the VMs. Well–known type II VMMs include VMware GSX server, KVM, and VirtualBox. In this paper we use the Oracle Virtual Machine 3.0, which is a type I VMM built on Xen.

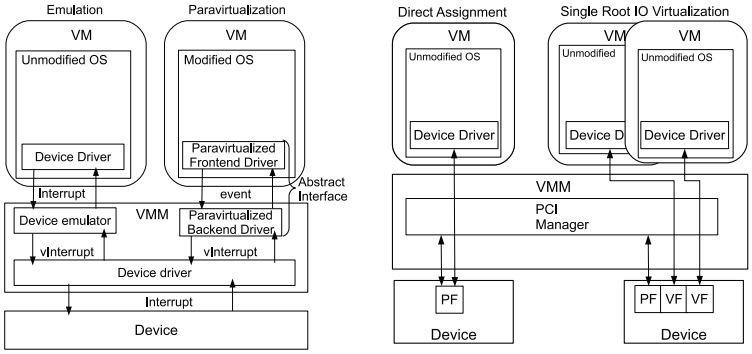
2.2. I/O Virtualization

I/O Virtualization (IOV) was introduced to allow VMs to access the underlying physical I/O resources. With the adoption of virtualization came an increase in the number of VMs per server, which has made I/O a major bottleneck. This is due to two reasons: First, the available I/O resources at the physical level has not scaled with the number of cores. This problem is being addressed by the introduction of new versions of PCI Express (PCIe) that increase internal system bandwidth and new versions of IB and Ethernet that increase external network bandwidth. Second, a physical server handling multiple VMs must support the I/O requests from all these VMs and, due to the initially low I/O performance of even a single VM, this significantly reduces I/O performance. In practice, the combination of storage traffic and inter-server communication imposes an increased load that can overwhelm the I/O resources of a single server, leading to backlogs and idle processors waiting for data.

With the increase in the number of I/O requests, the goal of IOV is not only to provide availability, but to improve the performance, scalability, and flexibility of the I/O resources to match the level of performance seen in CPU virtualization. The following sections describe the recent progress made in IOV, the concepts applied, and the benefits and drawbacks of the existing approaches.

2.2.1. Emulation

Virtualization through emulation is an IOV technique in which a physical I/O device is emulated through software [7]. The VMM, as shown in Fig. 1a, is responsible for the emulation and must expose an *emulated device* that can be discovered and that can perform data transfers on behalf of the VMs. The emulated device can be used by existing drivers and it is straightforward to support decoupling and multiplexing. Furthermore, the emulation mechanism can support different operating systems without any changes in the VMM and can easily support VM migration, since the emulated device in the VM is decoupled from the physical device on a given host.



(a) (b)

Figure 1: (a) Emulation versus paravirtualization. (b) Direct assignment versus SR-IOV.

The major drawback with emulation is the lack of performance and scalability. Performance is reduced because the multiplexed interaction between the VMs and the emulated device needs to be trapped by the VMM, which then performs the operation on the physical device and copies the data back to the emulated device. The multiple layers of interaction, especially in interrupt handling, are expensive. As a result, the VMs increase the load on the emulated device and the I/O domain increases proportionally, which limits scalability significantly.

2.2.2. Paravirtualization

To reduce the performance bottleneck caused by emulation, Barham et al. suggested the use of paravirtualization [21]. Paravirtualization is an IOV technique that presents an abstract interface to the VM that is similar but not identical to the underlying hardware (Fig. 1a). The purpose of this abstraction layer is to reduce the overhead of the interaction between the VMs and the VMM. For example, rather than provide a virtual networking device, the VMM provides an abstract networking device together with an application programming interface where a callback function handles interrupts from a physical device instead of the traditional interrupt mechanism. Then the device driver in the VM is altered to call the application programming inter-

face to perform read and write operations. The paravirtualized device driver is split into two parts: the backend driver that resides in the VMM and the frontend driver that resides in the VM. Paravirtualization decreases the number of interactions between the VMs and the VMM compared to device emulation and allows for more efficient multiplexing. Paravirtualization also maintains the decoupling between the paravirtualized device and the physical device on a given host.

Even though paravirtualization improves performance compared to device emulation, its performance overhead is still substantial compared to native I/O access. In addition to the performance and multiplexing overhead, another drawback is that both the device driver and the operation system kernel in the VM must be modified to support the new abstractions required for paravirtualization.

2.2.3. Direct Assignment

To further reduce the I/O overhead present in paravirtualization Intel proposed direct assignment, also known as *device passthrough* [7, 9]. As shown in Fig. 1b, direct assignment provides exclusive access to a device for a given VM, which yields near native performance. It outperforms emulation and paravirtualization because it allows the VM to communicate with a device without the overhead of going through the VMM. But it requires hardware support to reach its full potential in improving performance. At the time of this writing, major processor vendors have equipped their high-end processors with instructions and logic to support direct assignment. This includes interrupt virtualization, address translation, and protection tables for direct memory access [22]. Interrupt virtualization allows the efficient delivery of virtual interrupts to a VM, while the I/O memory management unit (IOMMU) protects the memory assigned to a VM from being accessed by other VMs.

The drawbacks of direct assignment are that it does not support multiplexing or decoupling. Multiple VMs cannot have multiplexed access to a single physical device so the number of VMs can only scale by allocating as many passthrough devices as are physically present in the machine. Furthermore, live migration is complicated by the lack of decoupling. This is due to the fact that the device is directly assigned to a VM and the VMM has no knowledge of the device state [17].

2.2.4. Single Root IO Virtualization

The SR-IOV specification was created to address the lack of scalability with direct assignment [8]. This specification extends the PCIe specification with the means to allow direct access to a single physical device from multiple VMs while maintaining close to native performance. Currently SR-IOV capable network devices are available for both Ethernet and IB devices.

As depicted in Fig. 1b, SR-IOV allows a single PCIe device to expose multiple virtual devices so the physical device can be shared among multiple VMs by allocating one virtual device to each VM. Each SR-IOV device has at least one *physical function* (PF) and one or more associated *virtual functions* (VFs). A PF is a normal PCIe function controlled by the VMM, whereas a VF is a light-weight PCIe function. Each VF has its own memory address and is assigned a unique request ID that enables the IOMMU to differentiate between the traffic streams from different VFs. The IOMMU also provides memory and interrupt translations between the PF and the VFs.

The use of an SR-IOV device yields close to native performance and improved scalability, as shown by [15]. But it shares the drawback of not able to live migrate and, for the same reasons, with direct assignment.

2.3. Live Migration

Live migration is one of the key features of virtualization [16]. It makes it possible to migrate a running VM, together with the applications, from one physical host to another with minimal downtime and transparent to the running applications. Thus, it is possible to dynamically relocate VMs in response to system management needs such as on-demand server consolidation, hardware and software maintenance, system failure, and data centre scale-out requirements.

A key requirement for live migration is a fast and flexible network architecture. The network must be fast to quickly migrate a VM and it must be flexible to quickly reestablish the network connection when the VM is restarted in its new location. From the IOV point of view, migrating an emulated or paravirtualized device only requires that an identical physical device be present at the destination host. The device configuration is maintained on the destination host since the migrated VM contains the internal state of the emulated device. Live migration is more difficult for SR-IOV and direct assignment devices because the dependency between the VM and the physical device at the source host cannot be easily maintained or recreated at the destination host. We elaborate on these challenges in Section 4.

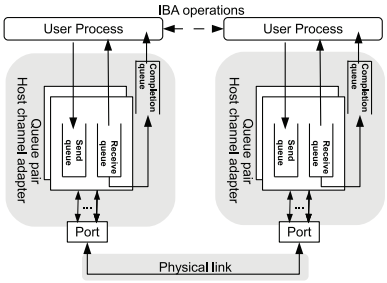


Figure 2: Conceptual diagram of IB communication using QPs.

2.4. The IB Architecture

The IB architecture [23] is a serial point-to-point full-duplex network technology that was first standardized in October 2000 as a merger of the two technologies Future I/O and Next Generation I/O. An IB network is referred to as a subnet, where the subnet consists of a set of hosts interconnected using switches and point-to-point links. An IB subnet requires at least one subnet manager, which is responsible for initializing and bringing up the network, including the configuration and address assignment for all the switches and *host channel adaptors* (HCAs) in the subnet. Switches and HCAs within a subnet are addressed using *local identifiers* (LIDs) and a single subnet is limited to 49,151 unicast LIDs.

To provide both remote direct memory access (RDMA) and traditional send/receive semantics, IB supports a rich set of transport services. Independent of the transport service used, all IB HCAs communicate using queue pairs (QPs), as shown in Fig. 2. A QP is created by the *communication manager* during the communication setup and supplies a set of initial attributes such as the QP number (QPN), HCA port, destination LID, queue sizes, and transport service. An HCA can contain many QPs and at least one QP is present at each end node in the communication. A QP consists of a *send queue* (SQ) and a *receive queue* (RQ). The SQ holds work requests to be transferred to the remote node, while the RQ holds information on where to direct the incoming data. In addition to the QPs, each HCA has one or more completion queues associated with a set of SQs and RQs. The completion

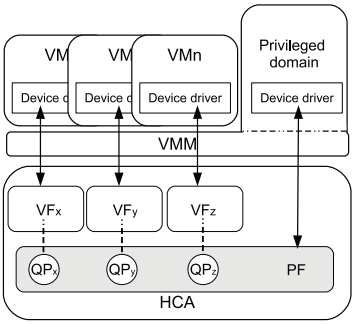


Figure 3: The *shared port* model of the SR-IOV implementation for IB.

queue holds completion notifications for the work requests posted to the SQ and RQ. Although the complexities of the communication are hidden from the user, the QP state information is stored in the HCA.

The implementation for SR-IOV IB is based on a *shared port* model, as shown in Fig. 3. This design is used in Mellanox ConnectX hardware. In the shared port model all the VFs share a single LID and a single QP name-space, and only a single HCA port is discoverable by the network. The single QP name-space complicates migration because the QP attributes cannot be easily reused. This is discussed further in Section 4.

3. Related Work

Several efforts to enable live migration over IB devices have been proposed by reusing the solutions used in Ethernet devices. One approach is to create a paravirtualized device driver that hides the complexity of IB from the guest VMs. Ali et al. from Mellanox proposed an Ethernet service over IPoIB (eIPoIB) [24], which is a paravirtualized driver for IPoIB device. The eIPoIB implementation is a shim layer that performs translation between an IPoIB header and an Ethernet header in the PF driver. The IPoIB is an ULP protocol that provides IP services over IB fabric and the failover mechanism can depend on TCP for retransmitting loss packets. In this way, an IPoIB device can be provisioned as a network device for guest VMs, similar to the conventional ethernet devices. Another approach is vRDMA,

which is a project proposed by VMware [25]. This approach creates an RDMA paravirtualized device in the privileged domain, which is based on the IB verbs. Both eIPoIB and vRDMA allow migration because the VMM is aware of the device state. Nevertheless, these approaches sacrify the high throughput and low latency properties of IB in order to achieve transparent migration.

Efforts to enable live migration over SR-IOV devices have been limited to conventional Ethernet devices. Due to the architectural differences between Ethernet and IB, these efforts are not applicable to IB. Zhai et al. [26] proposed an approach that combines the PCI hot plug mechanism with the Linux bonding driver. This solution is a workaround for the requirement of migrating the internal state of an SR-IOV device by quiescing the device before migration. The VF is detached when the migration starts and then reattached after the migration is finished. During the migration the bonding driver maintains network connectivity using a secondary device. A general limitation of this approach is that it depends on a secondary network interface residing in the privileged domain. Another limitation specific to IB is that the bonding driver does not support IB native operations such as RDMA. Although IB can be bound by using upper layer protocols such as IPoIB or EoIB in combination with the Linux bonding driver, using these upper layer protocols (ULPs) significantly reduces overall network performance due to the overhead of multiple protocol stacks. Another proposal is the *network plugin architecture* (NPIA) [27]. This architecture creates a shell in the kernel of the running VMs, which is used to plug or unplug an abstract hardware device during run time. The NPIA can also use a software interface as a backup device, but this requires a complete rewrite of the network drivers. This has prevented NPIA from being widely employed. Kadav et al. [28] proposed using a shadow driver to monitor, capture, and reset the state of the device driver for the proper migration of direct assignment devices. In a manner similar to that of Zhai et al. [26], this approach hot unplugs the device before migration and hot plugs the device after migration. It then uses the shadow driver to recover the device state after migration. The concept of a shadow driver, however, does not scale in an IB context because the state of each QP must be monitored and captured, which can amount to tens of thousands of QPs. The ReNIC proposed by Dong et al. [19] suggests an extension to the SR-IOV specification where the internal VF state is cloned and migrated as part of the VM migration. That treatment of the subject, however, was limited to conventional Ethernet devices. In IB, the migration

of a VF must include its internal state and the internal state of the QP. This is discussed in Section 4.2.

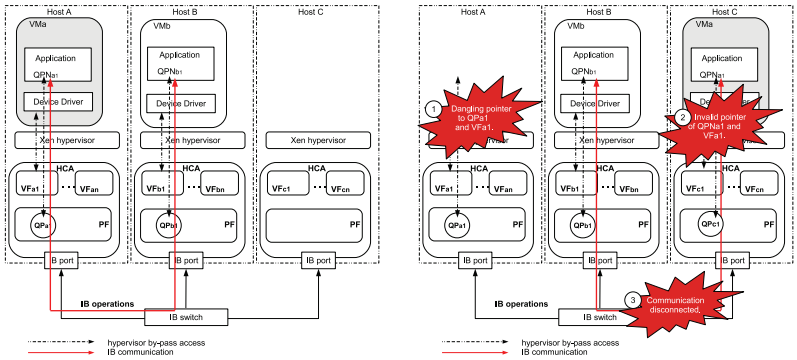
Apart from the abovementioned efforts in enabling live migration with SR-IOV Ethernet, several previous works that focus on process migration are also relevant to our effort in enabling live migration with SR-IOV IB [29, 30]. Although process migration differs from VM migration, in principle, both face the same problem. Before migration, the resources of active processes must be released. Then a new set of resources must be reallocated after migration is completed.

4. Challenges

The challenge of live migrating an SR-IOV enabled IB device is that the hardware architecture for both SR-IOV and IB must be considered. For SR-IOV devices, the internal device state of a VF must be migrated along with the VM. For IB devices, the active QPs must be recreated at the destination server after VM migration. In addition, the architecture of the SR-IOV model for IB, as shown in Fig. 3, must also be considered. Currently, the available SR-IOV enabled IB is based on the shared port model and none of the approaches mentioned in Section 3 can be directly applied to it. The key to a proper solution for IB requires the correct handling of active QP resources and correct reallocation of these resources after migration. Otherwise, the IB communication will be broken, as shown in Fig. 4. To summarize, to migrate an IB VF we have to handle the i) *detachment of an active device*, the ii) *reallocation of physical resources*, and the iii) *reestablishment of the remote connection*. In the remainder of this section, we discuss in detail each of these challenges.

4.1. Detachment of an Active Device

As part of the migration process, the VM being migrated must be detached from the device it is directly assigned to and it must be reattached to a new device at its destination. If we use the PCI hot plug mechanism suggested by [26] to quiesce the IB device by detaching the VF before live migration, migration of the internal state of the VF is not required. But any attempt to detach an IB VF with active QPs that have ongoing data transfers will fail. The reason is that the process memory of an active QP is pinned to physical memory for the duration of the transfer to reduce data movement operations in the host (e.g.,Fig. 4 shows that the application can



(a) Before VM migration.

(b) After VM migration.

Figure 4: IB operations are performed between VMa and VMb, hosted by host A and host B, respectively, as shown in Fig. 4a. After VMa is migrated to host C, the communication is broken due to the three reasons cited in Fig. 4b. First, $VFa1$ and $QPa1$ are not being detached from VMa(label 1 in Fig. 4b). Second, the $QPNa1$ opaque pointer cached by the application in VMa has become an invalid pointer. This is because VMa is no longer associated with $VFa1$ or $QPa1$ (label 2 in Fig. 4b). Third, the established IB connection between the applications in VMa and VMb based on $QPNa1$ and $QPNb1$, respectively, is disconnected because $QPNa1$ is no longer valid (label 3 in Fig. 4b).

access the HCA directly). Therefore, it is not possible to detach an IB VF with active data transfers using the PCI hot plug mechanism.

The main problem is that there is no transparent mechanism yet to migrate a VF from one physical server to another. Thus, an active VF must be detached before migration. In this case, there must be an interaction between the user process and the kernel about the attempt to detach the VF. This can be fixed by defining an interface between the user space and the kernel space that allows ongoing data transfers to be paused until migration is complete. Only then can we properly support the live migration of RDMA devices.

4.2. Reallocation of Physical Resources

All IB connections are composed of QPs and each QP is managed in hardware by the HCA. The QP attributes consist of various resources that can be generalized into *location–dependent resources* and *connection state information*. Both types of resources can only be accessed by software through an opaque pointer (e.g., the $QPNa1$ used by the application, as depicted in Fig. 4), which becomes an obstacle when we try to migrate QPs between different physical hosts.

4.2.1. Location–Dependent Resources

A QP contains the following location-dependent resources: the LID and the QPN associated with the HCA where the QP was created. In addition, the remote key (RKey) and local key (LKey) are indirectly dependent on a QP because these keys are associated with the physical memory registered for a QP. These QP–associated resources are explained in detail as follows.

1. The LID is the address that is assigned to an HCA by the subnet manager. With the current implementation of SR-IOV, based on the shared port model, all the VFs on one HCA share a single LID with its PF. As a result, it is not feasible to change the LID assigned to a VF during migration because this will affect all the VFs on the same HCA.
2. The QPN is the identifier that represents a QP but it is only unique within an HCA. Thus, the same QPN can refer to another QP in another HCA. As a result, improper handling of a QPN after migration can end up in a conflict of QPNs.

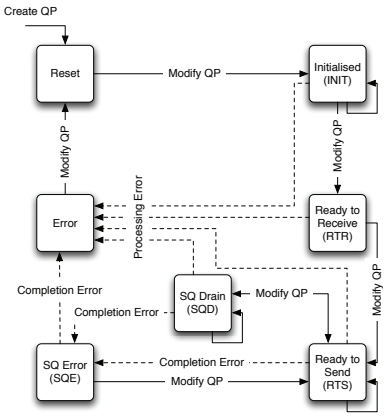


Figure 5: The QP state diagram.

3. The LKey and RKey are associated with a memory region registered by a QP. These keys are created by the HCA upon creation of the QP, are only unique within the HCA, and are used to authorize access to local and remote memory regions, respectively. After migration, even though the same memory regions are allocated, a new LKey and RKey are generated by the HCA upon creation of a new QP in the new location. Therefore, the previous RKey presented to the peer QP becomes invalid and must be updated after migration.

4.2.2. Connection State Information

In addition to location–dependent resources, we also have to migrate the connection state of the IB connection. The connection state includes the QP state and the SQ and RQ states. When a QP is created, it goes through the reset (RESET), -initialized(INIT), -ready-to-receive(RTR), and -ready-to-send(RTS) state transitions, as show in Fig. 5. The migration of a QP includes the migration of any outstanding packets residing in the SQ of the old QP. When a QP has been migrated to its destination server it must be transitioned back to the RTS state before communication can be resumed. A QP is only considered fully operational when it has reached the RTS state. Unfortunately, the IB specification does not define states for *suspending* and

resuming a QP before and after migration, which complicates the migration process due to ongoing communication and the necessity of avoiding out-of-order packets and dropped packets.

4.3. Reestablishment of the Remote Connection

While IB supports both reliable and non-reliable transport services, this work focuses on the former. When a QP is created, it is not associated with any other QPs and must be connected to a remote QP through a connection establishment process to transfer data. The communication manager uses an out-of-band channel to establish a connection between two QPs, exchanging QP attributes, and transitions both QPs into the RTS state. If migration occurs when both QPs are in the RTS state, the QP to be moved is destroyed and an event is sent to the remote QP to disconnect its connection. The challenge is to recreate an identical QP at the destination and to reestablish the connection with the remote QP after the migration is complete.

5. Design and Implementation

Two different strategies enable live migration of SR-IOV devices over IB: the *non-transparent* approach and the *transparent* approach. The non-transparent approach assumes that the complexity of the low-level device drivers is a black box. Reconfiguration of the underlying network structures during live migration is handled by the ULP. The selected ULP must be robust enough to handle the challenges mentioned in Section 4, such as the reallocation of physical resources and reestablishment of a remote connection. The Oracle Reliable Datagram Socket (RDS) or IP over IB (IPoIB) are the examples of ULP that fulfils these requirements. As a result, it can be combined with the hot plug mechanism to migrate a VM independent of the underlying network technology[IPoIBref][rdsref]. Performance might suffer, however, and the *non-transparent* approach is not a generic solution because the application depends on a dedicated ULP that increases the overhead of data movement to support migration. Thus, we propose a transparent approach to provide a generic solution that supports the IB transport services.

Our transparent design leverages the hot plug mechanism by detaching the VF to quiesce the device. Then, motivated by previous work on process migration [29, 30], we release the hardware managed resources of the active QPs before the migration starts and reallocate them after the migration is complete. Since the VM migration consists of three separate stages,

we named our transparent design the three-stage migration process. In the following sections, we discuss the design choices, the details of our implementation, and how our design addresses the challenges described in Section 4.

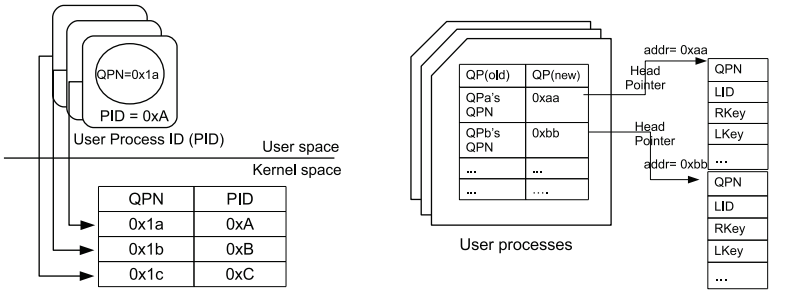
5.1. Detachment of an Active Device

Recall from Section 4.1 that an IB VF with active QPs and ongoing data transfers will fail to be detached using the hot plug mechanism. However, if the user process that owns the QP can be temporarily suspended, then the VF will become quiescent and it can be detached after the kernel device driver instances are. Therefore, we propose a method that provides interaction between the user processes and the drivers in the kernel. This method includes the following steps after the VMM decides to migrate a given VM: i) The VMM notifies the kernel in a given VM about the migration, ii) the kernel initiates the device detachment process and requests that all user processes with active QPs suspend operation, iii) the user processes suspend their QPs, which is discovered by the kernel through polling on the device driver, and iv) the kernel completes the detachment of the device and the migration process proceeds.

For the kernel to know about the QPs in use by a given process, we introduce a process–QP mapping table (MT) as shown in Fig. 6a. The table maps between the process identification (PID) and the QPN. It is implemented as a kernel module in the privileged domain that tracks the association between the PID and the QPN. As shown in Algorithm 1, the PID is registered in the mapping table when a QP is created and the PID is unregistered when the QP is destroyed. If the table contains a matching PID when the privileged domain attempts to detach a VF, the process registered to the QP using the VF is notified. When this notification is received by the process with the given PID, the callback function, implemented in the user library, releases the pinned memory. As a result, the PCI drivers can be unloaded successfully and the VF can be detached from the VM. To prevent any send/receive events during migration, the user process is suspended until it receives the resume event signal.

5.2. Reallocation of Physical Resources

When a VM is migrated to another physical server, the physical resources used by the QPs in the VM must be reallocated. Recall from Section 4.2 that



(a) The QPN–PID MT that resides in the VM kernel.

(b) The user space QPN MT that maps the QPN to the QP context through the linked list.

Figure 6: The Mapping Table (MT) implementation.

Algorithm 1 Registration of the PID–QPN mapping.

if *ib_module_loaded* **then**
  if $ib\_create\_qp(pid\_from\_user, qpn)$ **then**
    $reg\_to\_pid\_qpn\_t(pid\_from\_user, qpn)$
  else if $ib\_destroy\_qp(pid\_from\_user, qpn)$ **then**
    $unreg\_from\_pid\_qpn\_t(pid\_from\_user, qpn)$
  end if
else if *unloading_ib_module* **then**
  if $pid\_qpn\_t\_not\_empty()$ **then**
    $notify\_user\_callback\_func(pid\_from\_user)$
  end if
end if

each QP contains location–dependent resources and a connection state managed by the HCA hardware. The user application accesses these resources through an opaque pointer that references the physical resources.

5.2.1. Location–Dependent Resources

The location-dependent resources in a QP consist of the LID, the QPN, the RKey and the LKey. These resources must be recreated at the destination server and, to ensure application transparency, remapped so they are reachable through the opaque pointer that is cached in the user application.

The complexity lies in how to remap these resources. The most efficient approach is to offload the remapping mechanism to the hardware, but hardware offloading requires modifications to the existing hardware and software architecture, as discussed further in Section 7. As an alternative approach compatible with today's hardware architecture, we propose placing the MT in the user space library. By putting it here, we avoid any conflicts between QPNs that might have occurred if this had been handled by the kernel. The MT is per process, which reduces the size of each table and minimizes the time to search and retrieve the updated QP attributes. Furthermore, instead of using one mapping table for each attribute, we create a linked list of pointers to the new QP attributes, as shown in Fig. 6b. The QPN is used as the key to retrieve the updated QP attributes (QPN, LID, RKey, and LKey) after a new IB VF is attached to the migrated VM.

5.2.2. Connection State Information

The IB specification does not define the states for suspending and resuming a QP, as shown in Fig 5. However, three approaches can be used when migrating a QP.

1. A *full cycle reset* can be carried out at any time by triggering the RTS–RESET transition, but this resets the QP attributes and clears the SQs, RQs, and completion queues. This means that the QP attributes and any data in the queues are lost.
2. The *automatic path migration* mechanism provided by IB can automatically switch a QP to a predefined secondary path without destroying the established connection. This only works, however, if the predefined path is located within the same HCA. Therefore, it is not useful for reconfiguring QPs across physical hosts.

3. The *SQ drain* (SQD) state is the only state, as shown in Fig. 5, that allows a QP transition to and from the RTS state. In other words, it is the only way to modify the QP attributes after the QP is in the RTS state. A QP can only be transition into the SQD state after the remaining elements in the SQ have been successfully executed.

From a comparison of these three approaches, it is clear that the SQD state is the only one suitable for reconfiguring QP attributes without losing connection state information. The SQD state requires that all the outstanding operations in the SQ be sent before the transition from the RTS state to the SQD state is allowed. Thus, the SQD state ensures that all the outstanding send operations are completed before a QP can be temporarily suspended. With this approach the QP is in a deterministic state where no in-flight packets exist and it can be safely suspended and migrated. Nevertheless, in our prototype we have to emulate the SQD state in software because it is not currently supported by the HCAs used in this experiment.

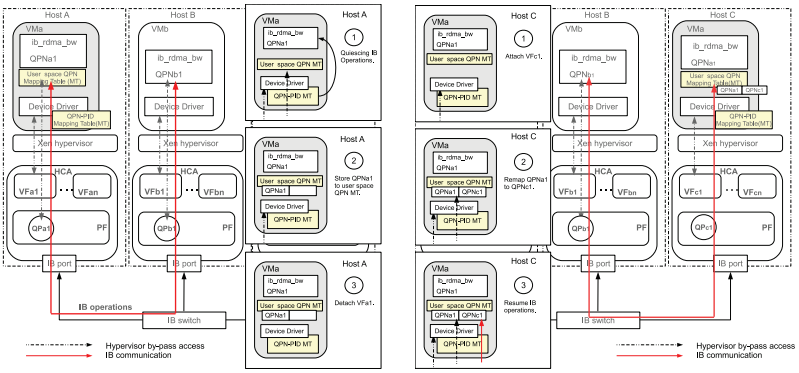
5.3. Reestablishment of a Remote Connection

To migrate a VM with an IB VF, the communication manager must be tolerant of the device detachment event. The communication manager closes the connection upon device removal and restarts the connection once a new device is reattached. The current communication manager implementation, however, is intolerant of the device removal event. It will disconnect the connection and destroy both QPs after a VF has been detached. Therefore, we implement the reconnection mechanism in the user space library in our prototype. Before the VF is detached, the migrating VM saves the QP attributes and the out-of-band connection address to reestablish a new connection with the same remote QP later.

During migration, the remote QP must be prevented from continuing to send data to the migrating VM. Hence, the migrating VM forwards a *suspend* message to notify the remote QP about the migration. After the remote QP receives the message, it must complete all the outstanding operations in the SQ. Then the remote QP is then transitions into the RESET state and waits for a new event to resume the communication.

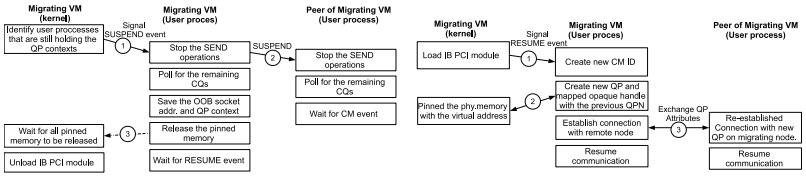
5.4. Summary of the Three-Stage Migration

In this section, we summarize the design of the three-stage migration process. For ease of explanation, we use a migration scenario as illustrated



(a) Before VM migration. (b) After VM migration.

Figure 7: The IB operations are performed between VMa and VMb, that is, hosted by host A and host B, respectively. With our prototype, which has added the user space QPN MT and the QPN–PID MT, the communication can be resumed, even after VMa is migrated to host C. Before migration, as shown in Fig. 7a, the QPN–PID MT informs the application that has an active QP to quiesce the IB operations. Then the opaque pointer, $QPNa1$, is stored in the user space QPN MT. Finally, $VFa1$ can be detached from VMa. After migration, as shown in Fig. 7b, $VFc1$ is attached to VMa. Then the new opaque pointer, $QPNc1$, is added to the user space QPN MT. Finally, the IB operations can be resumed, although each operation is intercepted by the user space MT to map $QPNc1$ to $QPNa1$.



(a) The suspend stage. (b) The resume stage.

Figure 8: The suspend and resume stage during three-stage migration process.

in Fig. 7. Initially, vm_a and vm_b are running *ib_rdma_bw*, one of the IB performance tests bundled with OpenFabrics Distribution (OFED), with vm_a as a client and vm_b as a server, as shown in Fig. 7a, and vm_c is left idle. While the communication between vm_a and vm_b is ongoing, we initiate the three–stage migration process.

First, the privileged domain of host A initiates the ***detachment of an active device***. This process triggers the suspend stage summarized in Fig. 8a. Here, the kernel IB module verifies that the PID exists in the QPN–PID MT, as shown in Fig. 7a, and signals a suspend event (step 1 in Fig. 8a) to quiesce the send operations in vm_a. A similar suspend event is forwarded, using the reliable service, to quiesce the send operation in vm_b, which is the peer of the migrating VM (step 2 in Fig. 8a). Quiescing the IB operations is an important step for the *reallocation of physical resources* because the QP can be maintained in a deterministic state where no outstanding send operations need to be resent after migration. As soon as the QP attributes and the out-of-band connection address are saved, the pinned memory is released. After the pinned memory is released (step 3 in Fig. 8a), the VF is successfully detached from vm_a and the user process remains in the suspend state until it receives a resume event signal. Then the privileged domain of host A migrates vm_a to host C.

After vm_a is migrated to host C, the attachment of a new VF to vm_a will trigger the resume stage. This also initiates the reallocation of physical resources. As shown in step 1 in Fig. 8b, a resume event is signalled to notify the user process of vm_a to create a new QP and its associated resources. Then the cached QP opaque pointer, $QPNa1$, is associated with the physical resources of the new QP ($QPc1$) via the MT shown in Fig. 7b (step 2 in Fig. 8b).

Finally, the ***reestablishment of a remote connection*** is executed by exchanging QP attributes between QPs (step 3 in Fig. 8b) and transferring both QPs into the RTS state. After that, the communication can be resumed. If RDMA operations are performed, an MT that remaps the RKey of the peer QP is needed. At this point the application can continue to use the cached QP attributes, but all operations are intercepted by the user library to retrieve the correct QP attributes from the user space QPN MT, as shown in Fig. 7b.

5.5. Late–detach Migration

Another improvement that we have made in the migration process, named as *late–detach migration*, is to reduce the service downtime during migration. Without using the Linux bonding driver, any SR-IOV device, including Ethernet, is expected to have service downtime on the order of seconds during VM migration [28]. This is related to the steps involved in today's virtualization model. First, the migration is initiated by the privileged domain to detach the VF. Next, the actual task of VM migration is executed to synchronize the dirty pages (memory) between the source (host A) and the destination (host C, the new location for vm_a). Finally, a new VF is reattached to vm_a at host C. However, the IB device is detached before the migration starts, which increases service downtime.

Ideally, if the service downtime of live migration is less than the timeout of reliable connection in IB, then the remote QP in vm_b does not have to be quiescent. The retry mechanism will retransmit any undelivered messages after migration, which can be used to avoid the remote QP transitions into the error state. This idea is similar to the transmission control protocol (TCP) timeout that is widely deployed in TCP/IP networks to perform VM migration. Hence, to minimize service downtime, the IB network outage should take place at the start of the stop-and-copy stage, the stage where the VM is suspended. Therefore, we propose performing the VM migration without detaching the VF from the migrating source, which allows the IB device to be operational until the start of the stop-and-copy stage. One assumption that we made in this improvement is that all the physical hosts in the subnet have a similar hardware specification. E.g All hosts have an IB HCA that supports SR-IOV.

6. Performance Evaluation

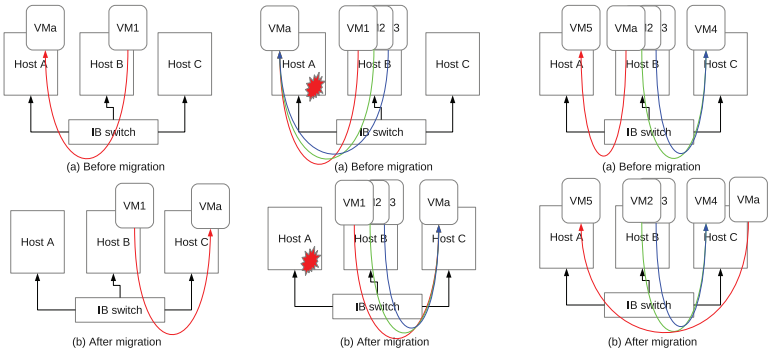
In this section, we present results from three different experiments. In the first experiment, we compare our proposed solution with the existing solutions, eIPoIB and SR-IOV IPoIB.In the second experiment, we demonstrated a high–availability scenario using live migration. In the third experiment, we show that live migration can be used to improve network throughput by load balancing VMs.

6.1. Experimental Setup

Our test bed consists of three hosts A, B, and C, connected by an IB switch. Each host is an Oracle Sun Fire X4170M2 server installed with Oracle VM Server (OVS) 3.0. Moreover, each host has a dual port Mellanox ConnectX2 QDR HCA that supports SR-IOV. Each VM is configured with two virtual CPUs, one IB VF, and 512MB of memory. The benchmark applications are IB performance test (perftest) and netperf [31]. Perftest consists of a collection of tests, that are bundled with the OpenFabrics Enterprise Distribution (OFED), written over IB uverbs intended to use as a performance micro-benchmark. In our experiment, we use *ib_rdma_bw*, one of the benchmarks in perftest, to stream data between VMs using RDMA write transactions. For the eIPoIB and IPoIB experiments we used netperf to stream data using TCP. The average page dirty rate during the pre-copy stage for these applications are approximately 500Mbps for each iteration.

6.2. Experiment I

In this experiment, as illusrated in Fig. 9a, we establish a point-to-point communication between 2 VMs (VMa on host A and $VM1$ on host B). Then, we migrate VMa to host C while the communication is still on–going. As we are comparing our three–stage migration process with the existing approaches, this experiment was carried out with: a system equipped with our three-stage migration process; a system with eIPoIB; and a system with SR-IOV IPoIB. Both SR-IOV IPoIB and eIPoIB depend on the recovery mechanism in the TCP/IP stack. We cannot use vRDMA as it is still work in progress, whereas RDS is an ULP that operates above IPoIB. The evaluation consists of the *service downtime*, the *network throughput*, and *CPU utilization* of these three systems.



(a) Experiment I testbed. The red line is the data flow.

(b) Experiment II testbed. The blue, green and red lines are the data flows.

(c) Experiment III testbed. The red line is the victim flow whereas the blue and green lines are the congested flows.

Figure 9: The experimental testbed.

6.2.1. Service Downtime

The service downtime is the network downtime during live migration, and is a key metric to evaluate the transparency of an IOV approach. Fig. 10 shows that the service downtime using eIPoIB, is approximately 3 s. The service downtime is higher than a conventional paravirtualized Ethernet due to the additional steps required for IB address resolution. Even though the eIPoIB shim layer translates the IPoIB header into the Ethernet header (and vice versa), the underlying L2 address (LID) is still required to be updated after migration.

Fig. 11 and Fig. 12 show the service downtime when using SR-IOV devices with late-detach migration The service downtime using IPoIB and our three–stage migration process is approximately 5 s and 3 s, respectively. SR-IOV IPoIB has a higher service donwtime because it relies on the TCP/IP stack to perform address resolution. During migration, the IPoIB connections are destroyed when the VF is detached. Then, new IPoIB connections are reestablished again after the address resolution. On the opposite, our approach has a lower service downtime compared to SR-IOV IPoIB, because the reestablishment of the connection happens at the IB verb layer. We present a detailed timing breakdown in Fig. 13 to explain the events happen during migration.

The first improvement, as shown in Fig. 13, is the implementation of late–detach migration. By delaying the detach of a VF has reduced the service downtime by 3 s or 60%, from the pre-migration stage until the iterative pre-copy stage. After multiple iterations of the pre-copy stage, the VF is detached at the start of the stop-and-copy stage before VMa is suspended. Even though the total migration time is approximately 4.7 s, the IB device downtime is only 1.9 s because the interval between the *pci-detach* and *pci-attach* events is reduced as illustrated in Fig. 13. Finally, a new VF is reattached at the destination host (host C) before VMa is resumed. Nevertheless, when a new VF is reattached to the VM, an additional 0.7 s is needed for reconfiguration. The reconfiguration process includes recreating the hardware–dependent resources and reestablishing the connection with the remote QP. As a result, the network connectivity loss experienced by the user application is approximately 3 s as shown in Fig. 12 and 13.

6.2.2. Network Throughput

Fig. 10, 11 and 12 show that the operating network throughput for a VM using eIPoIB, SR-IOV IPoIB and, SR-IOV IB is approximately 9800 Mbps,

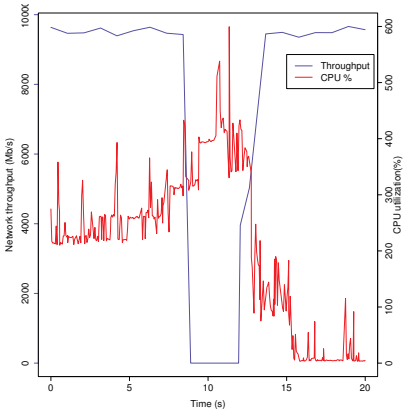


Figure 10: Network throughput and CPU utilization for Paravirtualized IPoIB (eIPoIB).

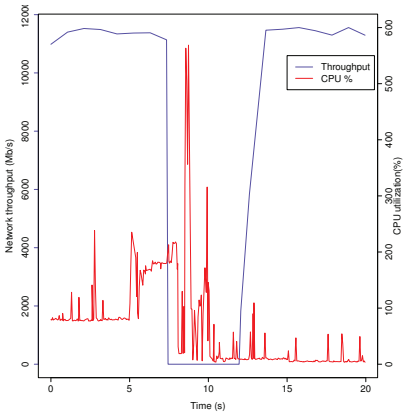


Figure 11: Network throughput and CPU utilization for SR-IOV IPoIB.

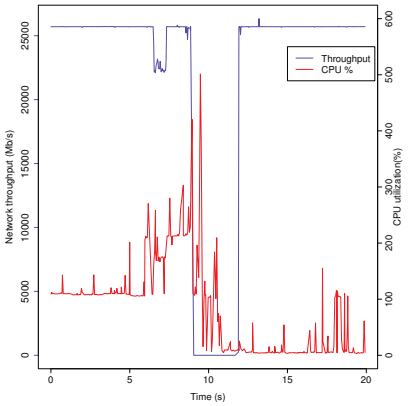


Figure 12: Network throughput and CPU utilization for SR-IOV IB (three–stage migration process).

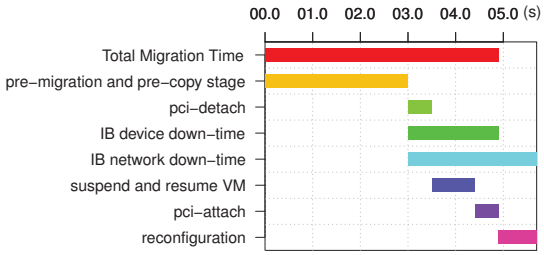


Figure 13: A detailed timing breakdown for each event during the migration.

11000 Mbps, and 25000 Mbps, respectively. If IB operates with IP or any other ULP, the performance suffers due to the overhead in the ULP. On the other hand, the RDMA operation using SR-IOV IB can achieve near to native performance. There is, however, a small dip in the network throughput after 5 s (Fig. 12) because partial of the network bandwidth is allocated by the migration tool to transfer the dirty pages during the pre-copy stage. Both eIPoIB and SR-IOV IPoIB do not observed the performance drop during pre-copy stage because the entire IB link bandwidth has not been utilized.

6.2.3. CPU utilization

The CPU utilization measurement was carried out at the source of the migrating host. Fig. 10 shows that the CPU utilization during data transmission for eIPoIB is approximately 300%. On the opposite, Fig 11 and 12 show that the CPU utilization during data transmission for both SR-IOV IPoIB and SR-IOV IB with RDMA operations are at approximately 150%. These results illustrate that eIPoIB, a paravirtualization IOV, consumes more CPU time than SR-IOV. One reason is that the split driver model (front-end and back-end driver) is consuming CPU time at the privileged domain during normal data transfer. The CPU utilization during live migration for three IOV approaches are the same, which can consume as high as 600%.

6.2.4. Summary

In this section, we summarize the results for experiment I. The eIPoIB paravirtualized IOV approach, has low service downtime (3 s) but high CPU utilization (300%) and low network throughput (9800 Mbps). The SR-IOV IPoIB approach has high service downtime (5 s), but low CPU utilization (150%) and low network throughput (11000 Mbps). Our solution, the three-stage migration process, has low service downtime (3 s), low CPU utilization (150%) and high network throughput (25000 Mbps). The network throughput is almost double of the network throughput using eIPoIB or SR-IOV IPoIB. In short, our three-stage migration has the best balancing point between efficiency, transparency and scalability among the three systems.

6.3. Experiment II

In this experiment, we emulated a high-availability scenario using our 3–host cluster, as illustrated in Fig. 9b. Three VMs ($VM1$, $VM2$, and $VM3$) were instantiated at host B, and one VM at host A (VMa). We create three RDMA connections using *ib_rdma_bw* between VMa and $VM1$, $VM2$, and

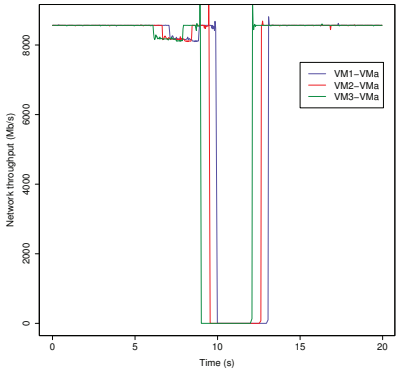


Figure 14: Per flow network throughput for experiment II in Fig. 9b.

$VM3$, as shown in Fig. 9b. We can only perform this experiment using our proposed solution, the three–stage migration process, because both eIPoib and SR-IOV IPoIB do not support RDMA write. Even though RDS support RDMA, it does not work with applications that are written using IB verbs API.

As shown in Fig. 14, each flow is getting a fair share of the link bandwidth at 8200 Mbps. While the data transmissions were still on-going, the management software detected that host A has a faulty component and host A can breakdown at any time. Thus, the management software instructed host A to migrate VMa to its high–availabity redudant host, host C. With our three-stage migration process, the established RDMA connections do not required to be halted. Even though there is a 3 s service downtime during migration, the RDMA connections resume without human intervention to re-establish these connections.

6.4. Experiment III

In this experiment, we demonstrated a load–balancing scenario in a congested network. As illustrated in Fig. 9c, we instantiated five VMs: three VMs (VMa, $VM2$,and $VM3$) at host B, one VM at host A ($VM5$) and last one at host C ($VM4$). There are three unidirection RDMA connections; VMa is communicating with $VM5$. $VM2$ and $VM3$ are communicating with

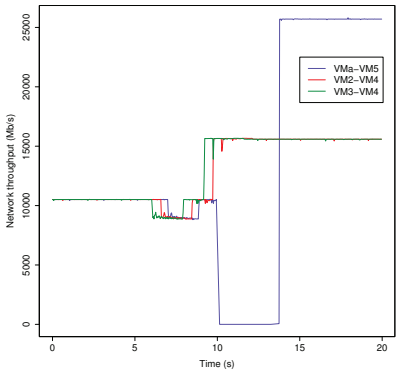


Figure 15: Per flow network throughput for experiment III in Fig. 9c.

$VM4$ simulataneously. This synthetic traffic pattern created a bottleneck at the link between the IB switch and host B.

As shown in Fig. 15, each flow is getting approximately 10000 Mbps. Even though there is only one connection between VMa and $VM5$, it obtains the same throughput as flow $VM2$–$VM4$ due to the head-of-line blocking. To load–balance the network, we migrate VMa to host C. After the service downtime, flow VMa–$VM5$ achieve the full link bandwidth at 25000 Mbps, whereas the two flows ($VM2$–$VM4$ and $VM3$–$VM5$) obtain higher throughput, as illustrated in Fig. 15. To identify a congested flow in the IB network, we can use dynamic rerouting approach that we proposed in [32] or the IB congestion control mechanism.

7. Future Work

As explained in Section 6, we have demonstrated an application transparent mechanism that can live migrate an IB VF. In near future, we plan to further reduce the service downtime,because it is still higher than the retry timeout of a reliable connection in IB. Consequently, both QPs (the migrated QP and the peer of the migrated QP) must be quiesced via handshaking, as mentioned in Section 6, impeding a more transparent migration. Another motivation for minimal service downtime is the benefit of a generic solution

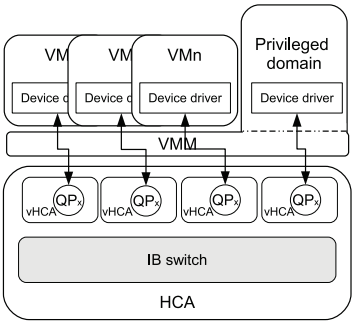


Figure 16: The virtual switch SR-IOV model for IB.

that supports live migration. In the long run, we believe that SR-IOV is an ideal approach for IOV. From the PCIe point of view, each VF is an isolated entity that has its own memory and a set of PCI configuration registers. However, from the IB architecture perspective, each VF is not an IB endpoint because the QP namespace is still shared among all the VFs. The sharing of the QP namespace among all VFs complicates the VM migration because the QP attributes cannot be reused. For instance, the identifier used to differentiate each QP, the QPN, must be reassigned after migration because the same QPN may already be assigned to a different QP. If an IB HCA can differentiate each VF as a virtual endpoint that has its own resource pool for the QP, then the QP attributes, such as the QPN, can be reused after migration.

We think that virtual switch model is a better SR-IOV model. As illustrated in Fig. 16, the HCA integrates a virtual IB switch to isolate each VF as a virtual endpoint, or vHCA. Each vHCA has its own QP namespace and is also discoverable by the network manager. The QP namespace isolation provides a better model, in terms of protection and flexibility, because each VF is restricted to access the QP resources within its partition, which extends the protection and isolation properties of VMs for the I/O operations. Moreover, no remapping table, as shown in Fig. 7, is needed because the QP attributes, such as the QPN and LID, can be reused after the migration.

Another feature that IB should have is a new suspend state to quiesce a

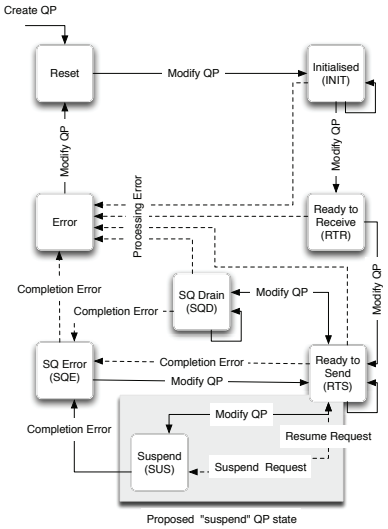


Figure 17: The proposed QP state diagram with the suspend state.

QP during migration, as shown in Fig. 17. The suspend state is driven by a software–initiated request or through an event request. During migration, the software interface transitions the migrating QP from the *RTS* state to the suspend state. When a QP enters the suspend state, it must complete any previously initiated message transmission and flush any remaining messages in the SQ without processing them. The hardware must also ensure that the remaining messages are cached by the user process buffer before flushing them. In the suspend state, any new message can be enqueued in both the SQ and the RQ, but remain posted without being processed until the QP is activated again.

8. Conclusion

Migrating a VM is different in conventional Ethernet network devices than in high-speed lossless networks such as IB. This is because they are based on distinct hardware architectures. The high–speed networking architecture

offers an efficient protocol stack. However, this requires additional complex offloading functionality in the hardware device, which makes VM migration less virtualization friendly compared to conventional network devices.

In this paper we have demonstrated the first design and implementation of live migration with SR-IOV IB. First, we described the challenges with respect to the live migration of SR-IOV enabled IB devices. Subsequently, we described a detailed implementation and measured the service downtime of an application during migration. The performance evaluataion shown that our prototype achieved near to native IB performance, low cpu utilization and low service downtime. Finally, we identified and debated the fraction of the cost of migration that can be mitigated by architectural changes in future hardware. With these changes, we argue that the service downtime of live migration with IB enabled SR-IOV devices (or similar high–performance lossless technologies) can be reduced to the millisecond range demonstrated with emulated Ethernet devices.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, M. Zaharia, Above the Clouds: A Berkeley View of Cloud Computing, Tech. rep. (2009).

[2] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, L. Smith, Intel Virtualization Technology, Computer 38 (5) (2005) 48–56.

[3] M. Rosenblum, T. Garfinkel, Virtual Machine Monitors: Current Technology and Future Trends, Computer 38 (5) (2005) 39–47.

[4] G. Vallee, T. Naughton, C. Engelmann, H. Ong, S. L. Scott, System-Level Virtualization for High Performance Computing, 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008) (2008) 636–643.

[5] W. Huang, J. Liu, B. Abali, D. K. Panda, A Case for High Performance Computing with Virtual Machines, in: Proceedings of the 20th annual international conference on Supercomputing, ACM, New York, NY, USA, 2006, p. 125.

[6] M. F. Mergen, V. Uhlig, O. Krieger, J. Xenidis, Virtualization for High-Performance Computing, ACM SIGOPS Operating Systems Review 40 (2) (2006) 8.

[7] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, J. Wiegert, Intel Virtualization Technology for Directed I / O, Intel Technology Journal 10 (03).

[8] PCI-SIG, Single Root I/O Virtualization and Sharing Specification Revision 1.1, 1st Edition (January 2010).

[9] J. Liu, W. Huang, B. Abali, D. K. Panda, High performance VMM-bypass I/O in virtual machines, in: USENIX Annual Technical Conference, USENIX Association, Berkeley, CA, USA, 2006.

[10] J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, W. Zwaenepoel, P. Willmann, Concurrent Direct Network Access for Virtual Machine Monitors, in: 2007 IEEE 13th International Symposium on High Performance Computer Architecture, IEEE, 2007, pp. 306–317.

[11] J. R. Santos, Y. Turner, G. Janakiraman, I. Pratt, Bridging the Gap between Software and Hardware Techniques for I/O Virtualization, in: USENIX 2008 Annual Technical Conference, USENIX Association, Berkeley, CA, USA, 2008, pp. 29–42.

[12] B. Li, Z. Huo, P. Zhang, D. Meng, Virtualizing Modern OS-bypass Networks with Performance and Scalability, in: IEEE Cluster 2010, 2010.

[13] Y. Dong, Z. Yu, G. Rose, SR-IOV Networking in Xen: Architecture, Design and Implementation, in: 1st Workshop on I/O Virtualization, 2008.

[14] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, H. Guan, High Performance Network Virtualization with SR-IOV, in: HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, Ieee, 2010, pp. 1–10.

[15] J. Liu, Evaluating Standard-based Self-Virtualizing Devices: A Performance Study on 10 GbE NICs with SR-IOV Support, in: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), Ieee, 2010, pp. 1–12.

[16] C. Clark, K. Fraser, S. H, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield, Live migration of virtual machines, in: In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2005, pp. 273–286.

[17] W. Huang, J. Liu, M. Koop, B. Abali, D. Panda, Nomad: Migrating OS-bypass Networks in Virtual Machines, in: Proceedings of the 3rd international conference on Virtual execution environments, VEE '07, ACM, New York, NY, USA, 2007, pp. 158–168.

[18] W. Huang, Q. Gao, J. Liu, D. K. Panda, High performance Virtual Machine Migration with RDMA over Modern Interconnects, 2007 IEEE International Conference on Cluster Computing (2007) 11–20doi:10.1109/CLUSTR.2007.4629212.

[19] Y. Dong, Y. Chen, Z. Pan, J. Dai, Y. Jiang, ReNIC: Architectural Extension to SR-IOV I/O Virtualization for Efficient Replication, ACM Trans. Archit. Code Optim. 8 (4) (2012) 40:1–40:22.

[20] R. P. Golberg, Architectural Principles for Virtual Computer Systems, Ph.D Thesis Div. Engineering and Applied Physics Harvard University.

[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the Art of Virtualization, in: 19th ACM Symposium on Operating Systems Principles, ACM Press, 2003, pp. 167–177.

[22] PCI-SIG, Address Translation Services Specification Rev 1.1, 1st Edition (January 2009).

[23] IBTA, Infiniband architecture specification, 1st Edition (November 2007).

[24] A. Ayoub, Ethernet Services over IPoIB, in: Open Fabrics Workshop 2012, March 2012.

[25] A. Ranadive, B. Havda, Toward a Paravirtual vRDMA Device for VMware ESXi Guests, in: VMware White Paper, December 2012.

[26] E. Zhai, G. D. Cummings, Y. Dong, Live migration with pass-through device for Linux VM, in: Ottawa Linux Symp.
(OLS), 2008, pp. 261–268.

[27] S. V. Howie Xu, Pankaj Thakkar, Networking IO Virtualization, in: VMworld 2008, September 2008.

[28] A. Kadav, M. M. Swift, Live migration of direct-access devices, SIGOPS Oper. Syst. Rev. 43 (2009) 95–104.

[29] D. S. Milojíčić, F. Douglis, Y. Paindaveine, R. Wheeler, S. Zhou, Process migration, ACM Comput. Surv. 32 (3) (2000) 241–299.

[30] X. Ouyang, S. Marcarelli, R. Rajachandrasekar, D. K. Panda, RDMA-Based Job Migration Framework for MPI over InfiniBand, in: Proceedings of the 2010 IEEE International Conference on Cluster Computing, CLUSTER '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 116–125.

[31] R. Jones, Netperf: A Network Performance Benchmark Revision 2.0, in: Hewlett Packard Information Networks Division White Paper, February 1995.

[32] W. L. Guay, S.-A. Reinemo, O. Lysne, T. Skeie, dFtree: A Fat-tree Routing Algorithm Using Dynamic Allocation of Virtual Lanes to Alleviate Congestion in Infiniband Networks, in: Proceedings of the first international workshop on Network-aware data management, NDM '11, ACM, New York, NY, USA, 2011, pp. 1–10.

Paper VI

A Scalable Signalling Mechanism for VM Migration with SR-IOV over InfiniBand

Wei Lin Guay, Sven-Arne Reinemo, Bjørn Dag Johnsen, Tor Skeie and Ola Torudbakken

2012 IEEE 18th International Conference on Parallel and Distributed Systems

A Scalable Signalling Mechanism for VM Migration with SR-IOV over InfiniBand

Wei Lin Guay*†, Sven-Arne Reinemo*, Bjørn Dag Johnsen†, Tor Skeie*‡, Ola Torudbakken†
†*Oracle Corporation*
{*wei.lin.guay, bjorn-dag.johnsen, ola.torudbakken*}*@oracle.com*
**Simula Research Laboratory, Norway*
{*svenar, tskeie*}*@simula.no*
‡*Department of Informatics, University of Oslo*

***Abstract*—Single Root I/O Virtualization (SR-IOV) is a promising I/O virtualization approach for achieving high performance in the virtualization over InfiniBand (IB) network. One challenge is related to the hardware address assignment for each virtual IB device. There are two schemes for the hardware address assignment; static assignment and dynamic assignment. Static assignment always preserves the hardware address of a virtual IB device that is attached to a VM, but the dynamic assignment does not. A drawback, however, using static assignment is that its communication will be disconnected after VM migration.**

In this paper, we point out the problem related to SR-IOV over IB that breaks the network connections after VM migration when the static assignment is deployed. Then, we propose a signalling mechanism that can maintain the network connectivity after VM migration. The performance evaluation using an experimental test bed shows that the proposed signalling mechanism does not increase the service downtime during hot migration. We also optimize the signalling method, where the same event can only be forwarded to a physical server once regardless of the hosted VMs, to reduce the management message overhead from $O(n * m)$ to $O(n)$.

I. Introduction

Today, one of the leveraging technologies that driving cloud computing is virtualization. It plays a critical role in cloud infrastructure, in resource allocation and management, efficient resource utilization, and elastic scalability.

Nevertheless, due to the intensive storage operations and IO communications, the virtualization of I/O resources has became a major performance bottleneck. Recent progress has improved the performance of I/O virtualization (IOV) by the introduction of *Direct assigned I/O* (DIO)[1] and *Single Root I/O Virtualization* (SR-IOV) [2]. Direct assigned I/O allows a VM to bypass the *virtual machine monitor* (VMM) and directly access the network hardware, which gives close to native performance [3], [4], [5], [6]. Unfortunately, it has one major drawback, and that is the lack of scalability. Only one VM can be attached to a device at a time, thus multiple VMs require multiple physical devices. This is expensive with regards to the number of devices and the internal bandwidth. To rectify this, SR-IOV was developed, which allows multiple VMs to share a single device because a single physical interface is presented as multiple virtual interfaces. Recent experience with SR-IOV has shown that such devices are able to provide high performance I/O in a virtualized environment [7], [8], [9].

A key feature provided by virtualization is VM migration [10], [11], [12]. VM migration is a powerful tool that extends the management of VMs from a single physical host to an entire cluster. Furthermore, it provides flexible provisioning of resources for improved load balancing, simpler maintenance, and increased efficiency. This leads to reduced downtime for the applications and services running on the VMs, and reduced power consumption during periods of low activity. VM migration has been demonstrated for both I/O virtualization with emulation over Ethernet [10] and I/O virtualization with DIO over InfiniBand [11], [12]. Compared to conventional Ethernet devices and migration based on emulation, high performance and lossless technologies like InfiniBand have more resources managed by hardware, making the necessary flexibility required from SR-IOV harder to realize.

One of the challenges in VM migration with SR-IOV over InfiniBand (IB) is the hardware address assignment for each virtual IB device, also known as the virtual function (VF). There are two schemes for the hardware address assignment; static assignment and dynamic assignment. In static assignment, the hardware address of a VF is always associated with a VM. On the opposite, in dynamic assignment the hardware address of a VF is always changed after VM migration. The drawback of static assignment is that the IB communication will be disconnected after VM migration. This is due to the architectural differences between SR-IOV over IB and the native IB. In the SR-IOV over IB the mapping between hardware address and the path information changes after VM migration, but not in the native IB. If the static assignment is used in the SR-IOV over IB, the hardware address mapping is not updated at the peer of the migrated VM that breaks the communication. So, in this paper we propose and implement a signalling mechanism to notify the migrated VM peer to update the hardware address mapping that allows the IB communication to be resumed after VM migration. Our experimental results also shows that the latency of the proposed signalling mechanism is negligible. Moreover, we restrict the signalling mechanism to only signal a same event to a physical server once regardless of the hosted VMs, in order to reduce the management message overhead.

This paper is organized as follows: Section II gives the necessary background on IOV and related technologies. Section III describes two hardware address assignment models. In Section IV, we discuss the signalling mechanism. In Section V, we present the experimental test bed and in Section VI, we evaluate the performance of the signalling mechanism using our test bed. Furthermore, we also use

1521-9097/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPADS.2012.60

mathematical analysis to analyze the management message overhead of the signalling mechanism. Lastly, we conclude and summarize in Section VII.

II. Background

In this section we give a brief description of IOV, the SR-IOV architecture, VM migration and IB architecture.

A. I/O Virtualization

IOV was introduced to provide availability of I/O by allowing VMs to access the underlying physical resources. Along with the wide adoption of virtualization came an increase in the number of VMs per server, which has made I/O a major bottleneck. This is due to two reasons: First, the available I/O resources at the physical level has not scaled with the number of cores. This is slowly being addressed with the introduction of faster version of PCI-express (PCIe) [13] and increased network speed. Second, a physical server handles multiple VMs must support the I/O requests from all these VMs, and due to the initial low I/O performance of even a single VM, this significantly reduces I/O performance. In practice, the combination of storage traffic and inter-server communication impose an increased load that may overwhelm the I/O resources of a single server, leading to backlogs and idle processors as they are waiting for data.

With the increase in number of I/O requests, the goal of IOV is not only to provide availability, but to improve performance, scalability and flexibility of the (virtualized) I/O resources to match the level of performance seen in the modern CPU virtualization. In the following subsection, we describe SR-IOV, the latest virtualization technique that can provide a scalable and high-performance IO virtualization.

1) Single Root IO Virtualization: To address the lack of scalability with direct assignment, the SR-IOV specification was created [2]. This specification extends the PCIe specification with the means to allow direct access to a single physical device from multiple VMs while maintaining near to native performance. Therefore, SR-IOV is expected to fully replace direct assignment as the IOV mechanism of choice in the near future. SR-IOV capable network devices are currently available for both Ethernet and InfiniBand devices.

SR-IOV allows a PCIe device to expose multiple virtual devices that can be shared among multiple VMs by allocating one virtual device to each VM. Each SR-IOV device has at least one *physical function* (PF) and one or more associated *virtual functions* (VF). A PF is a normal PCIe function controlled by the VMM, whereas a VF is a light-weight PCIe function. Each VF has its own memory address and is assigned a unique requester ID that enables IOMMU to differentiate between the traffic streams to different VFs. The IOMMU also provides memory and interrupt translations between the PF and the VFs.

The use of an SR-IOV capable device yields near to native performance and improved scalability as shown in [14]. The major drawback, however, is that SR-IOV is incompatible with hot migration and checkpoint/restart mechanisms as is also the case with direct assignment.

B. Virtual Machine Migration

VM migration, which is the ability to move a VM from one physical server to another under the management of VMM, is a virtualization feature that is increasingly utilized in today's enterprise environment. VM migration can provide system management needs such as hardware and software maintenance, and the data center scale-out requirements. There are two different schemes of VM migration : cold migration and hot migration.

1) Cold Migration: Cold migration is the traditional way to migrate a VM. The VM is shutdown and then restarted at the destination server. When the cold migration is initiated, the VM stops the running application and copies the page table (VM's memory) to the destination server. The total migration time is reduced because the page tables are not being modified during the cold migration. Nevertheless, the application uptime is not maintained because the VM is halted during the cold migration.

2) Hot Migration: Hot migration is to migrate a running VM from one physical server to another server while maintaining the running application uptime [10]. When the hot migration is initiated, the VM is not suspended instantly but the dirty pages (memory) are iteratively synchronized between the source and destination server. After the iterative copy phase, the VM is only suspended with a minimal downtime. The hot migration focuses on increasing the uptime of the VM, but the iterative copy phase that depends on the dirtied rate of page table has increased the total migration time.

C. The InfiniBand Architecture

The InfiniBand Architecture [15] was first standardized in October 2000, as a merge of the two technologies Future I/O and Next Generation I/O. As with most other recent interconnection networks, InfiniBand (IB) is a serial point-to-point full-duplex technology. InfiniBand networks are referred to as subnets, where a subnet consists of a set of hosts interconnected using switches and point-to-point links. An IB subnet requires at least one subnet manager (SM), which is responsible for initializing and bringing up the network, including the configuration of all the switches, routers and host channel adaptors (HCAs) in the subnet.

There are two SR-IOV models suggested for IB [16]: The *shared port* model which is used in the Mellanox ConnectX2 (CX2) hardware; And the *virtual switch* model. The latter model has several properties that simplifies IOV, but it requires more complex hardware. In the shared port model, all the VFs share a single port address and a single Queue Pair (QP) name space, and only the physical HCA port is discoverable by the network. In the virtual switch model each VF is a virtual HCA that contains a unique port address and a unique QP name space, and each virtual HCA is discoverable by the network.

III. The Hardware Address Assignment Models

Each physical IB device is assigned with two addresses: Local IDentity (LID) and Global Unique IDentity (GUID).

Both LID and GUID are the attributes in the path information. The LID is used to route IB packets within a subnet. On the other hand, The GUID is the hardware address that uniquely represents a physical IB device. The 64-bit GUID is combined with the local subnet prefix (64-bit) to form the Global Identifier (GID). The 128-bit GID is used to route IB packets between IB subnets.

In the current implementation of SR-IOV for IB, that is based on the shared port model, each VF shares the Local IDentity (LID) with the PF but has its own virtual GUID (vGUID). A virtual GUID is the hardware address that uniquely represents a VF. The *vGUID* is also referring as *hardware address* in this paper and both terms are used interchangeable.

In the following subsections, we explain two different hardware address assignment models that can be used to assign vGUID for each VF: dynamic assignment and static assignment.

A. Dynamic Assignment

This model is very similar to how addresses are assigned in the native IB. Along with assigning the LID and GUID for the PF, the SM is also responsible for assigning the vGUID for each VF. During the subnet initialization, the PF queries the SM for the vGUIDs allocated to it's VFs. Then, the SM generates the vGUIDs and responds to the requesting PF. After the PF receives the vGUIDs, they are stored in the GUID adminstration index table of dom0. Each VF is always associated with an appointed index as shown in Fig. 1. Nevertheless, the relation between an assigned VF and a VM is not constant. After VM migration, a new VF from the destination server is assigned to the VM. Consequently, the vGUID changes because it is obtained from the GUID adminstration index table in the destination server. E.g, at host A, vm_a is assigned with VF index 1 that has the vGUID value of $0xAAAA$ (as shown in Fig. 1a), but the vGUID changes to $0x11EE$ after vm_a is migrated to host B (as shown in Fig. 1b).

This model is simple and uses the SM to assign the addresses for both the VF and the PF, but the vGUID associated with a VM is not preserved after VM migration. Consequently, a Subnet Administration (SA) query needs to be performed after VM migration in order to obtain the path information for the new vGUID. These operations introduce additional latency in bringing up the IB VF. If hot migration is performed, this delay will increase the total service downtime.

B. Static Assignment

The static assignment is mainly targeted for the legacy application running in VM that has the requirement of preserving the hardware address of its networking interface after VM migration. In SR-IOV implementation for IB, the vGUID represents the hardware address of a virtual function (networking interface). Thus, the objective of static assignment is to preserve the vGUID that is assigned to a VM regardless of location. Compared to dynamic assignment, this implementation is more complex because it is

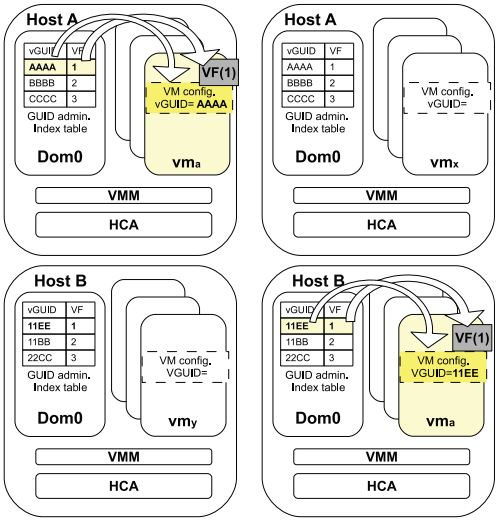
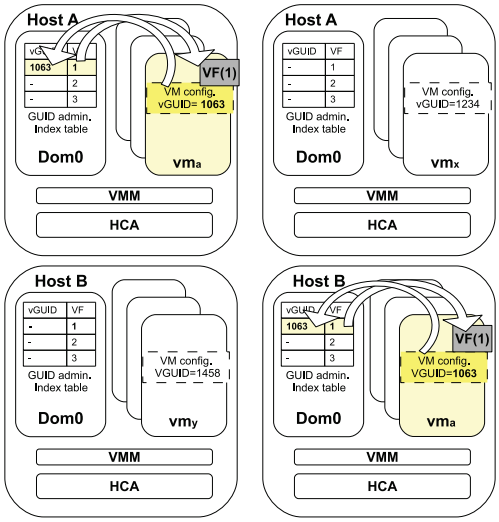


Figure 1. The dynamic assignment.

different from the native IB address assignment. In order to have a static vGUID throughout the VM life cycle, the implementation can be based on the SM or on a combination of the SM and the privileged domain (dom0). Due to the excessive management messages generated when only using the SM to maintain a static vGUID, the latter approach combining the SM and dom0 is a preferred solution. When a VM is instantiated, it is assigned with a vGUID that is kept as part of the VM configuration. The vGUID can be assigned by an external management software or the SM. When a VF is assigned to a VM, dom0 reads the vGUID from the VM configuration and writes the vGUID to the GUID administration index table. This event will also trigger a management message to update the SM with the latest vGUID to LID mapping. After VM migration, this model preserves the vGUID that is associated with the attached VF of a VM. E.g, at host A, vm_a has vGUID with the value of $0x1063$ (as shown in Fig. 2a). After vm_a is migrated to host B, the vGUID remains the same (as shown in Fig. 2b).

Although this model maintains the static vGUID for the assigned VF of a VM even if the VM is migrated to another physical host, this model is incompatible with VM migration. After VM migration, the peer of the migrated VM is not aware of the change in the LID to vGUID mapping. The peer continues to communicate with the migrated VM using its outdated cached path information. As a result, the peer fails to reach the migrated VM. This problem will be discussed in detail in section IV-A. In short, the static assignment must have a mechanism to notify the migrated



(a) (b)

Figure 2. The static assignment.

VM peers with the updated path information after VM migration.

IV. Signalling Method

In order to preserve the IB connection after VM migration when static assignment is deployed, we must implement a method to notify migrated VM peers to update their cached path information. In this section, first we identify and discuss the problem in the SR-IOV architecture that breaks the IB connection in subsection IV-A. Then, we propose an extension of our previous work [17], [18] to use as the signalling method to notify the migrated VM peers with an updated path information. The implementation of this signalling method is explained in Section IV-B.

A. What is the Problem?

In the native environment, a GUID to LID pair will not change unless the master SM changes. If master SM changes, the SM will also be accompanied with a client re-register event, that will refresh the cached path information. Nevertheless, the similar assumption, that a GUID to LID pair will not change, cannot be applied to the SR-IOV over IB environment.

In the dynamic assignment, the destination host will assign a new vGUID and LID to the migrated VM. Thus, both vGUID and LID will change after VM migration. In this case, a new path record query must be performed to update the cached path information in the peer of the migrated VM. Although this additional step is making the VM migration less transparent, the communication with the peer of the migrated VM can still be resumed.

In the static assignment, on the other hand, where only the LID will change after VM migration. This is because the vGUID remains the same but a new VF, from the destination server, that has a different LID is attached to the migrated VM. Although the SM and the migrated VM have the updated LID to vGUID pair, the peers of the migrated VM are not being notified. Due to the cached LID to vGUID pair in the migrated peers that was established earlier has became invalid, the migrated VM peers fail to reach the migrated VM or vice versa. In summary, the root of this problem is the lack of notification from the SM. In order to resolve this problem, a SM event, that has the latest vGUID to LID pair, must be triggered after VM migration to notify the migrated VM peers.

B. Implementation

Fig. 3 shows that the signalling method consists of two phases: The event registration phase and the event forwarding phase. In the event registration phase, each physical server must register its physical port GUID for the repath trap notification [15]. The physical server is acknowledged after the physical port GUID is registered successfully in the SM.

The event forwarding phase is performed when the VM migration happens. The SM must detect the changes in the LID-vGUID mapping and signals a repath trap, that has the latest LID-vGUID mapping, to all registered servers. In order to avoid triggering the repath trap during a new VM creation or a VM reboot event (the VM is shutdown and restarted at the same server), Alg. 1 is used in the SM to identify the changes in the LID to vGUID mapping. When

Algorithm 1 Trigger re-path trap with path record

```
1: if delete_guid(path_rec.guid) then
2:     add_to_repath_trap_table(path_rec)
3: else if set_guid(path_rec.guid) then
4:     if ret_rec=find_path_rec_from_repath_trap_table
       (path_rec.guid) then
5:         if ret_rec.dlid != path_rec.dlid then
6:             construct_repath_notice(ret_rec)
7:             signal_repath_trap(notice)
8:         end if
9:         remove_from_repath_trap_table(path_rec)
10:    end if
11: end if
```

a vGUID is deleted from the SM due to VM shutdown or VM migration, the path information will be added to the repath_trap table. The repath_trap table is a temporary storage that is used to differentiate between the VM creation event and the VM migration event.

If a vGUID is already existed in the repath_trap table when it is added to the SM, the associated path information is compared with the entry in the repath_trap table. If LID is

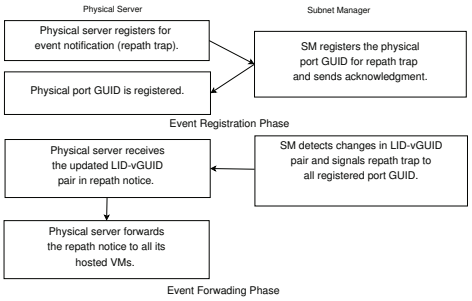


Figure 3. The Signalling mechanism.

different, this indicates that a VM migration has happened. This newly added vGUID and its associated path information is encapsulated in the repath trap. Then, the repath trap is signalled to all registered servers. After that, the vGUID entry in the repath_trap table is removed.

In order to avoid creating a bottleneck in the SM, we propose to use the dom0's GUID, or the physical port GUID, for event subscription instead of the vGUID. Then, dom0 is also responsible to broadcast the received notice to its hosted VMs. Alg. 2 shows the implementation in dom0 to forward the received repath trap notice to the active VMs. By only using the physical port GUID for event subscription, the generated management message (MMO) can be reduced. The evaluation of the MMO will be discussed in detail in section VI-C.

Algorithm 2 forward the repath notice to VMs

```
1: if is_repath_trap_notice(notice) then
2:   for i = 0 to max_supported_VFs do
3:     if guid_cache[i] ≠ NULL then
4:       ib_send_to_slave(notice)
5:     end if
6:   end for
7: end if
```

When the notice, that is forwarded by the dom0, is received by each VM, Alg. 3 will be executed to reconfigure the cached path information. The VM will extract its cached path information and compare them with the encapsulated path record in the repath trap. If there is a matched vGUID entry but with a different LID, the cached path information in the drivers will be updated.

V. Experimental Setup

Our test bed consists of three hosts: host A, B and C that are connected through IB using an IB switch. Furthermore, host A and B are also connected using the 10-Gigabit Ethernet (10GE). Each host is an Oracle Sun Fire X4170M2 server that is installed with Oracle VM Server (OVS) 3.0. Moreover, each host is also equipped with two PCI-express devices that have the SR-IOV capabilities, a dual port Mellanox ConnectX2 QDR host channel adapter (HCA) and a dual port Intel 82599EB 10 Gigabit-Ethernet network interface card (NIC). Each VM is instantiated with two vCPUs, 1GB memory and either an IB VF or a 10GE VF. In our experiment, we use *iperf*, a TCP/IP socket based application, to generate the TCP traffic. So, the IP over IB (IPoIB) protocol is used in order to run *iperf* with IB.

Algorithm 3 Reconfigure the cached path record

```
1: if received_repath_trap_notice(notice) then
2:   rec = get_SAagent_cache(notice.path_rec.gid)
3:   if rec.dlid ≠ notice.path_rec.dlid then
4:     update_SAagent_cache(notice.path_rec)
5:   end if
6: end if
```

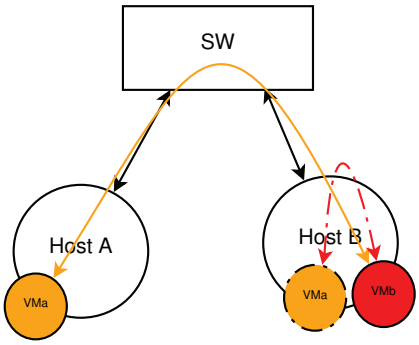


Figure 4. The scenario of experiment I. The solid line represents the flow before VM migration whereas the dotted line represents the flow after VM migration.

VI. Performance Evaluation

Our performance evaluation consists of measurements on an experimental test bed and an analysis of scalability. In the test bed measurement, we use *total service downtime* during hot migration as our main metrics to measure the additional latency that is introduced by the proposed signalling mechanism. For the mathematical analysis, we use *total management message overhead* as the metrics to evaluate the scalability of the signalling mechanism.

A. Experiment I

In this experiment, a TCP connection is established between vm_a and vm_b. vm_a is hosted by host A and vm_b is hosted by host B as illustrated in Fig. 4. Then, vm_a is migrated from host A to host B at 21s. This experiment is also carried out with two configurations. In the first configuration, we are using IPoIB whereas the second configuration with

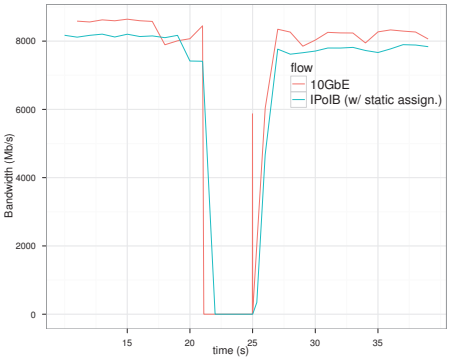


Figure 5. The service downtime of VM migration using IB VF and 10GbE VF.

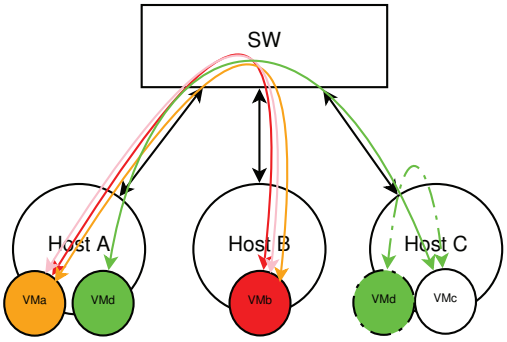


Figure 6. The scenario of experiment II. The solid lines represent the flows before VM migration whereas the dotted line represents the flow after VM migration.

10GbE. 10GbE is used in this experiment as the baseline result.

Due to the fact that the signalling mechanism will be triggered during VM migration, this experiment measures the additional latency that is introduced by the signalling mechanism during hot migration. We are not using IPoIB with the dynamic assignment addressing scheme as the baseline result even though it works with hot migration. This is because it must query the SA for the updated path information after VM migration. This additional step might increase the total service downtime. As a result, we are using the 10GE VF as the baseline result to identify the latency of the signalling mechanism. Another point worth mentioning here is that the service downtime with a SR-IOV VF (including 10GE) is expected to be in the magnitude of seconds, if hot migration is performed without the bonding driver [19]. This is because VM migration cannot be performed if a SR-IOV VF is still attached to the VM. Consequently, the SR-IOV VF must be detached from the VM and re-attached again after the VM migration is completed. As a result, the service downtime has increased from miliseconds range for the live migration operation to the magnitude of seconds due to the detachment and attachment of the SR-IOV VF.

Fig. 5 shows the service downtime during hot migration using a 10GE VF and an IB VF. Without the bonding driver, the service downtime is approximately 4s regardless of which VF is used. Even though with IB VF the signalling mechanism is triggered to maintain the IPoIB connectivity, it does not increase the total service downtime. Thus, we can conclude that the proposed signalling mechanism has a negligible latency that does not increase the service downtime. On the opposite, without the signalling mechanism to notify the peer of the migrated VM, the IPoIB communication will be disconnected after VM migration.

Another observation from Fig. 5 that we like to point out is the network throughput. The network throughput of IPoIB and 10GbE are approximately 8Gbps and 8.3Gbps, respectively. Although Fig. 5 shows that the throughput of 10GbE is slightly higher than IPoIB, the collected result shown in Fig. 5 is only based on a single run. The network throughput for both IPoIB and 10GE (with single connection) are usually fluctuated within the range of 7.5Gbps to 8.5Gbps, which matches what we have obtained in Fig. 5.

Another reason is the overhead in the IPoIB software stack, where the obtained network throughput is much lower than the IB native throughput of QDR at 32Gbps. With a single IPoIB connection runs in a VM, the throughput is approximately 8Gbps. If multiple IPoIB connections run in several VMs simultaneously, the aggregated throughput with IPoIB can reach 19Gbps. This is similar to the IPoIB performance on a physical (non-virtualized) configuration which is limited by PCIe Gen2. In the combination of new Intel x86 Romley platform and a new Mellanox CX3 HCA that both have PCIe gen3, a higher throughput can be demonstrated. The performance of IPoIB is CPU bound due to the TCP termination at the TCP server.

The IPoIB performance of a VM is limited to approximately 8Gbps because Xen does not support direct interrupt injection into a guest even though the x86 architecture, VT-d [1], does [20], [21].

B. Experiment II

In this experiment, vm_a and vm_d are hosted by host A, vm_b is hosted by host B, and vm_c is hosted by host C. The communication consists of three individual flows between vm_a and vm_b and a flow between vm_d and vm_c as illustrated in Fig. 6. All the communications are established using IPoIB. At 17s, vm_d is migrated from host A to host C. Even though the signalling event is broadcasted to all hosts during VM migration, this experiment shows that normal data flows will not be affected by the proposed signalling mechanism.

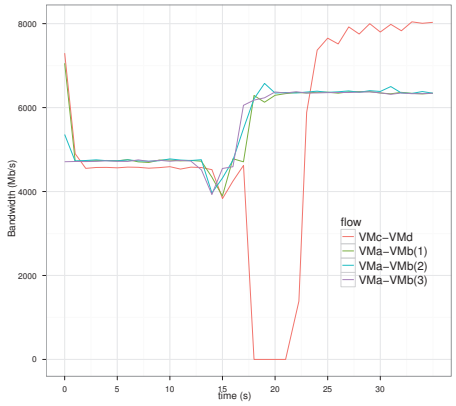


Figure 7. The per flow throughput in experiment II.

Fig. 7 shows the per flow throughput before and after VM migration. Before migration, each flow obtains approximately 4.75Gbps. When vm_d is migrated to host C at 17s, there is a 4s of service outage in the communication between vm_d and vm_c. On the other hand, the remaining three flows between vm_a and vm_b are instantly getting a higher share of the physical link bandwidth at 6.2Gbps. After migration, the communication between vm_c and vm_d is localized at host C. Consequently, the throughput of this flow is increased to approximately 8Gbps as shown in Fig. 7.

In this experiment, although vm_a and vm_b received the signalling event when vm_d was migrated to host C at 17s, these communication flows were not impacted by the signalling event. On the opposite, these flows achieved a higher throughput at 17s because the communication between vm_c and vm_d was no longer using the physical link between host A and the IB switch. This link was only dedicated to three individual flows between vm_a and vm_b. Another important observation from this experiment is to emphasis that the aggregated throughput (per server) with IPoIB can reach approximately 19Gbps.

C. Scalability

Even though the proposed signalling mechanism is an architecturally correct approach, without a proper handling or excessive use of the event forwarding might cause a bottleneck in the SM. This is due to the nature of handshaking in the IB event forwarding. In this section, we use the mathematical analysis to analyze the *total management message overhead* of the signalling mechanism.

Before a VM can receive the notice, it must register itself for the event notification from the SM/SA. In a virtualized environment, there are n physical servers and each server is hosting m VMs. If the event subscription is registered according to a VM's vGUID, the management message

Table I
TABLE OF NOTATION

Symbol	Meaning
n	Number of physical server.
m	Number of hosted virtual machine.
msg_{repath}	Message size of a re-path trap.
msg_{gmp}	Total message size of a general service management packet.
$msg_{register}$	Total message size of an event subscription.
msg_{ACK}	Total message size of a SA acknowledgement.
$ohd_{startup}$	Total message overhead of the startup phase of event registration.
$ohd_{startup\_opt}$	Total message overhead of the startup phase of event registration with optimization.
$ohd_{event\_forwarding}$	Total message overhead during event forwarding.
$ohd_{event\_forwarding\_opt}$	Total message overhead during event forwarding with optimization.
ohd_{LB}	Total message overhead in the load balancing scenario.
ohd_{HA}	Total message overhead in the high availability scenario.

overhead (MMO) that will be generated during the startup phase is derived by Eq. 1.

$$\begin{aligned} ohd_{startup} &= n * m * msg_{register} + n * m * msg_{ACK} \\ &= 2 * n * m * msg_{gmp} \end{aligned} \tag{1}$$

Each VM, that consists of m number of them that are hosted by n physical servers, will send a $msg_{register}$ to register for an event notification. In return, the SM/SA will respond with a msg_{ACK}. The MMO is bounded to both the number of VMs, m, and the number of physical servers, n. On the opposite, our signalling mechanism is using dom0 for event subscription where only the physical port GUID is registered in the SM. In this case, the MMO during the startup phase is only depending on the physical servers, n, as derived in Eq. 2.

$$\begin{aligned} ohd_{startup\_opt} &= n * msg_{register} + n * msg_{ACK} \\ &= 2n * msg_{gmp} \end{aligned} \tag{2}$$

Now we look at the MMO during the event forwarding. If the event forwarding is based on a VM's vGUID, the MMO can be derived by Eq. 3. Due to the fact that all active VMs must be notified, the physical server that hosted multiple VMs will receive multiple redundant notices.

$$\begin{aligned} ohd_{event\_forwarding} &= n * m * msg_{repath} \\ &= n * m * msg_{gmp} \end{aligned} \quad (3)$$

In our proposed signalling mechanism, only the dom0 receives the notice. It is responsible to broadcast the received notice to its hosted VMs. The MMO of our signalling mechanism can be derived in Eq. 4. Regardless of hosted VMs, m, a similar event will only be forwarded to a physical server once.

$$\begin{aligned} ohd_{event\_forwarding\_opt} &= n * msg_{repath} \\ &= n * msg_{gmp} \end{aligned} \quad (4)$$

In summary, the abovementioned mathematical equations show that our signalling mechanism, that is based on dom0, has improved the scalability by reducing the management message overhead from $O(n * m)$ to $O(n)$.

VII. Conclusion and Future Work

SR-IOV is a promising I/O virtualization approach for achieving high performance in the virtualization over IB. One challenge is related to the hardware address assignment for each VF. If the hardware address that is associated with a VM must be preserved throughout a VM life cycle, the communication fails to resume after VM migration.

In this paper, we have identified the problem related to SR-IOV that breaks the network connection after VM migration when the association between the hardware address and a VM must be preserved. Then, we proposed and implemented a signalling mechanism that maintains the network connectivity after VM migration. We have evaluated the performance on a small test bed and it is evident from our results that the latency of the signalling mechanism is negligible in a small scale cluster. Moreover, we also optimize the signalling method, where a same event can only be forwarded to a physical server once regardless of the hosted VMs, to reduce the management message overhead.

Future work includes studying this method in large clusters in order to better evaluate the performance and scalability. We also plan to reduce the total service downtime during VM migration by applying the concept of the Linux bonding driver in IB.

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel Virtualization Technology for Directed I / O," *Intel Technology Journal*, vol. 10, no. 03, 2006.

[2] *Single root I/O Virtualization and sharing specification*, 1st ed., PCI-SIG, January 2010.

[3] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance VMM-bypass I/O in virtual machines," in *USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1267359.1267362

[4] J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, W. Zwaenepoel, and P. Willmann, "Concurrent Direct Network Access for Virtual Machine Monitors," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, Feb. 2007, pp. 306–317. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4147671

[5] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for I/O virtualization," in *USENIX 2008 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: http://portal.acm.org/citation.cfm?id=1404014.1404017

[6] B. Li, Z. Huo, P. Zhang, and D. Meng, "Virtualizing Modern OS-bypass Networks with Performance and Scalability," in *IEEE Cluster 2010*, 2010.

[7] Y. Dong, Z. Yu, and G. Rose, "SR-IOV Networking in Xen: Architecture, Design and Implementation," in *1st Workshop on I/O Virtualization*, 2008.

[8] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. Ieee, Jan. 2010, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5416637

[9] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. Ieee, Apr. 2010, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5470365

[10] C. Clark, K. Fraser, S. H, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI*, 2005, pp. 273–286.

[11] W. Huang, J. Liu, M. Koop, B. Abali, and D. Panda, "Nomad: migrating os-bypass networks in virtual machines," in *Proceedings of the 3rd international conference on Virtual execution environments*, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 158–168. [Online]. Available: http://doi.acm.org/10.1145/1254810.1254833

[12] W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High performance virtual machine migration with RDMA over modern interconnects," *2007 IEEE International Conference on Cluster Computing*, pp. 11–20, 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4629212

[13] PCISIG, *PCI Express Base Specification*, 3rd ed., November 2010.

[14] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support," Apr. 2010, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2010.5470365

[15] IBTA, *Infiniband architecture specification*, 1st ed., November 2007.

[16] L. Liss, "Infiniband and rocee virtualization with sr-iov," in *Open Fabrics Workshop 2010*, March 2010.

[17] 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 (CCGrid 2011)*, R. Buyya, Ed. IEEE Computer Society Press, 2011, pp. 332 – 341.

[18] 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*, 2010, pp. 126–135.

[19] "Linux ethernet bonding driver," http://www.kernel.org/doc/Documentation/networking/bonding.txt.

[20] A. Kay and E. Dong, "Vt-d and sr-iov update," in *Xen Summit Oracle 2009*, February 2009.

[21] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, Jan. 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2010.5416637