Efficient Network Isolation and Load Balancing in Multi-Tenant HPC Clusters

Feroz Zahid[a,b,*], Ernst Gunnar Gran[a], Bartosz Bogdański[c], Bjørn Dag Johnsen[c], Tor Skeie[a,b]

[a] *Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway*
[b] *University of Oslo, Norway*
[c] *Oracle Corporation*

Abstract

Multi-tenancy promises high utilization of available system resources and helps maintaining cost-effective operations for service providers. However, multi-tenant high-performance computing (HPC) infrastructures, like dynamic HPC clouds, bring unique challenges, both associated with providing performance isolation to the tenants, and achieving efficient load-balancing across the network fabric. Each tenant should experience predictable network performance, unaffected by the workload of other tenants. At the same time, it is equally important that the network links are balanced, avoiding network saturation. The network saturation can lead to unpredictable application performance, and a potential loss of profit for the cloud service providers.

In this paper, we present two significant extensions to our previously proposed partition-aware fat-tree routing algorithm, pFTree, for InfiniBand-based HPC systems. First, we extend pFTree to incorporate provider defined partition-wise policies that govern how the nodes in different partitions are allowed to share network resources with each other. Second, we present a weighted version of the pFTree routing algorithm, that besides partitions, also takes node traffic characteristics into account to balance load across the network links more evenly. A comprehensive evaluation comprising both real-world experiments and simulations confirms the correctness and feasibility of the proposed extensions.

Keywords: `Interconnection Networks`, `Routing Algorithms`, `Performance Isolation`, `Load Balancing`, `InfiniBand`, `Virtual Channels`

1. Introduction

Over the last decade, we have seen a continuous growth in the popularity of InfiniBand (IB) [1] as a network interconnect for high-performance computing (HPC) systems and data centers. The recent *Top 500* [2] supercomputer list, released in November 2015, reports that about 47.4% of the most powerful supercomputers in the world use IB as their interconnect. The popularity of IB is largely attributed to the high-throughput and low-latency communication it offers. Furthermore, IB provides sufficient security mechanisms to complement in typical non-trusted data center environments.

Recently, the use of IB in cloud computing environments has also gained interest in the HPC community [3, 4, 5, 6]. Multi-tenancy is a salient feature of cloud computing, and is defined as a scheme where applications belonging to different users are co-located in a shared data center infrastructure [7]. Multi-tenancy promises high utilization of system resources and helps maintaining cost-effective operation for service providers. However, multi-tenant infrastructures also introduce several security and performance challenges [8, 9], the most critical one being associated with providing performance isolation to the tenants [10, 11]. Previous research has shown that the sharing of resources with other tenants in a shared cloud

*Corresponding author
Email address: `feroz@simula.no` (Feroz Zahid)

incurs unpredictable application performance [12, 13, 14]. Allocating a single tenant per physical machine, as employed by major HPC cloud providers like *Amazon* [15], eliminates performance interference imposed by server sharing between tenants, but the shared network infrastructure still remains an issue. From the networking perspective, ideally each tenant should experience predictable network performance, unaffected by the workload of other tenants in the system. At the same time, it is equally important for the service providers to avoid network saturation using efficient load-balancing techniques [16]. The network saturation can result in unpredictable application performance. As a result, cloud service providers can potentially face a loss of profit, for instance, they may not be able to support better Service Level Agreements (SLAs) for the clients.

Network isolation in IB systems is provided through partitioning. *Partitions* are logical groups of ports such that the members of a group can only communicate with other members of the same group. At host channel adapters (HCAs) and switches, packets are filtered using the partition membership information to enforce isolation. In multi-tenant IB systems, partitions can be used to create tenant clusters [4]. With partition enforcement in place, a node cannot communicate with other nodes that belong to a different tenant cluster, as packets with invalid partitioning information are dropped as soon as they reach an incoming port. The routing algorithm, however, is unaware of these partitions in the network. Hence, traffic flows belonging to different partitions might share links inside the network fabric. This sharing of intermediate links creates interference between partitions, resulting in non-predictable network performance. In addition, using current routing schemes, the load-balancing methods of the routing algorithm are also affected. This degradation is due to the fact that routes crossing partition boundaries are considered when distributing routes onto links in the network, despite the fact that these routes are never used. Degraded balancing may result in reduced effective bandwidth and sub-optimal network utilization [17].

To cater for the aforementioned routing challenges on multi-tenant IB clusters, in [18] we presented pFTree, a partition-aware routing algorithm for fat-tree topologies. The pFTree algorithm utilizes several mechanisms to provide network-wide isolation of partitions belonging to different tenants. Given the available network resources, pFTree starts by isolating partitions at the physical link level, and then moves on to utilize virtual lanes, if needed.

In this paper, we present two significant extensions to the pFTree routing algorithm. First, we extend pFTree to incorporate provider defined partition-wise policies that govern how the partitions share resources in the network. These policies are useful when some of the partitions in the network run critical operations, or have very high Quality of Service (QoS) or security requirements. In particular, we show that an extended pFTree routing algorithm is able to completely remove inter-partition interference for a partition marked physically isolated by the provider policy. Second, we present a weighted version of the pFTree routing, that considers node traffic characteristics to balance load across the network links more evenly. This extension targets subnets where nodes exhibit distinct traffic characteristics.

The main contributions of this paper are:

- We present an extended pFTree routing algorithm (pFTree-Ext) that supports different provider-defined partition-wise isolation policies.
- For better load-balancing with nodes having distinct traffic characteristics, we propose a weighted version of the pFTree routing algorithm (pFTree-Wt).
- We present implementations of our proposed extensions to the pFTree algorithm in OFED's subnet manager (OpenSM)[1].
- We evaluate pFTree-Ext and pFTree-Wt by comparing them with the current pFTree routing algorithm using both real-world experiments and large-scale simulations.
- We present a critical analysis of both extensions and discuss important trade-offs between higher network utilization and performance isolation on an HPC cluster.

[1]The OpenFabrics Enterprise Distribution (OFED) is the de facto standard software stack for deploying IB based applications. http://openfabrics.org/

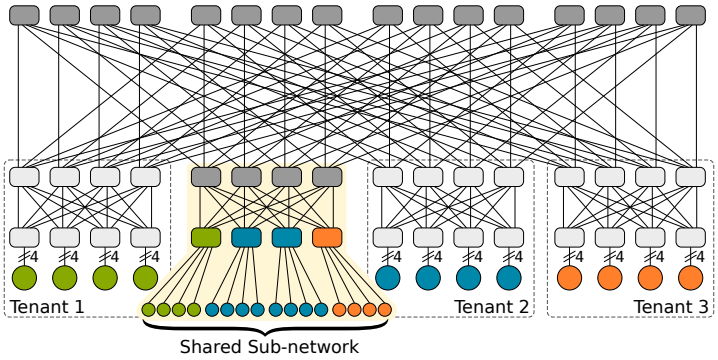


Figure 1: Tenant allocation in a fat-tree network with full resource utilization

The rest of this paper is structured as follows. In Section 2 we further motivate the importance of our work in the context of real-world multi-tenant HPC systems. The technical background about the IB architecture and fat-tree topologies is given in Section 3. Section 4 discusses challenges associated with routing in multi-tenant HPC clusters. We give an overview of the pFTree routing algorithm in Section 5. The extended pFTree routing algorithm with partition-wise isolation policies and the weighted pFTree routing algorithm are presented and evaluated in Section 6 and Section 7, respectively. In Section 8, we analyze the proposed extensions and present insight on the concerned trade-offs and potential future directions. Finally, we present the related work found in the literature in Section 9, before we conclude in Section 10.

2. Motivation

The cloud architectures offer significant advantages over traditional cluster computing architectures including flexibility, ease of deployment, high-availability, and on-demand resource allocation - all packed up in an economically attractive pay-as-you-go [19] business model for its users. Many HPC users would also like to benefit from feature-rich cloud offerings, potentially saving them substantial upfront costs while providing *instant* and *pseudo-unlimited* resource capacity for their applications. However, the effective use of cloud computing for the HPC systems still remains questionable [20, 21]. Applications running on shared clouds are vulnerable to performance unpredictability and violations of service level guarantees usually required for the HPC applications [12, 13]. The performance unpredictability in a multi-tenant cloud computing system typically arises from server virtualization and network sharing. While the former can easily be addressed by allocating only a single tenant per physical machine, the sharing of network resources still remains a major performance variability issue. Intuitively, the network performance received by the applications of a tenant in a shared cloud is affected by the workload of other tenants in the system. This holds true for multi-tenant enterprise HPC systems as well, where jobs belonging to different customers are run in a shared network infrastructure.

In hierarchical network topologies, like fat-trees [22], the tenants can be assigned to different leaf-switches or sub-networks providing network isolation inherited from the structure of the topology. However, such an allocation scheme only works for a restricted number of tenants, and for a very rigid server requirement from each tenant workload. For generalized job placement, isolation provided through the topology structure might either not be possible or result in significant resource underutilization. Consider the three-level fat-tree network shown in Figure 1, the topology can be thought of as composed of four two-level sub-networks each consisting of 8 switches and 16 end nodes. Four tenants, each requiring full 16 end nodes for their workload execution, can be accommodated with complete isolation in the network without requiring support from the routing algorithm. However, if we need to accommodate a different number of tenant groups or a different end node requirement from the tenants, some of the end nodes must share a sub-network with other tenant nodes. As shown in the figure, three tenants requiring 20, 24, and 20 end nodes, need to

share the second sub-network to comply with their server needs. The switches that may need a change in routing for providing isolation are shown in dark gray color in the figure. Relying only on the topology given isolation, only two of the three tenants can be supported leaving at least 20 server machines unused. The problem further escalates on oversubscribed topologies, which are commonly used in modern cloud systems [23]. With the support of oversubscription, service providers want to support as many tenants as possible, resulting in increased network link sharing among tenants. In general, irrespective of the topology, performance isolation for tenants can only be provided by a tenant-aware routing algorithm like the novel algorithm presented in this paper.

In dynamic HPC clouds, tenant server machines are allocated, freed and reallocated often. The frequent (re-)allocations result in non-contiguous blocks of server machines belonging to different tenants at the switches in the network. The problem is known as data center fragmentation [24], and requires costly mitigation processes involving resource migration, for instance. For such dynamic environments, it is even more iMoreover, different tenants might also have different isolation requirements depending on their SLAs. The different SLA requirements should also be taken into consideration when assigning network links to the tenants.

3. Technical Background

In the following, we provide technical background for the IB interconnect technology, including specifics about routing and network reconfiguration in IB networks. We also give an overview of the fat-tree topologies, and discuss OpenSM's fat-tree routing algorithm.

3.1. The InfiniBand Architecture

IB [1] is an open standard lossless network technology developed by the InfiniBand Trade Association[2]. The technology defines a serial point-to-point full-duplex interconnect that offers high-throughput and low-latency communication, geared particularly towards HPC applications and data centers.

An IB network consists of one or more *subnets* interconnected using routers. Within a subnet, hosts are connected using switches and point-to-point links. There is one active management entity, the *master subnet manager* (Master SM) - residing on any designated subnet device - that configures, activates, and maintains the IB subnet. Through the subnet management interface, the SM exchanges control packets, called subnet management packets (SMPs), with the subnet management agents (SMAs) that reside on every IB device. Using SMPs, the SM is able to discover the fabric, configure end nodes and switches, and receive notifications from SMAs. Except for the master SM, all other SMs in the subnet are in *standby* mode for fault-tolerance. In case a master SM fails, a new master is negotiated by the standby SMs using a *master election and handover protocol*. The SM also performs periodic light sweeps of the subnet to detect any topology changes, node addition/deletion or link failures, and reconfigures the network accordingly. More details about the subnet discovery mechanism are given in [25].

3.1.1. Routing

Intra-subnet routing in an IB network is based on the Linear Forwarding Tables (LFTs) stored in the switches. The LFTs are calculated by the SM according to the routing algorithm in use. The current OpenSM implementation offers several routing algorithms including MinHop, the fat-tree routing algorithm [26], LASH [27], and DFSSSP [28]. In a subnet, all HCA ports on the end nodes and all switches are addressed using local identifiers (LIDs). Each entry in an LFT consists of a destination LID (DLID) and an output port. Only one entry per LID in the table is supported. When a packet arrives at a switch, its output port is determined by looking up the DLID in the forwarding table of the switch. The routing is *deterministic* as packets always take the same path in the network between a given source-destination pair.

[2] http://infinibandta.org/

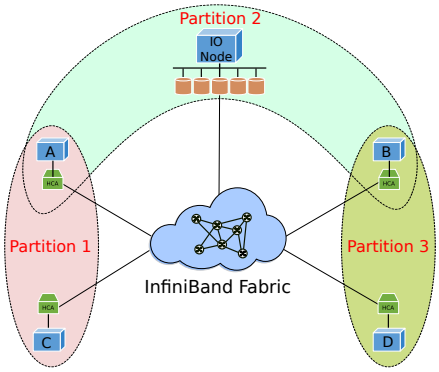


Figure 2: An example of the partitioning in IB networks.

3.1.2. Partitioning

Partitioning is a security mechanism provided by IB to enforce isolation of logical groups of systems sharing a network fabric. The IB partitions provide similar isolation features as Ethernet 802.1Q VLANs [29]. Each HCA port on a node in the fabric can be a member of one or more partitions. Partition memberships are managed by a centralized partition manager, which is a part of the SM. Two types of memberships are supported: *full* and *limited*. Limited members cannot communicate with other limited members. However, limited members can communicate with the full members of the partition. Full members can communicate with all the members of a partition regardless of their membership type. There is a *default* partition that is created by SM regardless of the presence of other partitions to allow management traffic in the subnet. The SM configures partition membership information on each port as a table of 16-bit partition keys (*P_Keys*). The SM also configures switches and routers with the partition enforcement tables containing *P_Key* information associated with the LIDs.

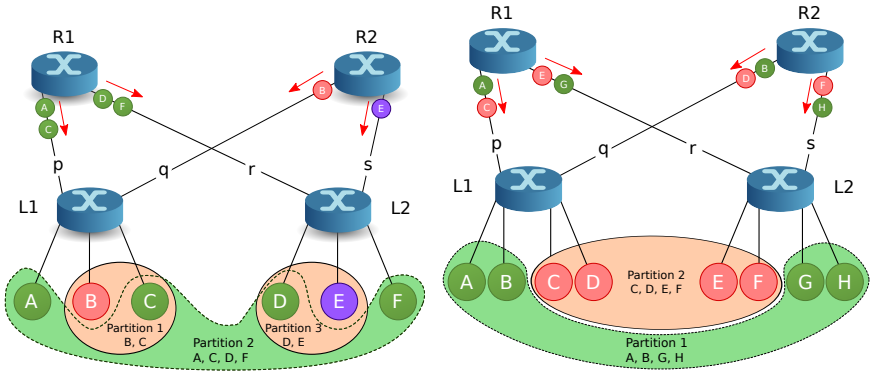
For the communication between nodes, Queue Pairs (QPs) and End-to-End contexts (EECs) are assigned to a particular partition, except for the management Queue Pairs (QP0 and QP1). The *P_Key* information is then added to every IB transport packet sent. When a packet arrives at an HCA port or a switch, its *P_Key* value is validated against the table configured by the SM. If an invalid *P_Key* value is found, the packet is discarded immediately. In this way, communication is allowed only between ports sharing a partition. An example of IB partitions is shown in Figure 2. Node C and node D are not allowed to communicate as they do not share a partition.

3.1.3. Quality of Service

IB has a layered architecture where each physical link can be divided into multiple virtual links using Virtual Lanes (VLs). Each VL has its own buffering, flow-control and congestion management resources. QoS is provided through a set of differentiated traffic classes, called Service Levels (SLs). The SL represents the class of service a packet receives in the network. On each link, the mapping between SLs and VLs is done using a configurable *SL-to-VL* mapping table. IB supports up to 16 VLs. However, the last VL is reserved for the subnet management traffic and cannot be used by user applications.

3.2. Fat-Tree Topologies and Routing

Many of the IB based HPC systems employ a fat-tree topology [22] to take advantage of the useful properties fat-trees offer. These properties include full bisection-bandwidth and inherent fault-tolerance due to the availability of multiple paths. The initial idea behind fat-trees was to employ *fatter* links, that is links with more available bandwidth between nodes, as we move towards the roots of the topology. The fatter links help to avoid congestion in the upper-level switches and the bisection-bandwidth is maintained. Different variations of fat-trees are later presented in the literature, including k-ary-n-trees [30], Extended Generalized Fat-Trees (XGFTs) [31], Parallel Ports Generalized Fat-Trees (PGFTs) and Real Life Fat-Trees (RLFTs) [32].



(a) Degraded load-balancing (b) No isolation between partitions

Figure 3: The issues with fat-tree routing in partitioned subnets.

A k-ary-n-tree [30] is an n level fat-tree with k^n end nodes and $n \times k^{n-1}$ switches, each with $2k$ ports. Each switch has an equal number of up and down connections in the tree, except for the root switches. The XGFT fat-tree extends k-ary-n-trees by allowing both different number of up and down connections for the switches, and different number of connections at each level in the tree. The PGFT definition further broadens XGFT topologies and permits multiple connections between switches. A large variety of topologies can be defined using XGFTs and PGFTs. However, for practical purposes, RLFT, which is a restricted version of PGFT, is introduced to define fat-trees commonly found in today's HPC clusters [33]. A RLFT uses the same port-count switches at all levels in the fat-tree.

3.2.1. Fat-Tree Routing Algorithm

The fat-tree routing algorithm [26, 32] is one of the most popular routing algorithms for IB based fat-tree topologies (implemented in OpenSM). The algorithm aims to generate LFTs that evenly spread shortest-path routes across the links in the network fabric. The algorithm traverses the fabric in the *indexing order* and assigns target LIDs of the end nodes, and thus the corresponding routes, to each switch port. For the end nodes connected to the same leaf switch, the indexing order depends on the switch port to which the end node is connected (port numbering sequence). For each port, the algorithm maintains a *port usage counter* and uses it to select the least-used port each time a new route is added (if more than one option is available). If there are multiple ports connecting the same two switches, the ports form a *port group*. In that case, the least loaded port of the least loaded port group is selected to add a new route.

4. Routing Challenges on Multi-Tenant Fat-Trees

In this section, we outline the challenges and issues of routing done oblivious to the presence of nodes belonging to different tenant groups in the IB subnet. Oblivious routing may result in both sub-optimal network utilization and higher inter-partition interference.

Recall from Section 3.1.2, in a partitioned subnet the nodes that are not members of a common partition are not allowed to communicate. Practically, this means that some of the routes assigned by the oblivious fat-tree routing algorithm are not used for the user traffic[3]. However, the algorithm will generate LFTs for those routes the same way it does for the other functional paths. This obliviousness may result in degraded balancing on the links, as nodes are routed in the order of indexing. The effect of the indexing order on the fat-tree load-balancing can be found in our previous work [34]. Furthermore, as routing is done oblivious to the partitions, fat-tree routed subnets provide poor isolation among partitions.

[3]These *stale* routes still need to be entered in the LFTs so that the management traffic can find the paths through the subnet.

To further elaborate on the issues of degraded load-balancing and poor isolation, we present two fat-trees routed by the fat-tree routing algorithm, as shown in Figure 3. The Figure 3(a) shows a 2-level fat-tree topology with four switches and six end nodes in three overlapping partitions. Partition 1 has nodes B and C - Partition 2 has A, C, D and F - and Partition *3* has nodes D and E. We see that the partitions *1* and *3* are confined within the leaf switches *L1* and *L2*, respectively. Hence, the communication between nodes in these partitions takes place through their corresponding leaf switches without moving traffic to the root switches, *R1* or *R2*. When this topology is routed by the fat-tree routing algorithm, the routes towards end nodes connected to the leaf switches, *L1* and *L2*, are assigned to the selected root switches, so the inter-leaf switch traffic can find its way in the topology. For load-balancing, the routes towards A and C are assigned to root switch *R1* (link p), while the root switch *R2* routes traffic towards node B (link q). Similarly for the leaf switch *L2*, D and F, in inter-leaf switch partition *2* are routed via the root switch *R1* (link r); and the traffic towards node E is routed via *R2* (link s).

The end port selection on the root switches is shown as the small circles annotated with the node identifiers. We see that, as the routing is done without considering the partitioning information, the paths in the subnet are not balanced properly. Links p and r are oversubscribed, while no intra-leaf switch flow will ever use link q or s. The routes assigned towards nodes B and E are not utilized (except for the relatively low management traffic) as none of the nodes can receive any communication from outside their leaf switches, due to partitioning. This balancing issue also occurs in fat-trees when a partition's communication is restricted to only some of the levels in the topology.

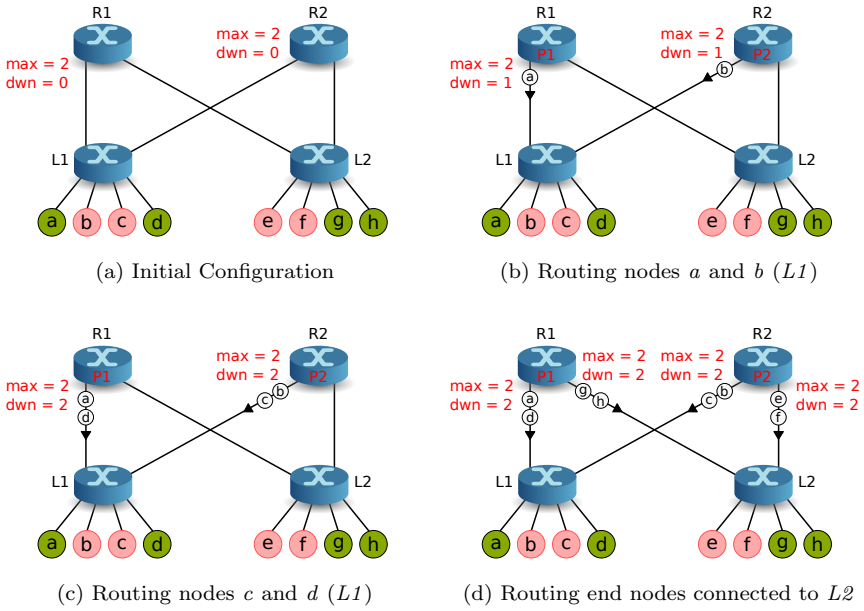
Now, refer to the fat-tree shown in the Figure 3(b). The fat-tree has two partitions, each having two nodes connected to each of the leaf switches. The fat-tree routing algorithm assigns downward ports on the root switches *R1* and *R2*, as shown in the figure. We see that each root switch routes traffic towards nodes belonging to both partitions. For example, the traffic towards nodes A and C is routed on the shared link p. The sharing of intermediate links between nodes of different partitions causes interference among them. Note that the network has adequate resources at the root level to provide complete isolation among partitions, in this case. The partitions can be isolated by partition-aware selection of the ports for the end nodes, without affecting the load-balancing on the links.

5. Partition-aware Fat-Tree Routing Algorithm

For the sake of completeness, we briefly present the partition-aware fat-tree routing algorithm (pFTree). A more detailed description and evaluation of the algorithm can be found in [18].

The pFTree routing algorithm aims to achieve two objectives in order of priority: first, it generates well-balanced LFTs for fat-tree topologies by distributing routes evenly across the links in the tree; second, while maintaining routes on the links balanced, pFTree removes contention between paths belonging to different partitions. The pFTree uses partitioning information about the subnet and ensures that the nodes in a partition receive a predictable network performance, unaffected by the workload running in other partitions. If the topology does not have enough links available to provide partition isolation (without compromising on the load-balancing), the pFTree assigns VLs to reduce the impact of contention.

The algorithm works recursively to set up LFTs on all relevant switches for the LIDs associated with each end node. After filtering out single leaf switch partitions, for each leaf switch, the algorithm sorts connected end nodes in a partitioning specific order. This ordering ensures that the nodes are routed according to their partitions, considering the available number of up-going ports at a leaf switch. The port selection at each level is based on the the least number of already assigned routes to make sure that the load is spread across the available paths. However, when several ports are available with the same load, the function iterates through these least-loaded ports and selects a port which is connected to a switch that is already marked with the partition key of the node being routed. If no switch is marked (means we are routing the first node for this partition), it falls to the default selection of the port with the highest *globally unique identifier* (GUID). When a switch is selected the first time for a partition, it is marked with the partition key. In this way the algorithm ensures that, given that enough paths are available for balancing, the nodes belonging to one partition will be routed through the same switches and corresponding links. Once the routing tables are generated, keeping the partition isolation criteria, the algorithm moves on to check if some of the links



(a) Initial Configuration

(b) Routing nodes a and b ($L1$)

(c) Routing nodes c and d ($L1$)

(d) Routing end nodes connected to $L2$

Figure 4: Port selection in the pFTree routing algorithm

are being used for flows towards nodes in different partitions. For those cases, the algorithm assign VLs to the interfering partitions to provide isolation.

The port selection in the pFTree routing is shown in Figure 4 with the help of a simple section of an oversubscribed fat-tree network. As shown in Figure 4(a), the example section consists of two leaf switches ($L1$ and $L2$), each connected to four end nodes and two switches at the next level above the leaf switches ($R1$ and $R2$). We also show variables **dwn** and **max**, representing the number of assigned routes in the downward direction, and the maximum number of nodes that can be routed to ensure proper balancing on each link, respectively. The end nodes are shown in green and pink colors to represent the two different partitions they belong to. Given that there are two up-going ports at each leaf switch with four end nodes to route, each of the up-links should route two end-nodes down to ensure that the links are balanced ($max = 2$).

For leaf switch $L1$, the routing of the first two nodes, a and b, is shown in Figure 4(b). The algorithm selects switch $R1$ to route traffic towards node a and mark the switch with node a's partition key, represented as $P1$ in the figure. Similarly, for node b, the switch $R2$ is selected and marked with b's partition key ($P2$). The variable dwn is also updated to count a single routed node on each of the two downward links. Now for the other two nodes, c and d, the switch which is already marked with the corresponding partition key is selected, as given in Figure 4(c). The resultant routing routes flows towards nodes belonging to the first partition, a and d, with the same link through switch $R1$. Similarly, the nodes of the second partition, b and c, are routed downwards through $R2$. This separation of the links avoids any interference between the traffic flows of the two partitions. Note that the number of nodes routed downwards on each links does not exceed the max variable, which means that the routing is still perfectly balanced.

Figure 4(d) shows routing for the end nodes connected to the leaf switch $L2$. As the second-level switches are already marked with the partition keys from the first leaf switch routing, the corresponding switches are selected to route each of the nodes e, f, g and h. As we see in the figure, the final routing has isolated the two partitions by dividing the intermediate network links into two equal sized logical sub-networks based on the routing itself.

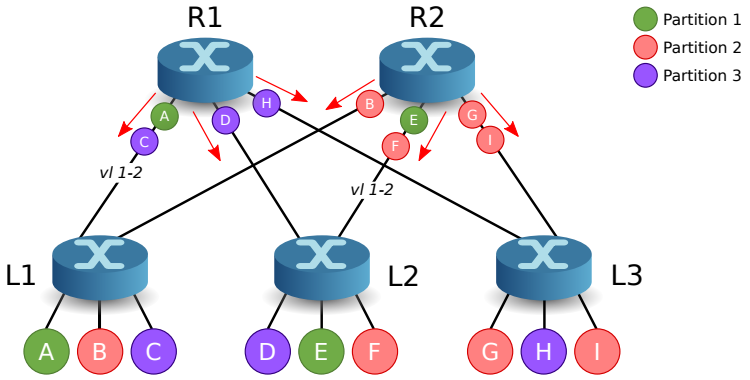


Figure 5: The pFTree routing without partition-wise policies.

6. Extended pFTree Routing Algorithm (pFTree-Ext)

When the network does not have enough resources to isolate partitions solely at the physical link level, the pFTree routing algorithm uses VLs to reduce inter-partition interference. However, different partitions may have different isolation needs depending on the corresponding SLAs or QoS requirements. For example, some of the partitions in the network may be running critical operations, and may require complete physical isolation in all cases. Similarly, in many networks, depending on the availability of the VLs, some partitions may have to share a VL with another partition, which may not be desirable for communication-intensive workloads. The pFTree algorithm is unable to specify the aforementioned partition-wise requirements in the routing, and all partitions are treated with equal priority assuming similar QoS requirements.

To further elaborate on the need of partition-wise policies, consider a fat-tree network with nine nodes in three different tenant partitions, as shown in Figure 5. The nodes belonging to each of the partitions are shown using a different color (Partition *1* as green, Partition *2* as pink, and Partition *3* as purple). Now consider that Partition *1* has very high QoS requirements, and that it is critically important that the workload running in this partition is not affected by any inter-partition interference. However, as the given fat-tree network has only two root switches *R1* and *R2* while having three different tenant partitions, it is not possible to isolate these partitions solely at the physical level. As described above, in such cases the pFTree routing algorithm will proceed with isolating partitions using VLs. Figure 5 also shows the routing obtained using the default pFTree algorithm, using small node circles just below the switches to denote flows towards the destination nodes. We see that the traffic towards node A of partition *1* has to share link $R1 \rightarrow L1$ with node C belonging to partition *3*. Similarly, node E shares the link $R2 \rightarrow L2$ with the node F of partition *2*. On both these links, the algorithm uses a separate VL for each of the partitions to provide isolation. Even though the use of a separate VL decreases the interference, it does not completely eliminate it (Refer Section V-B of [18]), thus, the pFTree routing algorithm fails to satisfy the requirements of partition *1*.

To address this issue, we present an extended version of the routing algorithm, which incorporates provider defined partition-wise policies. For example, to meet the high QoS requirements for tenant partition *1* of the previous example, the provider can mark partition *1* as physically isolated in the routing algorithm.

6.1. Isolation Policies

We extend the pFTree routing algorithm to include partition-wise and global isolation policies. For each partition, the isolation policies determine how the nodes in the partition are allowed to share network resources with nodes belonging to other partitions. The global policies determine whether the routing will fail, or continue with best-effort isolation if all partition-wise isolation policies cannot be satisfied on a given network.

The available policy parameters for the extended pFTree routing algorithm are shown in Table 1. Each partition can be marked with one of the three partition-wise policy parameters. Marking a partition with

Algorithm 1 The pFTree-Ext Routing Algorithm

Ensure: The LFTs are generated for the switches conforming isolation policies

```
1: global_param ⇐ get_global_isolation_policy()
2: partitions_info ⇐ get_partition_information()
3: OrderComputeNodes()
4: for each sw ∈ leafSwitches[] do
5:    for each cn ∈ computeNodes[] do
6:       Get lid of cn
7:       Get partition_key of the cn.hca_port
8:       Set LFT[lid] ⇐ cn.hca_port on sw
9:       RouteDowngoingByAscending() on sw
10:   end for
11: end for
12: AssignVirtualLanes()
13: ValidatePolicies()
```

phy-isolation guarantees that the routing algorithm reserves network resources specifically for the partition, and no nodes in the partition will share any link with any other node in a different partition. The parameter *vlane-isolation* allows a partition to share the network resources with other partitions using a separate VL only. The *def-isolation* scheme implements best-effort isolation for the marked partition. The global policy parameters, *strict* and *best-effort*, define whether the routing algorithm fails or falls back to the best-effort routing when partition-wise policy parameters cannot be satisfied in a given subnet. For example, when the network does not have enough links or VLs for providing the desired isolation. The policy parameters are provided to the routing algorithm using a partition configuration file.

6.2. The Algorithm

The extended pFTree routing algorithm (pFTree-Ext) works the same way as the original pFTree, by recursively traversing the fabric to set up LFTs in all switches for the LIDs associated with each end node. However, unlike pFTree, it also considers the defined global and partition-wise isolation policies when assigning routes.

The pseudo code of the pFTree-Ext routing algorithm is shown in Algorithm 1. Note that the algorithm is deterministic and that the routes are calculated backwards, starting at the destination nodes. The algorithm first sorts compute nodes in a partition specific order (Algorithm 1, line 3). The partition specific order

Parameter	**Scope**	**Definition**
phy-isolation	Partition	Nodes in this partition cannot share links with any other partition in the network.
vlane-isolation	Partition	Nodes in this partition can share links with other partitions using a separate virtual lane only.
def-isolation	Partition	The default best-effort isolation, given available network resources.
strict	Global	Routing fails if any of the partition-wise policies cannot be satisfied.
best-effort	Global	Routing continues when partition-wise policies cannot be satisfied with the best-effort isolation, logging a warning message.

Table 1: The pFTree-Ext isolation policy parameters.

Algorithm 2 OrderComputeNodes()

Require: List of switches and attached compute nodes
Ensure: The compute nodes are ordered for the pFTree-Ext routing algorithm

```
 1: for each sw in leafswitches[] do
 2:   num_up_ports ⇐ count(sw → upPorts[])
 3:   num_cns ⇐ count(sw → computeNodes[])
 4:   Sort nodes in increasing order of partition isolation policy (phy > vlane > def)
 5:   if num_cns ≤ num_up_ports then
 6:     return
 7:   end if
 8:   index_arr[] = array(num_cns)
 9:   taken[] = array(num_cns)
10:   pkey_tbl[] = map()
11:   id ⇐ 0
12:   for each cn in sw → computeNodes[] do
13:     pkey ⇐ cn → get_partition_key()
14:     if pkey not found in pkey_tbl then
15:       if taken[id] ≠ false then
16:         id ⇐ get_free_id()
17:       end if
18:       index_arr[cn[i]] ⇐ id
19:       taken[id] = true
20:       insert pkey in pkey_tbl
21:     else {pkey is already in pkey_tbl}
22:       id ⇐ id(pkey) + num_up_ports
23:       if id ≥ num_cns or taken[id] = true then
24:         id ⇐ get_free_id()
25:       end if
26:       index_arr[cn[i]] ⇐ id
27:       taken[id] = true
28:       update pkey_tbl
29:     end if
30:   end for
31:   Sort sw → computeNodes[] with respect to index_arr[]
32: end for
```

ensures faster execution of the algorithm, as once the nodes are ordered, they can be routed iteratively without maintaining maximum counters on each down-going and up-going port. As shown in Algorithm 2, for each leaf switch, OrderComputeNodes first sorts end nodes in the increasing order of their partition policy priority (Algorithm 2, line 4). The nodes belonging to the partitions marked with *phy-isolation* parameter are added first, while partitions with *vlane-isolation* are added second. Finally, the partition nodes with policy parameter value of *def-isolation* are added to the list of compute nodes. The algorithm then uses partitioning information of the nodes to generate a routing order where nodes belonging to one partition tends to get indices suggesting same up-going links in the network on iterative routing. This is done by adding the number of available up-going ports to the index chosen to route the first node belonging to a partition, using a partition key table (Algorithm 2, line 14-28). However, when such an index is already taken or the index is beyond the compute array bounds, the first free index is chosen and marked with the partition key for later selections (Algorithm 2, line 24).

Once the nodes are properly ordered, the pFTree-Ext calls RouteDowngoingByAscending (Algorithm 1, line 9) and moves up in the tree to select a port at the next level to route the LID in the downward

Algorithm 3 RouteDowngoingByAscending()

```
Require: A switch sw, an end node lid and partition_key
 1: Sort sw.upPorts[] with increasing load and then GUID
 2: Get least loaded ports as leastLoadedList[]
 3: partition_param ⇐ get_isolation_policy(partition_key)
 4: selected_port ⇐ null
 5: for each port in leastLoadedList[] do
 6:    r_sw ⇐ port.get_remote_switch()
 7:    if r_sw is marked with partition_key then
 8:       selected_port ⇐ port
 9:       break
10:    end if
11: end for
12: if selected_port = null then
13:    while selected_port = null do
14:       port ⇐ sw.upPorts[].get_next()
15:       r_sw ⇐ port.get_remote_switch()
16:       if r_sw is marked with a partition with isolation policy > partition_param then
17:          continue
18:       end if
19:       selected_port ⇐ port
20:    end while
21: end if
22: Set LFT[lid] ⇐ selected_port on r_sw
23: if r_sw is not marked then
24:    Mark it with partition_key in DWN direction
25: end if
26: RouteUpgoingByDescending() on sw
27: RouteDowngoingByAscending() on r_sw
```

direction, as shown in the Algorithm 3. The port selection is first based on the least-loaded port list obtained from the sorted available up-going ports (Algorithm 3, line 1-2). The function iterates through these least-loaded ports and selects a port which is connected to a switch that is already marked with the partition key of the node being routed (Algorithm 3, lines 5-11). If no switch is found marked, the algorithm iterates through all the up-going ports to find a suitable route for the LID. The up-going port list is sorted in the increasing order of the current load on the ports. For the ports with same load, sorting is done in decreasing order of their *globally unique identifiers* (GUIDs) in order to remain deterministic. Furthermore, the function does not select a port which is already marked with a partition key with a higher isolation policy parameter than the routed node (Algorithm 3, line 16-17). Finally, when a port is selected, the corresponding switch is marked in the downward direction with the partition key (Algorithm 3, line 24).

After the down-going port is set for a LID at a switch, the algorithm assigns upward ports for it on all the connected downward switches by descending down the tree calling RouteUpgoingByDesc (Algorithm 4). Again, the selection of the up-going port is first based on the load criterion and then on the partition marking of the remote switches, in the upward direction this time. The process is then repeated by moving up to the next level in the tree until all LFTs are set. Note that a switch can be marked with multiple partition keys. The pFTree-Ext algorithm maintains a table for each switch, storing the count of routed nodes for each partition. This counter is used to decide the selection of the port if several switches with marked partitions are available to route a node. The switch with the maximum number of already routed nodes for a partition is selected.

Algorithm 4 RouteUpgoingByDescending()

Require: A switch *sw*, an end node *lid* and *partition_key*
1: Get least-loaded ports from *sw.dwnPorts[]* as *dwnlist[]*
2: *selected_port* ⇐ *dwnList.get_port_max_guid*()
3: **for each** *port* in *dwnList*[] **do**
4: *r_sw* ⇐ *port.get_remote_switch*()
5: **if** *r_sw* is marked with *partition_key* **then**
6: *selected_port* ⇐ *port*
7: **break**
8: **end if**
9: **end for**
10: **if** *r_sw* is not marked **then**
11: Mark it with *partition_key* in UP direction
12: **end if**
13: Set LFT[*lid*] ⇐ *selected_port* on *r_sw*
14: RouteUpgoingByDescending() on *r_sw*

Algorithm 5 AssignVirtualLanes()

Require: The pFTree-Ext routing tables have been generated
Require: Switches have been marked with the partition keys
Require: Global policy parameter, *strict* or *best-effort*
Ensure: A partitions marked with *vl-isolation* has a separate VL
Ensure: No two partitions with the same SL share a link
1: *vlanes_needed* ⇐ 1
2: *max_vlanes* ⇐ *get_max_lanes*()
3: *strict* ⇐ *get_is_strict*()
4: **for each** *partition* in *partition_tbl* **do**
5: check if the isolation policy of the partition is *vl-isolation* and any intermediate communication link in this partition share a switch with a partition that has not been assigned a virtual lane
6: **if** require a separate vl **then**
7: **if** *vlanes_needed* = *max_vlanes* and *global_param* = *strict* **then**
8: *vlanes_needed* ⇐ 1
9: **else**
10: **error**: routing failed
11: **return**
12: **end if**
13: *vlanes_needed*++
14: *partition.vlane* ⇐ *vlanes_needed*
15: **end if**
16: **end for**

Once the routing tables are generated, keeping the partition isolation criteria, the algorithm moves on to check if some of the links are being used for flows towards nodes in different partitions. For those cases, the algorithm assign VLs to the interfering partitions to provide isolation. The VL assignment function is shown in Algorithm 5. The function iterates through all partitions and checks if the partition is marked with the *vl-isolation* policy parameter, and if any intermediate communication links used by the nodes in the partition shares an intermediate link with another partition that has not been assigned a separate VL. If so, a new VL is assigned. The VL assignment function also uses global policy parameters with two modes: *strict* and *best-effort*. In the strict mode, if the number of required VLs for pFTree-Ext routing exceeds the number of available VLs in the system, the routing fails (Algorithm 5, line 10). In best-effort mode, the

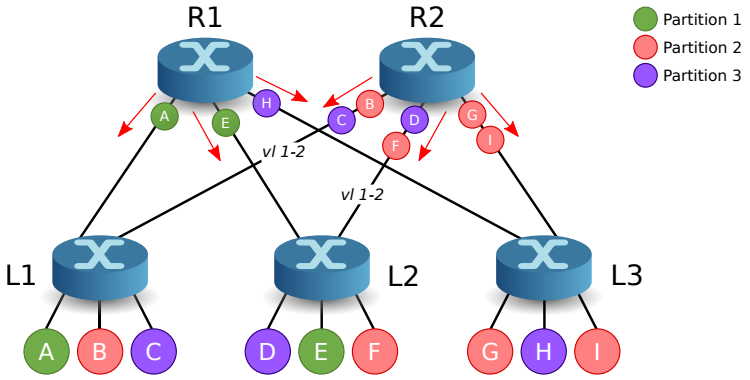


Figure 6: Extended pFTree routing with Partition 1 marked as physically isolated.

function restarts assigning VLs to the partitions from VL_1 (Algorithm 5, line 8). Note that the algorithm can easily be modified to consider a particular group of VLs, rather than all available VLs. Similarly, to make it less likely for partitions with higher isolation policies to share VLs, once all available VLs are used, the VL list can be ordered by decreasing priority of assigned partitions for selection (instead of selecting VL_1). After the VLs are assigned, the pFTree-Ext routing algorithm validates whether all the partition-wise and global policies are met (Algorithm 1, line 13).

Discussion

The pFTree-Ext incorporates isolation policies into the routing algorithm in the following ways:

(i) Unlike pFTree, which for each leaf switch sorts end nodes in the partition-specific order before routing, the pFTree-Ext routing algorithm first sorts end nodes in the order of their partition priorities. The end nodes in the partitions marked with *phy-isolation* get the maximum priority. After that, the algorithm proceeds by sorting end nodes in partition specific order as earlier. The additional sorting is done upfront to ensure that the nodes with the highest partition priorities are routed first.

(ii) The pFTree-Ext algorithm also changes the way a port is selected for routing a new node. For example, to select a down-going port among several candidate ports, the pFTree-Ext, besides checking the current load on the port, removes any port-group where the corresponding switch has already been marked with the key of a partition with a higher priority than the partition of the node currently being routed.

(iii) If the available network resources do not allow the partition-wise policy parameters to be satisfied, the pFTree-Ext routing algorithm either fails or proceeds according to the global policy parameters described above. The original pFTree routing algorithm only considers the available VLs in that case.

Recall the example fat-tree network of Figure 5, but this time with routing using pFtree-Ext routing algorithm, and partition *1* marked as *phy-isolation*. The resultant routing is shown in Figure 6. Note that now none of the partition *1* nodes *A* and *E* share links with any other partition. However, as no such policy was applied to partition *2* and partition *3*, these partitions share all down-going links from switch *R2*.

6.3. Evaluation

We have implemented the pFTree-Ext routing algorithm in OpenSM *v3.3.16*. The partition membership of the ports and the isolation policies are provided to the OpenSM using a partition file. To evaluate pFTree-Ext, we run both real-world experiments on a small cluster and simulations. Experiments are performed to show how the performance of flows running in a *victim* partition is affected by the workload of other partitions after the application of isolation policies; Simulations are run for large topologies to complement the results we obtain from our test cluster. In all cases, we compare pFTree-Ext with the original pFTree routing algorithm presented in [18].

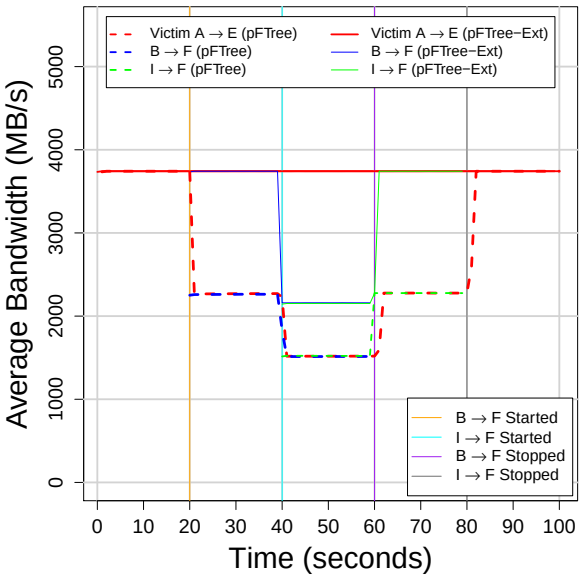


Figure 7: The effect of interference on victim Partition *1*, pFTree and pFTree-Ext.

6.3.1. Experiments

Our test setup is equipped with nine compute nodes and four switches. We use a mix of SUN Fire X2270 and HP ProLiant DL360p servers. Each node runs Ubuntu 12.04 LTS and is connected to the IB network using a Mellanox ConnectX-3 VPI adapter. We use the OFED software stack on top of Ubuntu to enable IB communication. We use SUN DCS 36 switches. All links operate at $4 \times QDR$ (40 Gb/s) speed.

We take the same fat-tree topology as shown in Figure 5 earlier in this section. Partition *1* with nodes A and E is marked as the victim partition, while we run some flows in partition *2* to evaluate the performance of both the original pFTree and the pFTree-Ext routing algorithms. Partitions are thus chosen to demonstrate the impact of interference in a typical case. However, the algorithm works equally well for any partitioning. We use OFED's IB performance testing utility, *perftest*, for our bandwidth measurements.

The results from the experiment are shown in Figure 7. The vertical color lines in the figure marks the events when a flows in the interfering partition is started or stopped, as explained in the legend at the bottom-right of the figure. When using the pFTree routing, the bandwidth of the victim flow $A \rightarrow E$ (shown as dashed red line) drops from 3742 MB/s to 2270 MB/s when the interfering flow $B \rightarrow F$ is started at time 20s (shown as dashed blue line). Furthermore, when another interfering flow $I \rightarrow F$ starts at time 40s, the victim flow bandwidth further drops to around 1518 MB/s. The bandwidth of the victim flow eventually recovers when the interfering flows are stopped at time 60s and 80s, respectively. However, for pFTree-Ext, with partition *1* marked as physically isolated, the bandwidth for the victim flow $A \rightarrow E$ remains unaffected by the flows in the interfering partition (shown as a solid red line). We achieve a constant 3742 MB/s bandwidth for the $A \rightarrow E$ flow when pFTree-Ext routing is in use.

6.3.2. Application Benchmarks

In order to see the effect of partition-wise isolation policies implemented by the pFTree-Ext routing algorithm on the performance of HPC applications, we use the NAS parallel benchmark (NPB) suite [35]. We employ a similar fat-tree topology as used in Section 6.3.1, where each leaf switch is connected to three end nodes. However, we use four leaf switches instead of three to satisfy the requirement of processing nodes in the power of 2 for the NPB applications. One of the end nodes on each leaf switch belongs to the partition that runs the NPB application benchmarks, while the rest of the nodes in the topology are used to create interference, as shown in Figure 8.

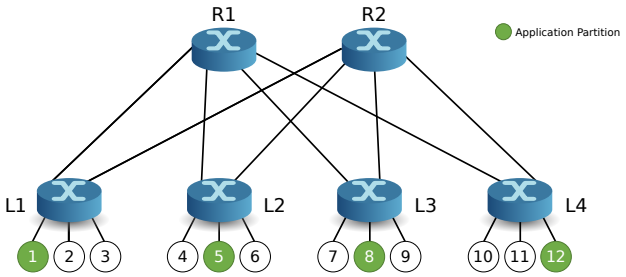


Figure 8: Topology for NPB application benchmarks

		Mean Time (seconds)	
Benchmark Application	**Class**	**pFTree**	**pFTree-Ext**
Conjugate Gradient (CG)	A	0.228	0.140
	B	10.382	8.936
	C	28.276	25.902
Embarrassingly Parallel (EP)	A	1.400	1.400
	B	5.574	5.534
	C	22.148	22.212
Fourier Transform (FT)	A	0.744	0.712
	B	8.890	8.394
	C	36.310	34.088
Integer Sort (IS)	A	0.164	0.120
	B	0.646	0.484
	C	2.820	2.156
Multi-Grid (MG)	A	0.274	0.262
	B	1.284	1.230
	C	10.612	10.568

Table 2: Completion times for NPB application benchmarks

The NPB benchmarks are derived from computational fluid dynamics (CFD) applications and consist of several kernels. We use the following benchmark kernels:

Conjugate Gradient (CG): In this benchmark, the conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large and sparse definite matrix. The kernel largely employs high irregular point-to-point communication.

Embarrassingly Parallel (EP): The EP benchmark generates pairs of Gaussian random deviates and tabulate the number of pairs in successive square annuli. The benchmark requires no inter-node communication, except for the coordination of pseudo-random number generation at the beginning and compilation of the results at the end.

Fourier Transform (FT): The FT kernel solves a 3D partial differential equation applying Fast Fourier Transformations (FFTs). The benchmark exercises heavy use of MPI collective operations, mainly all-to-all communication.

Integer Sort (IS): The IS kernel performs large integer sort operation, and tests both random memory access and communication performance.

Multi-Grid (MG): The MG benchmark performs multi-grid operation on a sequence of meshes. The kernel is memory-intensive and requires structured data communication.

More details about the NAS parallel benchmarks can be found in [36].

In our application partition, we use 4 cores per end node (one MPI process per core) to run benchmark applications. Furthermore, we run three reference problem sizes for each benchmark, classes A, B, and C,

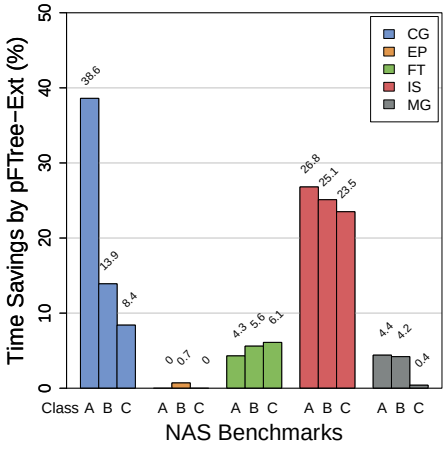


Figure 9: Percentage savings in time by pFTree-Ext as compared to pFTree

where the problem size roughly increases four times going from one class to the next. The inter-partition interference is created by starting eight random flows on non-application nodes using the IB *perftest* utility. For the pFTree-Ext algorithm, we mark the application partition as physically isolated. The benchmark completion times for both the pFTree and the pFTree-Ext routing algorithms are given in Table 2. We see that, using pFTree-Ext routing, the benchmarks benefit from an isolated application partition, saving substantial time to completion. The percentage improvements in benchmark completion times are shown in Figure 9.

The job completion time for the CG benchmark, which involves high irregular point-to-point communication, improves by about 38.6% for the problem size of class A. As the problem size increases, the percentage improvement drops due to our small cluster setup taking longer time in the computational part of solving the linear equations. Still the improvement of about 13.9% and 8.4% is observed for class B and C, respectively. Similarly, for the IS application the job completion times improves by 26.8%, 25.1%, and 23.5% for the three classes in increasing order of problem sizes. The improvement, as given by the pFTree-Ext routing algorithm, is less affected by the problem sizes in the IS benchmark because of relatively faster integer sort computation on our server machines. As shown in Table 2, the IS benchmark takes less than 3 seconds to complete for class C problem, as compared to CG taking more than 25 seconds for the same problem class.

When using pFTree-Ext, the MG benchmark completion times improve by a bit more than 4% for the problem classes A and B. However, for class C, the improvement is reduced to only 0.4% due to higher computational cost and memory-intensive operations. On the other hand, the FT benchmark, which mainly uses all-to-all communication with some reduce operations, yields better performance improvement as the problem size increases. For the class C problem, pFTree-Ext improves the job completion time by 6.1% over the pFTree routing, as compared to 4.3% improvement for the problem size A. This is due to higher data communication with bigger problem sizes, and collective operations involving all processes in the communication at the same time.

As shown in Figure 9. the EP benchmark application, which involves very low communication among processes, does not show any substantial improvement by pFTree-Ext over the original pFTree routing algorithm.

6.3.3. Simulations

For large scale simulations, we use the *Oblivious Routing Congestion Simulator* (ORCS) [37]. The ORCS is capable of simulating a variety of communication patterns on statically routed networks, and has been used extensively in the literature to evaluate and compare the efficiency of routing algorithms in IB based network topologies [28, 38]. In [18], we extended ORCS to make it possible to run patterns within partition

Total Nodes	XGFT(h;m;w)	Oversub Ratio	Victim Nodes
32	XGFT(2;8,4;1,4)	2:1	8
48	XGFT(2;12,4;1,4)	3:1	12
64	XGFT(2;16,4;1,4)	4:1	16
128	XGFT(2;16,8;1,8)	2:1	32
192	XGFT(2;24,8;1,8)	3:1	48
256	XGFT(2;32,8;1,8)	4:1	64
512	XGFT(2;32,16;1,16)	2:1	128
768	XGFT(2;48,16;1,16)	3:1	192
1024	XGFT(2;64,16;1,16)	4:1	256

Table 3: Topologies for simulations.

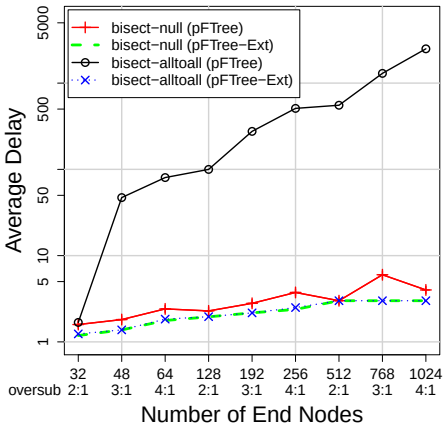
boundaries. Furthermore, we use OFED's *ibsim*, a tool that is distributed with the OFED software stack, to emulate physical topologies for generating routing tables.

We choose several topologies with different over-subscription ratios for our simulations, as shown in Table 3. Each of our test topologies is based on a k-ary-n-tree, where we increases the nodes connected to each leaf switch according to the over-subscription ratio. The number of *victim nodes* refer to the number of nodes we assign to the victim partition. The rest of the nodes are put in the interfering partition. The victim nodes are chosen randomly for each simulation. However, to focus on the impact of the interference on the intermediate links, we distribute the victim nodes evenly in all the leaf switches. The victim partition is assigned the partition-wise policy 'phy-isolation' parameter in pFTree-Ext.

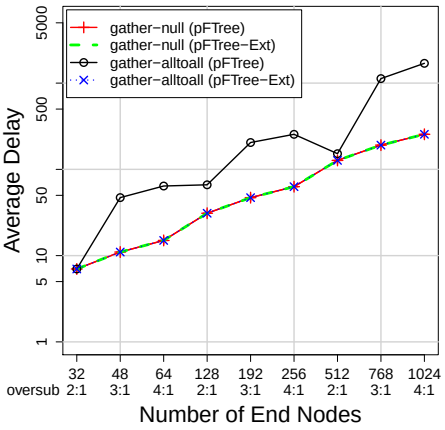
We test several communication patterns in the partitions, for both *noiseless* and *noisy* cases. The **null** pattern is used to record the delay in the noiseless case without any interfering communication, while the **alltoall** pattern in the interfering partition represents the noisy case. All patterns are simulated 50000 thousand times with randomly chosen nodes in both the partitions to eliminate the effect of node selection. The following patterns are used in the victim partition:

- The **bisect** pattern emulates effective bisection bandwidth. In this pattern, nodes in a partition are split into two equal sized halves. Each node in the first half sends a message to a node in the second half.
- In **gather** pattern, one randomly selected node receives a message from all the other nodes in the partition simultaneously.
- In **scatter** communication, one randomly selected node sends a single data message to all the other nodes in its partition.
- The **bisect_fb_sym** communication pattern works the same way as the bisect pattern except that the communication is bidirectional in this case.
- The **alltoall** communication pattern represents a bandwidth-intensive pattern where each node sends a message to all other nodes in its partition.

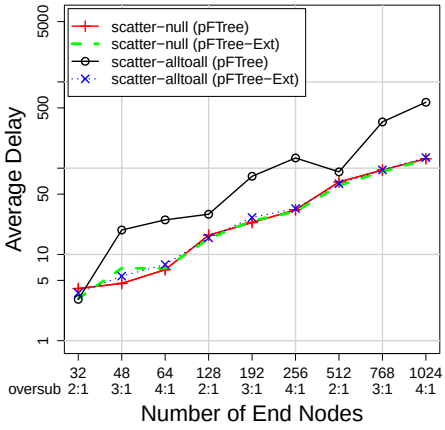
The ORCS supports several metrics to reduce the data obtained as congestion maps in a single result. We are particularly interested in the *dep_max_delay* metric it supports. The *dep_max_delay* metric is used to study the impact on one communication pattern, running in one group on nodes, caused by another communication pattern that is being run in a second and different group of nodes. The simulator examines the congestion in only the first group, and reports the delay the victim pattern experiences because of the interference from the communication in the other group. More details about the communication patterns and metrics supported by ORCS are given in [37] and [39].



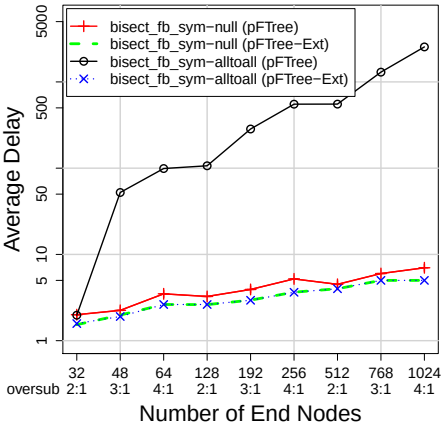
(a) Bisect Pattern



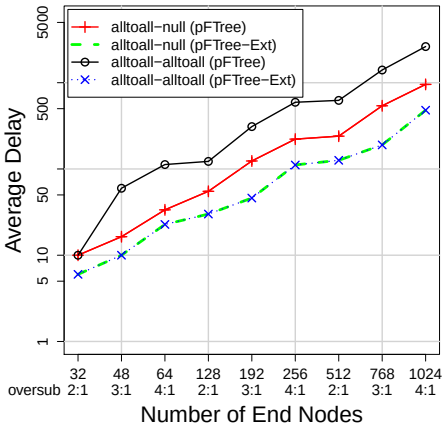
(b) Gather Pattern



(c) Scatter Pattern



(d) Bisect-Fb-Sym Pattern



(e) All-to-All Pattern

Figure 10: The effect of interference on the victim partition.

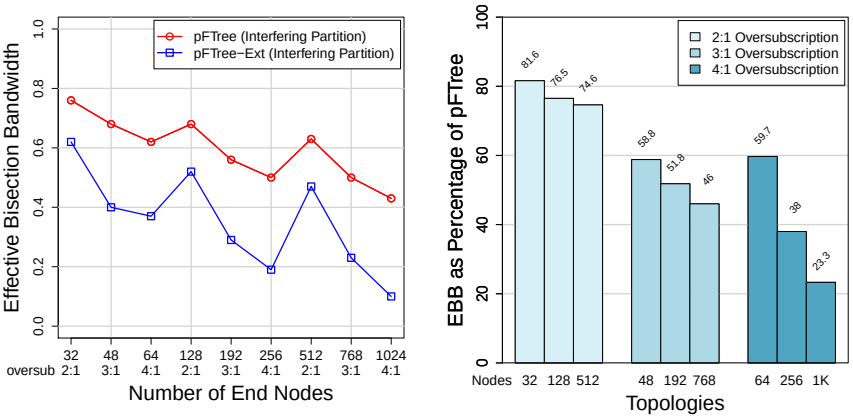
The results from the simulations are shown in the Figure 10. The Figure 10(a) shows the average delay we measured in the the victim partition when running bisect communication pattern. The *bisect-null* combination, shown as the red solid line with plus for the pFTree, and green dashed line for the pFTree-Ext routing, shows delay less than 10 units for all our test topologies. In addition, the pFTree-Ext yields slightly lower delays, due to the improved route selection for the victim partition. However, the greater impact is observed in the noisy case when the *alltoall* pattern is run in the interfering partition. The pFTree routing algorithm experiences exponentially increasing delays with the increase in the size of the topology and its oversubscription ratio. The delay reaches up to 2498 units (shown as the solid black line with circles) for the topology with 1024 nodes on 4 : 1 oversubscription. Note that y-axis is logarithmic. On the other hand, for the pFTree-Ext routing, we observe no change in the delay when compared to the *bisect-null* case (dotted blue line with crosses), for all the topologies. As the victim partition has been marked as physically isolated, there are no links in the topology that share flows from both the victim and the interfering partitions.

Similarly, there is no change in the delay observed for the pFTree-Ext routing using *gather* and *scatter* patterns, shown in Figure 10(b) and Figure 10(c), respectively. The change in the delay for the pFTree routing between the *null* and the *alltoall* cases, however, varies significantly depending on the oversubscription ratio of the topology. For topologies with low oversubscription ratio (2 : 1), the change in delay is comparatively small. On the other hand, for topologies with 4 : 1 oversubscription ratio, the delay between the two cases varies greatly. For example, for the *gather* pattern, on 512 nodes with 2 : 1 oversubscription, the average delay on *gather-null* and *gather-alltoall* is 127 and 152, respectively. The delay increases from 255 to 1696 ($>$ 6 times) between the *null* and the *alltoall* cases on the topology with 1024 nodes having 4 : 1 oversubscription. The reason for this change is that with higher oversubscription ratio and more nodes in the network in both the victim and the interfering partitions, the congestion on the root nodes for *gather* and *scatter* is far more severe with a denser *alltoall* pattern running in the interfering partition.

Similar trends are observed in Figure 10(d) and Figure 10(e) for *bisect_fb_sym* and *alltoall* patterns, respectively. Since all links in our simulations have equal capacity in both directions, *bisect_fb_sym* shows the same average delays as observed with the bisect pattern. The *alltoall* pattern, which is the most communication-intensive pattern in our tests, reports a linear growth of delay for the pFTree routing. The delay for *alltoall* case is roughly twice the delay of the *null* case on most of the topologies. However, for the pFTree-Ext routing algorithm, both cases have the same average delay in the victim partition, which is marked physically isolated. Furthermore, even for the *null* case, the pFTree-Ext reduces the delay up to 50% as compared to the pFTree routing.

			pFTree			**pFTree-Ext**		
Nodes	**Oversub**	**Pattern**	**min**	**avg**	**max**	**min**	**avg**	**max**
64	2:1	bisect	1	99.9	111	1	2	4
		bisect-fb-sym	61	106.6	112	1	2.6	4
		gather	31	66.2	112	31	31	31
		scatter	0	29.2	113	0	15.6	31
		all-to-all	96	122.4	128	30	30	30
128	3:1	bisect	126	275.8	291	1	2.2	5
		bisect-fb-sym	251	284.5	293	2	2.9	5
		gather	78	204.8	294	47	47	47
		scatter	0	80.5	293	0	26.8	47
		all-to-all	291	309.8	318	46	46	46
256	4:1	bisect	241	509.3	649	1	2.5	4
		bisect-fb-sym	386	551.9	649	3	3.6	6
		gather	80	254.9	560	63	63	63
		scatter	0	131.3	442	0	34	63
		all-to-all	480	593.3	656	110	111.2	112

Table 4: Delay for the all-to-all victim pattern on a 8-ary-2-tree



(a) EBB for pFTree and pFTree-Ext. (b) Decrease in EBB for pFTree-Ext.

Figure 11: Effect on the interfering partition.

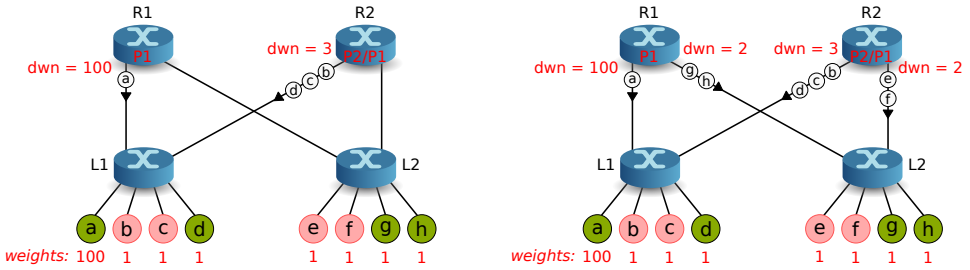
6.3.4. Delay Range and Oversubscription Ratio

To further analyze the effect of the over-subscription ratio on the delay measured in the victim partition, we now present more detailed results for the the topology based on a *8*-ary-2-tree. Table 4 shows the minimum, average, and maximum delays observed using different patterns and over-subscription ratios for both the pFtree and the pFTree-Ext routing algorithms. We see that for pFTree routing on most patterns, as noted above as well, the average delay increases significantly with the increase in the oversubscription ratio. For example, for *bisect* pattern, the average delays increases from 99.9 to 275.8 when the over-subscription ratio is increased from $2:1$ to $3:1$. Further, it goes to 509.3 on the topology with a $4:1$ over-subscription ratio. Another important observation is that the difference between the minimum and maximum delay observed varies significantly. However, for the pFTree-Ext routing, the difference between the delays is minimal. For example, For the *all-to-all* traffic pattern on the topology with a $4:1$ over-subscription ratio, the minimum delay observed is 110, while the average and maximum delay are 111.2 and 112, respectively.

6.3.5. Effect on the interfering partition

We now present the *reversed* simulation results to give an insight into how the interfering partition is affected with the stricter isolation policy on a victim partition. Understandably, using physical isolation for one partition affects the workload performance in the other partitions, as less network resources are then available for the nodes to communicate in the interfering partition. For these simulations, we do not run any traffic pattern in the victim partition, and run a *bisect* pattern in the interfering partition to measure the effective bisection bandwidth (EBB). The results for these reversed simulations are given in Figure 11.

Figure 11(a) shows the EBB in the interfering partition for the pFTree and the pFTree-Ext routing algorithms. We observe that the EBB decreases quite linearly for the pFTree routing with the increase in the oversubscription ratio. For the first three topologies, the EBB decreases from 0.76 to 0.62 as the oversubscription ratio is increased from $2:1$ to $4:1$ (a decrease of 18.42%). However, for the pFtree-Ext routing, the EBB decreases more severely from 0.62 for the $2:1$ oversubscription to 0.37 for the $3:1$, which is a decrease of about 31.7%. Similarly, for our last three topologies, the EBB for pFTree routing decreases about 31.75% with the increase in the oversubscription ratio, while for the same topologies, the EBB for pFTree-Ext decreases about 76.7%. This observation confirms that, for highly oversubscribed topologies, when some of the partitions are marked with physical isolation, the performance of other partitions may be severely affected. The EBB for the pFTree-Ext routing as compared with the pFTree routing in percentage, is shown in Figure 11(b). For the topologies with $2:1$ oversubscription ratio, the pFTree-Ext achieves more than 70% of the pFTree's EBB. This decreases down to only 23.3% for our largest topology with an oversubscription of $4:1$.



(a) Routing nodes connected to *L1*

(b) Routing nodes connected to *L1* and *L2*

Figure 12: Port selection in the pFTree-Wt routing

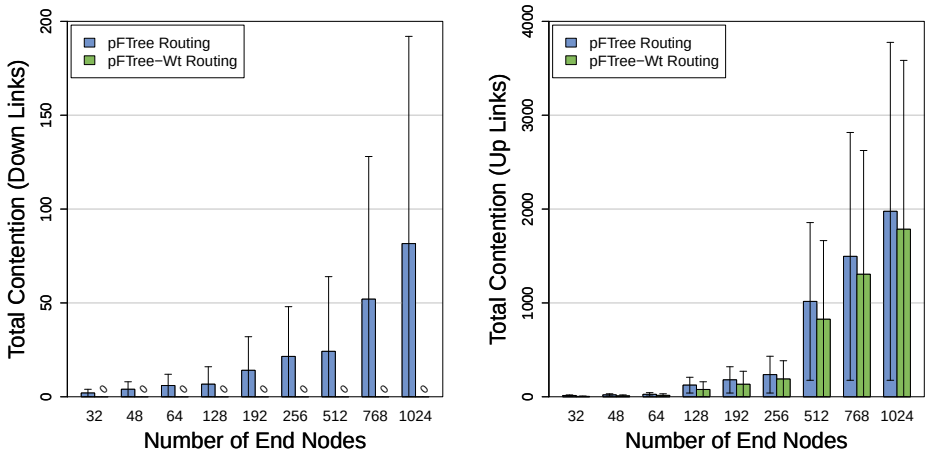
7. Weighted pFTree Routing Algorithm (pFTree-Wt)

We now present our second extension to the pFTree routing algorithm, which we call weighted pFTree routing algorithm (pFTree-Wt). The pFTree-Wt is based on the notion of *weights* associated with each compute node [34]. These weights are used to take known or learned traffic characteristics into account when calculating routes. Irrespective of the partitioning, the weight of a node reflects the degree of priority the flows towards a node receive when calculating routing tables. For example, a possible configuration could be to assign weights to the nodes in the range $[1, 100]$ depending on how much traffic a node is known to receive in the network. Such a scheme could assign $weight = 1$ for the nodes that receive very little traffic (primarily traffic generators, for example), and $weight = 100$ for the nodes receiving traffic near the link capacity. The values in between, $1 < x < 100$, will then reflect the proportion of traffic a node is expected to receive in the network. When no administrative information about the compute nodes is available, weights can be calculated using a simple port data counter based scheme. In OFED, a utility, *ibdatacounts*, is provided for reading data counters. After setting up the network with equal initial weights for all nodes, new weights can be *learned* after a specified time period. If B is the set of receive bandwidths for all the nodes measured over a time period, the weight for each node can be assigned in the range $[a, b]$ by using linear transformation as given by Equation 1.

$$W(x) = (x - a)\frac{b - a}{\max(B) - \min(B)} + a, \forall x \in B \tag{1}$$

The pFTree-Wt routing works as follows. Each compute node is assigned a parameter, *weight*. Unlike the original pFTree routing, where the load on a port represents the number of assigned routes towards nodes in the up and down directions, the load on a port in the pFTree-Wt routing scheme is the accumulated weight of the compute nodes routed from that port in each direction. For each leaf switch, the nodes in one partition are also sorted by their weights before routing. When a downward port at a switch is selected to route a compute node, pFTree-Wt updates the current load on the selected port by adding the weight of the corresponding compute node. Similarly, for the upward links, an upward load is maintained on each port. The port selection criteria is similar to the pFTree routing, and considers the partitions of the node as well. However, unlike port counters, the port selection at each level in pFTree-Wt is based on the least accumulated weight on all the available ports. When several ports are available with the same load, the function iterates over these least-loaded ports and selects a port which is connected to a switch that is already marked with the partition key of the node being routed. The algorithm still tends to isolate the partitions in the network, even though the criteria is more sensitive to the weights of the nodes. Once the routing tables are generated, the pFTree-Wt runs VL assignment to ensure that different VLs are assigned to nodes associated with different partitions sharing links in the network.

Recall the sample fat-tree we used to explain the port selection in the original pFTree routing in Section 5; assuming node a in partition *1* is assigned $weight = 100$, while all other nodes in the subnet have $weight = 1$. The pFTree-Wt routing in the downward direction is shown in Figure 12. As shown in Figure 12(a), when



(a) Down Links

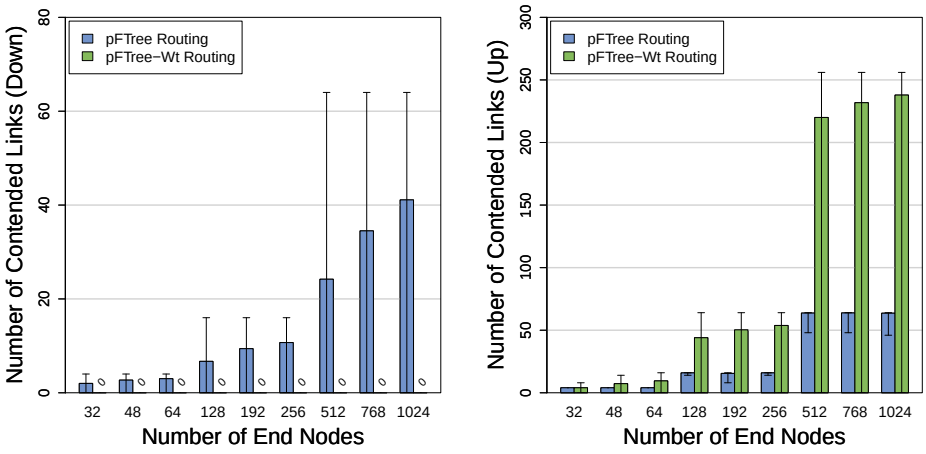
(b) Up Links

Figure 13: Total Contention

routing nodes connected to the leaf-switch *L1*, two up-going ports are available connected to the switches *R1* and *R2*, respectively. As the node a has a weight equal to 100, it is assigned one of those links, $R1 \rightarrow L1$, while the other three nodes share the other link, $R2 \rightarrow L1$. This is because the sum of the weights of the other three nodes is only 3, which is lower than 100. Even though the selected switches are marked with the partition keys, still the partitions can not be isolated in the subnet due to the weighted partition-aware routing. However, when routing nodes connected to the leaf-switch *L2*, as shown in Figure 12(b), where all nodes have equal weights, the partitions are isolated. The nodes g and h, belonging to the same partition, are routed through the link $R1 \rightarrow L1$, while e and f of the second partition are routed through $R2 \rightarrow L1$, in the downward direction. The pFTree-Wt satisfies the weighted load balancing on the links, while keeping the partitions as isolated as possible. Note that the final routing has only one link shared by the nodes of the two partitions.

7.1. Evaluation

In order to evaluate the pFTree-Wt routing algorithm, we use the topologies presented in Table 3. Again, the number of *victim nodes* refer to the nodes we assign to the victim partition, and the rest of the nodes are put in the interfering partition. We generate routing tables for both the original pFTree and the pFTree-Wt routing algorithms with different numbers of receiver nodes (nodes with $weight = 100$) in the victim partition. Both the victim nodes and receiver nodes in the victim partition are chosen randomly for each test. More specifically, with each of our test topologies, we perform multiple experiments, each with a different number of receiver nodes in the partition. The number of receiver nodes is chosen uniformly per switch, and ranges from one receiver node to a case where all the nodes in the victim partition are designated as receivers. We then analyze the generated routing tables for contention, and compare the two routing algorithms. Here, we define that a link is contended if routes towards more than one receiver node pass through it in one direction. If R receiver node flows share a link, we set contention on the link to $R - 1$. Total contention in the network is defined as the sum of the contention at all links. We note total contention, and the number of contended links in both up and down directions in the fat-tree topology.



(a) Down Links

(b) Up Links

Figure 14: Number of Contended Links

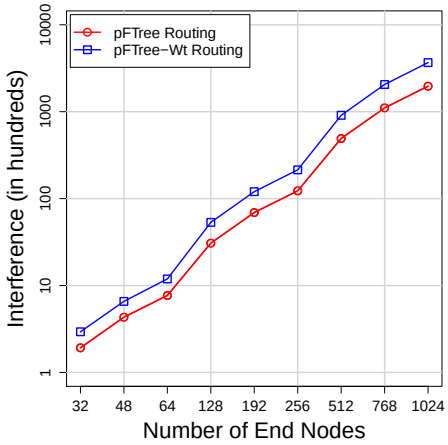


Figure 15: Inter-partition Interference, pFTree and pFTree-Wt.

Figure 13 shows the total contention for the original pFTree and pFTree-Wt routing. The error bars on the plots show the minimum and maximum values observed in all experiments. As shown in Figure 13(a), the pFTree-Wt completely removes the contention in the downward direction, by selecting separate links to route receiver nodes. The original pFTree routing, however, induces contention which on average ranges from 2 (for 32 nodes) to 8.6 (for 1024 nodes) on our test topologies. Furthermore, the maximum contention observed goes up to 192 for our largest topology. Similarly, the average contention for the pFTree routing is also higher than the average contention for pFTree-Wt, as shown in Figure 13(b). For our last two topologies with 768 and 1024 nodes, the average contention for the pFTree routing is 10.7% and 14.64% greater than that of the pFTree-Wt routing, respectively.

The number of contended links noted for the routing algorithms are given in Figure 14. Again, for pFTree-Wt routing, no link is contended in the downward direction, as shown in Figure 14(a). On the contrary, for the pFTree routing algorithm, the number of contended links goes up to 41.1 on average for our largest topology (with max = 64). On the upward links, as shown in Figure 14(b), more contended links are observed for the pFTree-Wt, as compared to the pFTree routing. This is because pFTree-Wt distributes

contention in the network more evenly in the network, with less contention on each link. Hence, congestion is less likely to occur for the receiver nodes in pFTree-Wt routing.

As explained in the start of this section, the pFTree-Wt routing algorithm has to compromise on the partition isolation for better load-balancing in the network, in the presence of nodes with distinct traffic characteristics. We also calculate the total average inter-partition interference for both the pFTree and the pFTree-Wt routing algorithms. The total inter-partition interference represents the number of occurrences when a traffic flow between any source-destination pair, belonging to one partition, shares a link with any other flow of a different partition. The results are given in Figure 15. Note that the total interference includes links in both up and down directions. For the test topology with 32 nodes, the total interference for the pFTree routing is 192, while for the pFTree-Wt routing it is 293 (an increase of 52%). Similarly, for the topology with 1024 nodes, the total interference for pFTree-Wt routing is increased by around 86.76%, as compared to the original pFTree routing. This indicates that for larger topologies, the trade-off between perfect load-balancing and network isolation could be even more significant in pFTree-Wt routing.

8. Analysis of the Proposed Extensions and Future Directions

The two proposed extension to the pFTree routing algorithm presented in this paper, the pFTree-Ext and the pFTree-Wt, targets two distinct use cases. The pFTree-Ext routing is suitable for subnets where tenant groups have different QoS or security requirements, whereas the pFTree-Wt routing targets networks where nodes, irrespective of their partitioning, exhibit distinct traffic characteristics. As a consequence, the pFTree-Ext tends to satisfy the partition-wise isolation policies at the cost of load-balancing. Conversely, the pFTree-Wt routing compromises on partition isolation to keep the load on the links balanced, based on nodes' weight profiles. The two algorithms, in combination, may yield contradictory routing decisions.

A potential combined routing algorithm may use a network objective function to unify the pFTree-Ext and the pFTree-Wt routing algorithms. A very simple objective function is given in Equation 2, where A represents the fractional weight of the pFTree-Ext routing over the pFTree-Wt routing. If $A = 1$, the routing will be solely based on the pFTree-Ext algorithm, whereas for $A = 0.5$, both routing algorithms will be given equal consideration in the final routing of the network. However, for such a routing scheme to work properly, the objective function should be used to decide the port selection for all individual end nodes, in accordance with the overall routing strategy.

$$f(x) = A \times f_{pFTreeExt}(x) + (1 - A) \times f_{pFTreeWt}(x) \tag{2}$$

8.1. Network Reconfiguration

The routing time for both the pFTree-Ext and the pFTree-Wt routing algorithms depends on the number of partitions and the node distribution in the subnet. Furthermore, if the partitioning information in the network is altered due to changes in the tenant information or new allocations of end nodes to the tenants, LFTs need to be updated. In a simple scheme, to avoid reconfiguration cost, new route calculations can be postponed until they are induced by an external factor like a topology change. As both algorithms supports full all-to-all connectivity, the network will continue to work during this period, albeit with degraded performance or with unsatisfied isolation policies. A more efficient solution however, is to update the routing information in real-time, reflecting the current tenant information and end node distribution. As a complete routing calculation is an expensive operation, to save reconfiguration time, the routing can be divided into two distinct phases: calculation of paths, and assignment of the calculated paths to the actual destination nodes based on the partitioning information. The calculation of paths can be done once for a given topology, and a new assignments of paths can be performed as soon as a new reconfiguration is induced. We plan to present such a generalized routing and network reconfiguration scheme in future work.

8.2. Future Directions

Several research directions can be identified as future work to this paper. This includes further enriching global and partition-wise policy parameters. The global policy should be able to limit the percentage of link

resources allocated to the physically-isolated partitions, leaving the rest of the links explicitly for the shared partitions. Furthermore, the notion of *partition groups* can be added to the algorithm, prescribing which specific partitions are allowed to share links in the subnet. Also, as the IB architecture supports adding a port to multiple partitions, the algorithm can be extended to use multi-path routing where different partitions for a single node can be mapped to different routes, and have different isolation characteristics. Finally, the *weights* can be assigned using both node and partition as a basis, so that a node may have different weights for its different partitions. This separation can help the routing function to optimize based on the node weights among the links selected for a specific partition.

9. Related Work

Network and performance isolation is a much discussed topic in the literature, particularly in the context of Ethernet based data centers. Both hypervisor level rate-limits and QoS features have been used to provide appropriate bandwidth to the tenants. SeaWall [40] provides a fair network sharing policy among competing virtual machines (VMs). However, as the sharing policy applies to the VMs instead of tenants, a tenant can practically increase its share of the bandwidth by launching additional source VMs. Other solutions, like Netshare [41], Oktopus [42], and SecondNet [43] work on per tenant bandwidth share basis, but require some kind of centralized control plane resulting in reaction time overhead. A more recent approach, EyeQ [44] uses congestion control to provide predictable bandwidth guarantees to the tenant VMs. The isolation system works by enforcing admission control on traffic, thus pushing bandwidth contention to the network edge. However, unlike our work, the Ethernet based solutions does not separate network links for the tenants' nodes physically using routing. Hence, the intermediate links are still *shared* by the flows belonging to different tenant clusters. In addition, such solutions provide poor load-balancing of the available network links.

In the context of IB based interconnection networks, cloud computing has recently gained attention due to an increased interest in on-demand HPC provisioning. Several approaches have been proposed to build HPC clouds. In [3], the requirements of a high-performance cloud computing infrastructure based on IB interconnect technology are discussed, and a complete model of IB based clouds is presented. Recently, an extension to the OpenStack [45] cloud orchestration platform has been proposed, where the *single root I/O virtualization* (SR-IOV) technique is used to provide efficient virtualization [6]. Similarly, to minimize the virtualization overhead, a software-defined networking approach is presented in [5]. These approaches, however, do not target the tenant performance isolation in a multi-tenant cloud environment. Similarly, a number of IB supported topology-agnostic routing algorithms [27, 28] use VLs to achieve deadlock freedom, without differentiating between nodes belonging to different tenants [46, 47].

IB provides QoS features that could be used to guarantee each partition its assigned share of bandwidth, regardless of the nodes in the other partitions. The bandwidth guarantees are then provided by assigning each partition an available SL. Each SL is then mapped to one of the available fifteen VLs[4] of the link according to the SL-to-VL mapping table [48]. The problem with assigning SLs to the partitions is that we can only use 15 VLs to create distinct partitions in the network, while an IB network in general can have any number of partitions[5]. Furthermore, it is common to support only nine VLs (including the one reserved for subnet management) in existing IB hardware. Moreover, as SLs are a scarce resource, it may be desirable to leave as many of them as possible free for other purposes, e.g. to provide fault-tolerance or service differentiation in the network [49]. The HPC virtualized cloud [4] describes a method to isolate virtual clusters belonging to different tenants using partitions in the IB networks. The authors have added support for per virtual machine (VM) partition mapping to the IB architecture. The proposed solution effectively enables a physical machine to host VMs belonging to different partitions. Originally, IB provides partitions on per port basis and VMs sharing a physical port cannot be associated with different partitions.

[4]The last VL, VL_{15}, is reserved for subnet management.

[5]At ports, each partition is identified by a 16-bit *P_Key* value. The most significant bit is used to store membership type information. Hence, the rest of the 15-bits are available to store the partition identifier. That is, each port can be a member of up to 32768 partitions.

For network isolation, SLs are used. Again, the use of SLs (and VLs) limits the number of partitions possible in the cloud.

The fat-tree is a widely used topology in data center networks. Several proposals have been presented to improve system utilization by intelligent job allocation and scheduling in fat-tree topologies. In [50], the authors have presented an allocation algorithm for achieving high network utilization and application isolation in fat-trees. The algorithm assumes centralized knowledge of the complete data center wide workload, and allocates processing nodes on a per job basis. This requirement is in contrast to the practical server allocations to the tenants in HPC clouds. In addition, the proposed algorithm was not implemented in a real-world system. A recent work [51] also targets job performance predictability in HPC systems by creating virtual network blocks depending on the expected workload distribution. Again, the system targets typical HPC systems and is not generally implementable in HPC clouds. In [18], we presented a partition-aware routing algorithm for the fat-tree topologies, which utilizes both physical-level and VL-based mechanisms to provide network-wide isolation of partitions belonging to different tenant groups. Another closely related work is the more recent Link-as-a-Service (LaaS) proposal [52] for HPC clouds. The solution works only on physical link isolation without virtual channels, hence imposing stricter conditions for tenant admission. In addition, the LaaS system requires an additional OpenStack-based link allocation service, and does not use the readily available partitioning feature of the IB interconnection network. Furthermore, no support for tenant-wise isolation policies differentiating tenants with distinct SLAs is provided.

10. Conclusion

In this paper, to improve network isolation and load-balancing in HPC cloud systems, we presented two significant extensions to our previously proposed partition-aware fat-tree routing algorithm. First, we proposed pFTree-Ext, an extended pFTree routing algorithm that incorporates provider defined partition-wise policies governing the sharing of network resources among partitions. Second, we presented a weighted version of the pFTree routing, pFTree-Wt, that considers node traffic characteristics to balance load across the network links more evenly. Our experiments and simulations verify the feasibility and effectiveness of the proposed extensions. In particular, the pFTree-Ext routing is able to completely remove inter-partition interference for selected physically-isolated partition. Similarly, with *weighted* nodes in the network, the pFTree-Wt completely removes contention in the downward direction while reducing it up to 14.6% in the upward direction, as compared to the original pFTree routing algorithm.

Acknowledgement

This work was supported by the Norwegian research council under the ERAC project (Project number: 213283/O70). The authors would also like to thank Mellanox Technologies for providing some of the hardware we use in our experiments.

References

[1] InfiniBand Architecture Specification: Release 1.3, http://www.infinibandta.com/ (2015).

[2] Top 500 Super Computer Sites, accessed February 1, 2016.
URL http://www.top500.org/

[3] V. Mauch, M. Kunze, M. Hillenbrand, High performance cloud computing, Future Generation Computer Systems 29 (6) (2013) 1408–1416. doi:10.1016/j.future.2012.03.011.

[4] M. Hillenbrand, V. Mauch, J. Stoess, K. Miller, F. Bellosa, Virtual InfiniBand clusters for HPC clouds, in: Proceedings of the 2nd International Workshop on Cloud Computing Platforms, ACM, 2012, p. 9. doi:10.1145/2168697.2168706.

[5] P. Rad, R. V. Boppana, P. Lama, G. Berman, M. Jamshidi, Low-latency software defined network for high performance clouds, in: 10th System of Systems Engineering Conference (SoSE), 2015, IEEE, 2015, pp. 486–491. doi:10.1109/SYSOSE.2015.7151909.

[6] J. Zhang, X. Lu, M. Arnold, D. K. Panda, MVAPICH2 over OpenStack with SR-IOV: An Efficient Approach to Build HPC Clouds, in: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015., IEEE, 2015, pp. 71–80. doi:10.1109/CCGrid.2015.166.

[7] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, Journal of internet services and applications 1 (1) (2010) 7–18. doi:10.1007/s13174-010-0007-6.

[8] H. Takabi, J. B. Joshi, G.-J. Ahn, Security and Privacy Challenges in Cloud Computing Environments., IEEE Security & Privacy 8 (6) (2010) 24–31. doi:10.1109/MSP.2010.186.

[9] K. Ren, C. Wang, Q. Wang, Security challenges for the public cloud, IEEE Internet Computing (1) (2012) 69–73. doi:10.1109/MIC.2012.14.

[10] T. Dillon, C. Wu, E. Chang, Cloud computing: issues and challenges, in: 24th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2010., Ieee, 2010, pp. 27–33. doi:10.1109/AINA.2010.187.

[11] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, B. Gao, A framework for native multi-tenancy application development and management, in: The 9th IEEE International Conference on E-Commerce Technology, 2007., IEEE, 2007, pp. 551–558. doi:10.1109/CEC-EEE.2007.

[12] A. Gupta, D. Milojicic, Evaluation of HPC Applications on Cloud, in: 6th Open Cirrus Summit (OCS), 2011., 2011. doi:10.1109/OCS.2011.10.

[13] P. Bientinesi, R. Iakymchuk, J. Napper, HPC on competitive cloud resources, in: Handbook of Cloud Computing, Springer, 2010, pp. 493–516.

[14] A. Iosup, N. Yigitbasi, D. Epema, On the performance variability of production cloud services, in: 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2011., IEEE, 2011, pp. 104–113. doi:10.1109/CCGrid.2011.22.

[15] Amazon AWS High Performance Computing, accessed February 1, 2016.
URL https://aws.amazon.com/hpc/

[16] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, B. Saha, Sharing the Data Center Network.

[17] X. Yuan, W. Nienaber, Z. Duan, R. Melhem, Oblivious routing for fat-tree based system area networks with uncertain traffic demands, in: ACM SIGMETRICS Performance Evaluation Review, Vol. 35, ACM, 2007, pp. 337–348. doi:10.1109/TNET.2009.2012853.

[18] F. Zahid, E. G. Gran, B. Bogdański, B. D. Johnsen, T. Skeie, Partition-Aware Routing to Improve Network Isolation in InfiniBand Based Multi-tenant Clusters, in: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015., 2015, pp. 189–198. doi:10.1109/CCGrid.2015.96.

[19] M. A. Rappa, The utility business model and the future of computing services, IBM Systems Journal 43 (1) (2004) 32.

[20] T. Sterling, D. Stark, A high-performance computing forecast: partly cloudy, Computing in Science & Engineering 11 (4) (2009) 42–49. doi:10.1109/MCSE.2009.111.

[21] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, D. Epema, A performance analysis of EC2 cloud computing services for scientific computing, in: Cloud computing, Springer, 2009, pp. 115–131. doi:10.1007/978-3-642-12636-9_9.

[22] C. E. Leiserson, Fat-trees: universal networks for hardware-efficient supercomputing, IEEE Transactions on Computers 100 (10) (1985) 892–901. doi:10.1109/TC.1985.6312192.

[23] R. Householder, S. Arnold, R. Green, On cloud-based oversubscription, arXiv preprint arXiv:1402.4758.

[24] A. Greenberg, J. Hamilton, D. A. Maltz, P. Patel, The cost of a cloud: research problems in data center networks, ACM SIGCOMM computer communication review 39 (1) (2008) 68–73. doi:10.1145/1496091.1496103.

[25] A. Bermúdez, R. Casado, F. J. Quiles, T. M. Pinkston, J. Duato, On the infiniband subnet discovery process, in: Proceedings of the IEEE International Conference on Cluster Computing, 2003., IEEE, 2003, pp. 512–517. doi:10.1109/CLUSTR.2003.1253361.

[26] E. Zahavi, G. Johnson, D. J. Kerbyson, M. Lang, Optimized InfiniBand fat-tree routing for shift all-to-all communication patterns, Concurrency and Computation: Practice and Experience 22 (2) (2010) 217–231. doi:10.1002/cpe.1527.

[27] T. Skeie, O. Lysne, I. Theiss, Layered Shortest Path (LASH) Routing in Irregular System Area Networks., in: International Parallel and Distributed Processing Symposium (ipdps), 2002., Vol. 2, Citeseer, 2002, p. 194. doi:10.1109/IPDPS.2002.1016559.

[28] J. Domke, T. Hoefler, W. E. Nagel, Deadlock-free oblivious routing for arbitrary topologies, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2011., IEEE, 2011, pp. 616–627. doi:10.1109/IPDPS.2011.65.

[29] IEEE, 802.1 Q/D10, IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks. (1997).

[30] F. Petrini, M. Vanneschi, k-ary n-trees: High performance networks for massively parallel architectures, in: Proceedings of the 11th International Parallel Processing Symposium, 1997., IEEE, 1997, pp. 87–93. doi:10.1109/IPPS.1997.580853.

[31] S. R. Öhring, M. Ibel, S. K. Das, M. J. Kumar, On generalized fat trees, in: Proceedings of the 9th International Parallel Processing Symposium, 1995., IEEE, 1995, pp. 37–44. doi:10.1109/IPPS.1995.395911.

[32] E. Zahavi, D-Mod-K routing providing non-blocking traffic for shift permutations on real life fat trees, CCIT Report 776, Technion.

[33] E. Zahavi, Fat-tree routing and node ordering providing contention free traffic for MPI global collectives, Journal of Parallel and Distributed Computing 72 (11) (2012) 1423–1432. doi:10.1016/j.jpdc.2012.01.018.

[34] F. Zahid, E. G. Gran, B. Bogdański, B. D. Johnsen, T. Skeie, A Weighted Fat-Tree Routing Algorithm for Efficient Load-Balancing in InfiniBand Enterprise Clusters, in: 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2015., 2015, pp. 35–42. doi:10.1109/PDP.2015.111.

[35] NAS Parallel Benchmarks, accessed February 1, 2016.
URL https://www.nas.nasa.gov/publications/npb.html/

[36] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al., The NAS parallel benchmarks, International Journal of High Performance Computing Applications 5 (3) (1991) 63–73. doi:10.1177/109434209100500306.

[37] T. Schneider, T. Hoefler, A. Lumsdaine, ORCS: An oblivious routing congestion simulator, Indiana University, Computer Science Department, Tech. Rep.

[38] T. Hoefler, T. Schneider, A. Lumsdaine, The effect of network noise on large-scale collective communications, Parallel Processing Letters 19 (04) (2009) 573–593. doi:10.1142/S0129626409000420.

[39] T. Hoefler, T. Schneider, and A. Lumsdaine, Multistage switches are not crossbars: Effects of static routing in high-performance networks, in: IEEE International Conference on Cluster Computing, 2008., IEEE, 2008, pp. 116–125. doi:10.1109/CLUSTR.2008.4663762.

[40] A. Shieh, S. Kandula, A. Greenberg, C. Kim, Seawall: performance isolation for cloud datacenter networks, in: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, USENIX Association, 2010, pp. 1–1.

[41] S. Radhakrishnan, R. Pan, A. Vahdat, G. Varghese, et al., Netshare and stochastic netshare: predictable bandwidth allocation for data centers, ACM SIGCOMM Computer Communication Review 42 (3) (2012) 5–11. doi:10.1145/2317307.2317309.

[42] H. Ballani, P. Costa, T. Karagiannis, A. Rowstron, Towards predictable datacenter networks, in: ACM SIGCOMM Computer Communication Review, Vol. 41, ACM, 2011, pp. 242–253. doi:10.1145/2043164.2018465.

[43] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, Y. Zhang, Secondnet: a data center network virtualization architecture with bandwidth guarantees, in: Proceedings of the 6th International COnference (Co-NEXT), 2010., ACM, 2010, p. 15. doi:10.1145/1921168.1921188.

[44] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, A. Greenberg, EyeQ: Practical Network Performance Isolation at the Edge, in: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, 2013, pp. 297–312.

[45] OpenStack Cloud Computing Software, accessed February 1, 2016.
URL https://www.openstack.org/

[46] W. L. Guay, B. Bogdanski, S.-A. Reinemo, O. Lysne, T. Skeie, vFtree- A fat-tree routing algorithm using virtual lanes to alleviate congestion, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2011., IEEE, 2011, pp. 197–208. doi:10.1109/IPDPS.2011.28.

[47] 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, ACM, 2011, pp. 1–10. doi:10.1145/2110217.2110219.

[48] F. J. Alfaro, J. L. Sánchez, J. Duato, QoS in InfiniBand subnetworks, IEEE Transactions on Parallel and Distributed Systems 15 (9) (2004) 810–823. doi:10.1109/TPDS.2004.46.

[49] J. C. Sancho, A. Robles, P. Lopez, J. Flich, J. Duato, Routing in infiniBand^{TM} torus network topologies, in: Proceedings of the International Conference on Parallel Processing, 2003., IEEE, 2003, pp. 509–518. doi:10.1109/ICPP.2003.1240618.

[50] F. O. Sem-Jacobsen, Å. G. Solheim, O. Lysne, T. Skeie, T. Sødring, Efficient and contention-free virtualisation of fat-trees, in: IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011., IEEE, 2011, pp. 754–760. doi:10.1109/IPDPS.2011.218.

[51] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, J. Labarta, Quiet neighborhoods: Key to Protect Job Performance Predictability, in: IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2015, IEEE, 2015, pp. 449–459. doi:10.1109/IPDPS.2015.87.

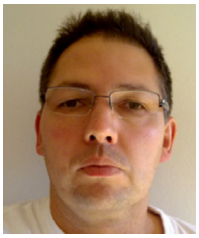
[52] E. Zahavi, A. Shpiner, O. Rottenstreich, A. Kolodny, I. Keslassy, Links as a Service (LAAS): Feeling alone in the shared cloud, CCIT Report 888 September 2015, Technion.

Author Biographies

Feroz Zahid is a doctoral candidate at the University of Oslo. He is also affiliated with Simula Research Laboratory, and working on the RCN funded research project ERAC; Efficient and Robust Architecture for the Big Data cloud. His research interests include Interconnection Networks, Distributed Systems, Cloud Computing, Energy Efficient Systems, Network Security, Machine Learning and Big Data. He has an M.S.(Computer Science) degree from the National University of Computer and Emerging Sciences and a B.S.(Computer Science) degree from the University of Karachi.

Ernst Gunnar Gran received the candidatus scientiarum degree (MS) and Ph.D. degrees in computer science from the Department of Informatics at the University of Oslo in 2007 and 2014, respectively. His research focuses on congestion management in lossless interconnection networks in general, and congestion control as specified for InfiniBand in particular. Prior to his Ph.D., he has worked full time for several years as a system administrator and scientific programmer, first at the Department of Informatics at the University of Oslo, and later at the Simula Research Laboratory. He has also been incharge of designing, implementing, and deploying the multihomed IP based research testbed NorNet Core, and he is currently heading the cloud department at Simula, and leading the RCN funded research project ERAC; Efficient and Robust Architecture for the Big Data cloud.

Bartosz Bogdański received his M.S. degree in electrical engineering from Technical University of Lodz in Poland in 2009 and his Ph.D. degree in computer science from the University of Oslo, Norway in 2014. He is currently working as a Senior Software Engineer at Oracle Corporation. He has co-authored several scientific papers and is currently the holder of many patents in several areas including interconnection networks, clusters and high availability systems. His research interests include routing protocol development, performance evaluation and Exadata/Exalogic systems.



Bjørn Dag Johnsen received his M.S. degree in Computer Science from the University of Bergen in 1989. He is currently a Senior Principle Software Engineer at Oracle and have been a software and systems architect and developer in Dolphin Server Technology, Dolphin Interconnect Solutions and Sun Microsystems before joining Oracle as part of the Oracle acquisition of Sun Microsystems in 2010. He has co-authored several scientific papers and is currently the holder of more than thirty patents in several areas including interconnect fabric, IO virtualization, clusters and high availability systems.

Tor Skeie received this M.S. and Ph.D. degrees in Computer Science from the University of Oslo in 1993 and 1998, respectively. He is a professor at the Simula Research Laboratory and the University of Oslo. His work is mainly focused on scalability, effective routing, fault tolerance, and quality of service in switched networked topologies. He is also a researcher in the Industrial Ethernet area. The key topics here have been the road to deterministic Ethernet end-to-end and how precise time synchronization can be achieved across switched Ethernet. He has also contributed to wireless networking, hereunder quality of service in WLAN and cognitive radio.