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

Wei Lin Guay*, Sven-Arne Reinemo*, Olav Lysne*, Tor Skeie*†
*Simula Research Laboratory, Norway
†Department of Informatics, University of Oslo
{weilin, svenar, olavly, tskeie}@simula.no

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

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

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

I. Introduction

For fat-trees, as with most other topologies, the routing algorithm is crucial for efficient use of the underlying topology. The popularity of fat-trees in the last decade has led to many efforts trying to improve the routing performance in fat-trees. This includes the current approach that the OpenFabrics Enterprise Distribution [1], the de facto standard for InfiniBand (IB) system software, is based on [2], [3]. These proposals, however, have several limitations when it comes to flexibility and scalability. One problem is the static routing used by IB technology that limits the exploitation of the path diversity in fat-trees as pointed out by Hoefler et al. in [4]. Another problem with the current routing is its shortcomings when routing oversubscribed fat-trees as addressed by Rodriguez et al. in [5]. A third problem is that performance is reduced when the number of compute nodes connected to the tree is reduced as addressed by Bogdanski et al. in [6]. And finally we have the problem of reducing the negative impact of congestion due to head-of-line (HOL) blocking [7]. This is not a routing problem per se as this should be handled by a congestion control mechanism, e.g. the mechanism found in IB [8], [9]. This mechanism, however, has its own set of challenges; one being that it is not supported by all IB hardware, another being that it is not yet understood how to configure congestion control for large networks [10]. Therefore, it is important to minimize the problem by other means. A recent proposal by Rodriguez et al. [11] addresses the congestion issue from a routing perspective, but in an application-specific manner and without using virtual lanes (VLs). Another approach using a combination of multipath routing and bandwidth estimation was proposed by Vishnu et al. in [12], but this is significantly more complex to implement than our proposal. A third proposal by Escudero-Sahuquillo et al. [13] uses multiple queues at the input ports in the switches to avoid HOL blocking, but this is not compatible with any existing network technology and requires new hardware to be built. In [14] we suggested the vFtree algorithm that uses a combination of efficient routing and virtual lanes to alleviate congestion. A problem with this approach, however, is that it is based on a static distribution of source-destination pairs across a set of VLs. The static behaviour of the vFtree algorithm limits the performance whenever there is a mismatch between the current hot-spot and the precalculated distribution of source-destination pairs across VLs.

To rectify this we now propose the dFtree algorithm where the allocation of VLs is performed dynamically during network operation using an optimisation feedback cycle (Fig. 1). We introduce a *performance manager* [8] that monitors the network using hardware port counters to detect congestion and optimises the current VL allocation by classifying flows as either *slow lane* (contributors to congestion) or *fast lane* (victims of congestion). Then the optimisation is applied using the host side dynamic reconfigration method we proposed in [15]. The effect being that all flows contributing to congestion are migrated to a separate VL (slow lane) in order to avoid the negative impact of head-of-line blocking on the flows not contributing to congestion (victim flows). Compared to the vFtree approach we avoid the bottleneck of static allocation of VLs and we reduce the number of VLs required to two. Compared to normal IB congestion control [10] we remove the need for source throttling of the contributors. Furthermore, the current available IB congestion control (CC) parameters cause oscillations among all the flows because IB CC is dynamically adjusting the injection rate of the senders. As a result, this solution might not be suitable for congestion problem of a more persistent nature because the oscillations that can reduce the overall network throughput. In our previous work [15], we are able to obtain a better overall network throughput in certain congestion scenarios by avoiding the oscillations. Such persistent congestion problems occur when traffic has been moved away from a failed link, when multiple jobs run on the same system, and compete for network resources, or when a system is not balanced for the application that runs on it. Our approach handles persistent congestion problems by first detecting them, and thereafter dynamically redistributing the VL resources so as to obtain a balance that will be impossible to achieve statically at system start-up.

The rest of this paper is organized as follows: we introduce the InfiniBand Architecture in Section II followed by the dFtree design and implementation in Section III. Then we describe the experimental setup in Section IV followed by the performance analysis in Section V. Finally, we conclude in Section VI.

II. The InfiniBand Architecture

InfiniBand is a serial point-to-point full-duplex technology, that was first standardized in October 2000 [8]. The current trend is that IB is replacing proprietary or low-performance solutions in the high performance computing domain [16], where high bandwidth and low latency are the key requirements.

The de facto system software for IB is Open Fabrics Enterprise Distribution(OFED) developed by dedicated professionals and maintained by the OpenFabrics Alliance [1]. OFED is open source and is available for both GNU/Linux and Microsoft Windows. The dFtree algorithm that we propose in this paper was implemented and evaluated in a development version of OpenSM, which is the subnet manager (SM) distributed together with OFED.

A. The Subnet Manager

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

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

B. The Performance Manager

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

The task of the *performance manager* (PM) [8] is to retrieve performance and error-related information from these registers. The information is retrieved by issuing a performance management datagram (MAD) to the PMA of a given device. The PMA then executes the retrieval and returns the result to the PM. As a result, the PM can use this information to detect incipient failures and based on this information, the PM can advise the SM about recommended or required path changes and performance optimisations.

III. The dFtree Design and Implementation

In this section, we present the design and implementation of the dFtree algorithm. The algorithm is generic and can be applied to any topology and routing algorithm, but in this paper we focus on fat-trees because of the simplicity they provide with respect to freedom from deadlock.

A. Overview

Performance tuning is the main activity associated with performance management where tuning consists of finding and eliminating bottlenecks. Hence, we are using the PM as one of the key components to enable dynamic allocation of virtual lanes to alleviate network congestion. Fig. 1 summarises the optimisation feedback cycle that consists of the SM, the PM and our modified host stack with the host side dynamic reconfiguration capability [15], [17]. In a subnet, the SM periodically sweeps the subnet to discover changes and maintain a fully connected subnet. Similarly, the PM periodically collects information from every component in the subnet in order to analyse the network performance. After the analysis, the PM forwards the relevant information to our modified host stack that reconfigures the virtual lanes in order to improve network performance.

B. Design

Head-of-Line blocking during traffic peaks or ”hot-spot” traffic patterns is one reason for performance degradation

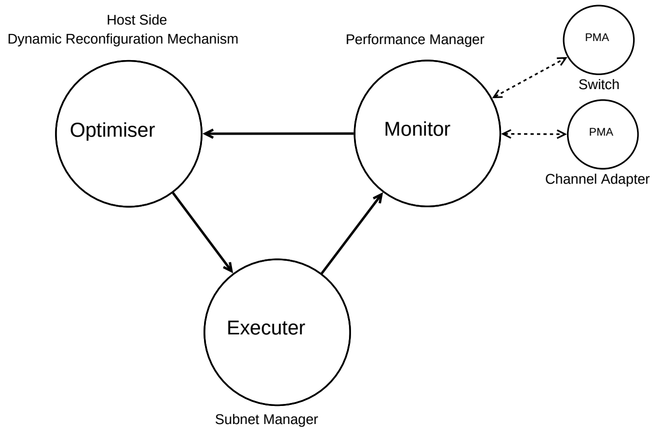


Figure 1. The performance optimisation feedback cycle.

in interconnection networks [7]. Common sources for hot-spots include complex traffic patterns due to virtualisation, migration of virtual machine images, checkpoint and restore mechanisms for fault tolerance, and storage and I/O traffic.

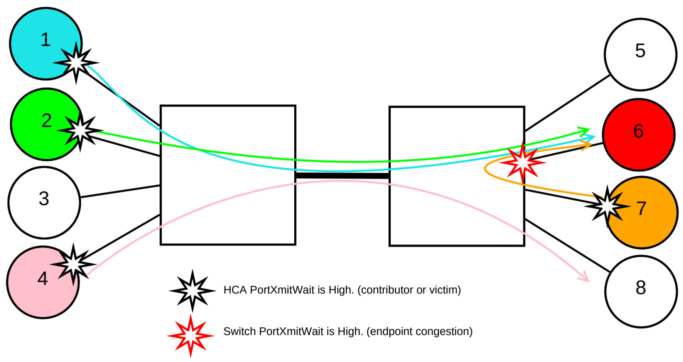
One way to avoid this problem is to use a congestion control (CC) mechanism such as the one recently specified and implemented in IB. This mechanism was evaluated in hardware and shown to be working by Gran et al. in [10], however, IB CC is not always available, e.g due to a mixture of old and new equipments in large clusters and this is often the case that will exist for years.

In [14], we suggested an enhancement to the routing algorithm in OpenSM [3] that utilises multiple virtual lanes to improve performance during the existence of hot-spots. The virtual lanes are assigned statically during the routing table generation and can avoid the negative impact of the victim flows. However, the assumption is that the topology is a balanced, fully populated and fault-free fat-tree. In order to overcome the shortcomings in our previous work [14], we need a mechanism to identify the hot-spot flows and assign the virtual lanes dynamically. Thus, we are using two standard IB performance counters (Table I) as the metrics to identify endpoint hot-spots and their contributors dynamically during network operation. In addition, we have derived three equations to calculate the *normalised port congestion* and the *port utilisation* that are based on the above mentioned performance counters.

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

Table I
NOTATION

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



1
2
3
4
5
6
7
8
HCA PortXmitWait is High. (contributor or victim)
Switch PortXmitWait is High. (endpoint congestion)

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

has a high value for $Congestion_{port}$.

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

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

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

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

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

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

C. Implementation

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

Algo. 1 is executed after every iteration of the performance sweep. The algorithm checks if the remote switch port of an endnode has a $Congestion_{port}$ value exceeding the *threshold*, if this is true the conclusion is that the endnode is a hot-spot and the remote switch port is marked as a $hotspot_{port}$. After discovering an endpoint hot-spot, Algo. 1 triggers the forwarding a repath to all potential contributors. This trap encapsulates the LID, $hotspot_{LID}$, of the congested node.

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

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

When a repath trap is received by a potential contributor, Algo. 2 is executed. The host will retrieve all the active QPs and compare them with the DLID in the repath trap. If a matching DLID is found in one of the QPs, the QP is reconfigured to use a *'slow lane'*. Initially, all QPs are initalised using a *'fast lane'*.

D. Limitations

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

Another concern is that the detection of the hot-spot flows depends on the interval of the performance sweeps. If a hot-spot appeared just after iteration n, the hot-spot detection

Algorithm 1 Detect endpoint hot-spot and its contributors

```
Ensure: Subnet is up and running and PM is constantly sweeping
 1: for $sw_{src} = 0$ to $sw_{max}$ do
 2:   for $port_{sw} = 0$ to $port_{max}$ do
 3:     if $remote\_port(port_{sw})$ == HCA then
 4:       if $congestion_{port} >$ Threshold then
 5:         if $port_{sw} \neq$ hot-spot then
 6:           Mark $port_{sw}$ as $hotspot_{port}$
 7:         end if
 8:         Encapsulate $hotspot_{LID}$in a repath trap
 9:         Encapsulate $slow\ lane$ as $SL_{repath\ trap}$
10:         for $hca_{src} = 0$ to $hca_{max}$ do
11:           if $congestion_{port} >$ Threshold then
12:             if $hca \neq hotspot_{LID}$ contributor then
13:               if $Utilisation_{port} <$ 0.5 then
14:                 Mark $hca$ as $hotspot_{LID}$ contributor
15:                 Forward repath trap to HCA
16:               end if
17:             end if
18:           end if
19:         end for
20:       else if $congestion_{port} <$ Threshold then
21:         if $port_{sw}$ == hot-spot then
22:           Clear $port_{sw}$ as $hotspot_{port}$
23:           Encapsulate $hotspot_{LID}$ in a unpath trap
24:           Encapsulate $fast\ lane$ as $SL_{repath\ trap}$
25:           for $hca_{src} = 0$ to $hca_{max}$ do
26:             if $hca$ is $hotspot_{LID}$ contributor then
27:               Clear $hca$ as $hotspot_{LID}$
28:               Forward unpath trap to HCA
29:             end if
30:           end for
31:         end if
32:       end if
33:     end if
34:   end for
35: end for
```

Algorithm 2 Reconfigure QP to slow/fast lane

```
Ensure: Host receives repath trap
1: for $QP_i = 0$ to $QP_{max}$ do
2:   if $DLID_{QP} == DLID_{repath\ trap}$ then
3:     Reconfigure $SL_{QP}$ according to $SL_{repath\ trap}$
4:   end if
5: end for
```

and the *'slow lane'* assignment can only be performed at iteration $n+1$, i.e. t seconds later.

IV. Experiment Setup

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

A. Experimental Test Bed

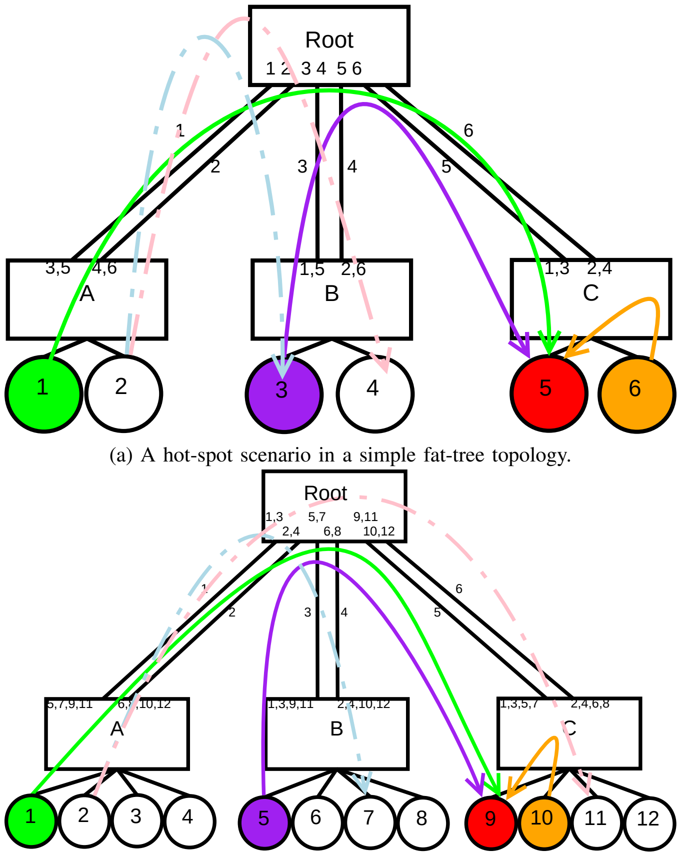
Our test bed consists of twelve nodes and four switches. Each node is a Sun Fire X2200 M2 server that has a dual port Mellanox ConnectX DDR HCA with an 8x PCIe 1.1 interface, one dual core AMD Opteron 2210 CPU, and 2GB of RAM. The switches are one 24-port Infiniscale-III DDR switches and three 36-port Infiniscale-IV QDR switches which we used to construct the topologies illustrated in Fig. 3. All the hosts have CentOS 5.3 installed with a customised IB host stack and the subnet is managed by a modified version of OpenSM 3.3.5 that includes the PM. The *Perftest* [18] tool was also modified to support regular bandwidth reporting and continuous sending of traffic at full link capacity. The modified *Perftest* is used to generate the hot-spots shown in Fig. 3a and Fig. 3b.

B. Simulation Test Bed

To perform large-scale evaluations and verify the scalability of our proposal, we developed an InfiniBand model for the OMNeT++ simulator [19]. The simulations were performed on a 648-port fat-tree topology as shown in Fig. 4 with a nonuniform traffic pattern, where 5% of all packets generated by a compute node was sent to a predefined hot-spot and the rest of the traffic was sent to a randomly chosen node. Additionally, we used multiple localised hot-spots by partitioning the 648-port switched network into one, three or nine segments as described in section V-D.

V. Performance Evaluation

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



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

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

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

A. Synthetic Traffic Patterns - Non-oversubscribed fat-tree

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

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

Fig. 5 shows the per flow throughput with and without the dFtree algorithm. In Fig. 5a, the victim (2-3) is running at 12.9 Gbps before the congested flows are introduced. Starting from 13s, the congestion towards node 5 blocks the traffic on link 1 and 3. Consequently, the bandwidth of flow 2-3 is reduced to 3 Gbps, the same bandwidth that the congested flow 1-5 achieved across link 1 due to

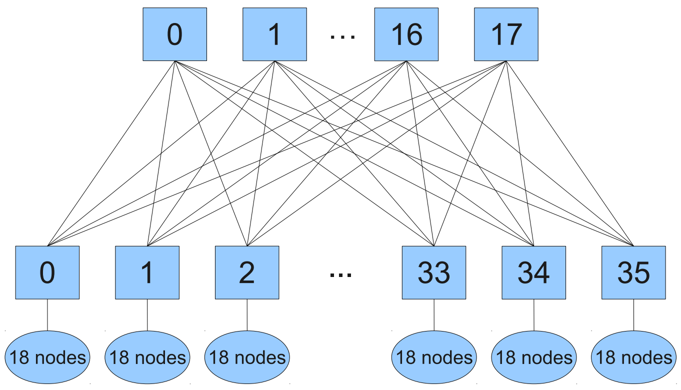


Figure 4. A 648-port switch fat-tree topology.

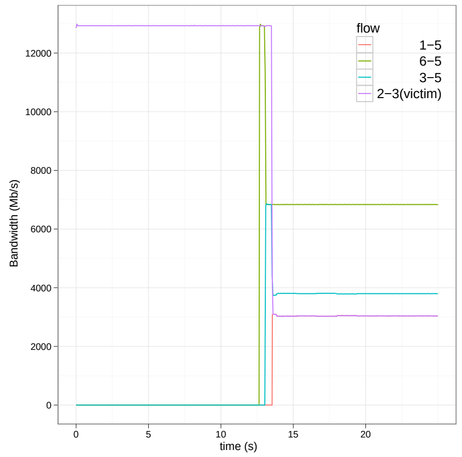
the HOL blocking. Fig. 5b shows the per flow throughput with the dFtree algorithm. Now, the victim flow achieves a throughput of 7.5 Gbps and it is not affected by the congestion. We have also summarised the worst case per flow throughput with and without the dFtree algorithm during the congestion in Fig. 6. With dFtree, the victim flow has improved approximately 150% from 3 Gbps to 7.5 Gbps without impacting the contributors.

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

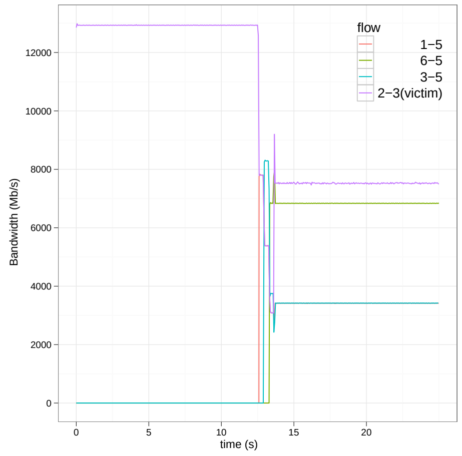
Another observation is that flow 6-5 has a higher share of the bandwidth at 6.8 Gbps toward the hot-spot than flow 1-5 and 3-5 because of the parking lot problem [20]. In order to resolve the parking lot problem, we would need to use additional VLs.

2) Experiment II: In this experiment, the victim flow is now flow 2-4. It has to share the upstream link with flow 1-5, one of the contributors, if a fault happens on link 1 or 2. Thus, in this experiment all flows are started at the same time, but after 13s we disconnect link 2 to emulate a link failure.

Fig. 7 shows the per flow throughput with and without the dFtree algorithm in a faulty link scenario. Fig 7a shows that the victim flow 2-4 achieves its maximum bandwidth at 12.9 Gbps in the presence of congested flows (before



(a) Per flow throughput without dFtree.



(b) Per flow throughput with dFtree.

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

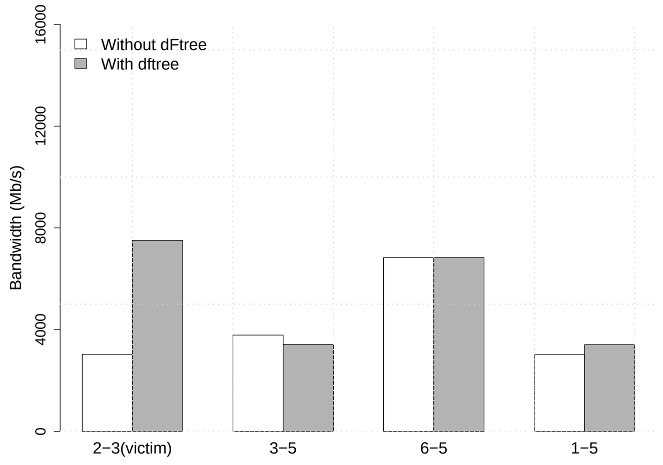


Figure 6. Per flow worst case bandwidth during congestion.

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

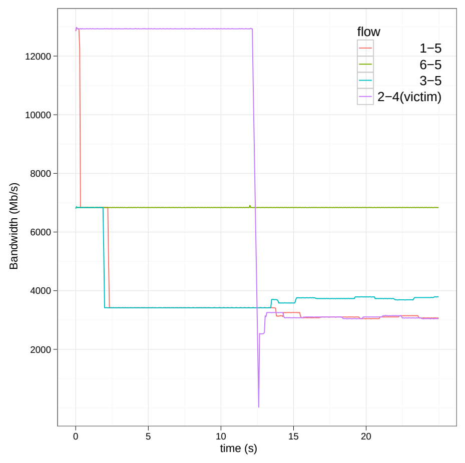
In summary, Fig. 8 shows that the dFtree algorithm achieves approximately a 150% improvement in throughput from 3 Gbps to 7.5 Gbps for the victim flow worst case throughput without affecting the congested flows after the fault happened.

B. *Synthetic Traffic Patterns - 2:1 oversubscribed fat-tree*

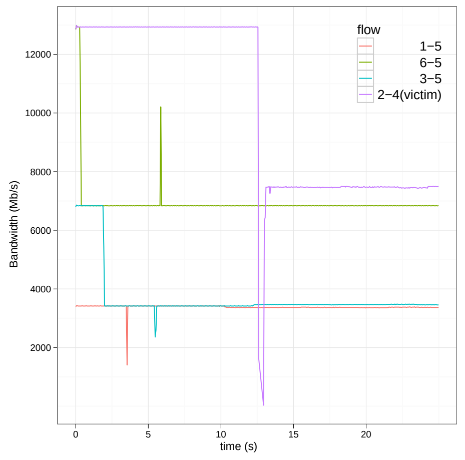
We have also carried out two different experiments on a *2:1 oversubscribed fat-tree* as shown in Fig. 3b. In an *oversubscribed fat-tree*, the downward path is not dedicated to a single destination, but it is shared by several destinations. The term *2:1* means that a downward path is shared by two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

In a 2:1 oversubscribed fat-tree, there are two situations where the victim flows may suffer from HOL blocking because the links are oversubscribed. The first case is similar to section V-A1 where the performance reduction is due to the upstream link being shared with the congestion contributors. The second case is when both the upstream and downstream links are shared as discussed in section V-B2. For both experiments, we use the synthetic traffic flows {1-9, 5-9, 10-9} to evaluate the negative impact of HOL blocking in the *2:1 oversubscribed fat-tree*.The hot-spot is at node 9, and node 1, 5 and 10 are the contributors. The victim flow is started first and the contributors are added after 13s.

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



(a) Per flow throughput without dFtree.



(b) Per flow throughput with dFtree.

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

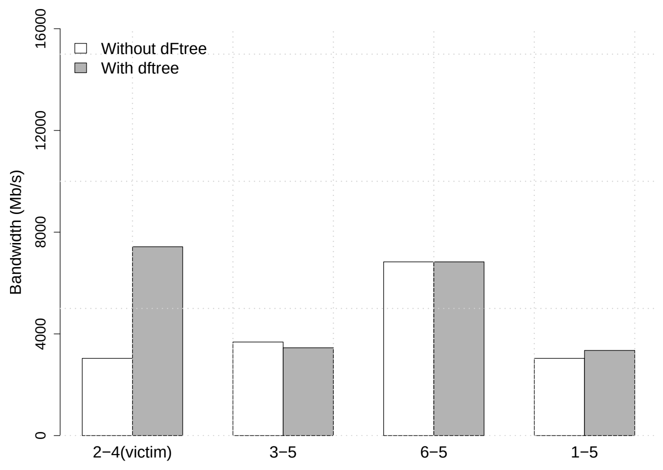
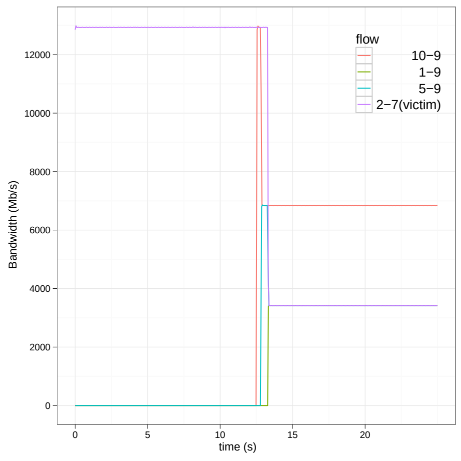
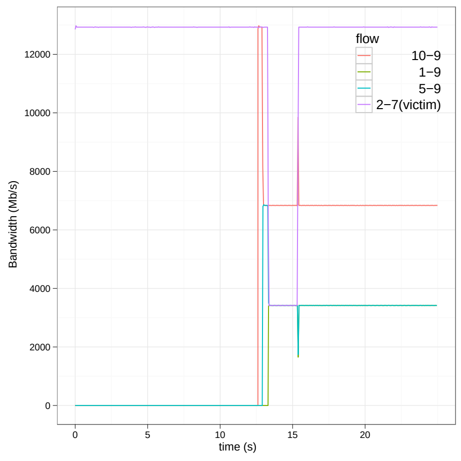


Figure 8. Per flow worst case bandwidth during congestion.



(a) Per flow throughput without dFtree.



(b) Per flow throughput with dFtree.

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

algorithm managed to recover the throughput for the victim flow 2-7 to 12.9 Gbps during congestion.

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

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

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

Without the dFtree algorithm, as shown in Fig. 11a, both victim flow 2-11 and congested flow 1-9 have a throughput of approximately 2.2 Gbps because they share both the

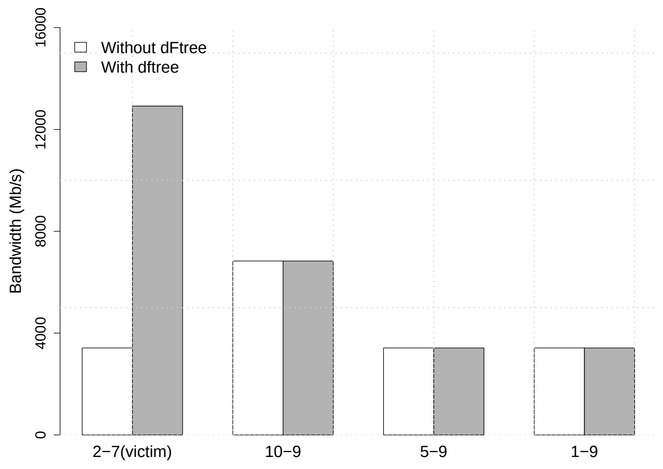


Figure 10. Per flow worst case bandwidth during congestion.

upstream (link 1) and downstream link (link 5) during the congestion after 13s. The victim flow 2-11 suffers severely by the HOL blocking even though it is not communicating with the hot-spot. Moreover, link 1 and 5 are QDR links where victim flow should be able to achieve a bandwidth of 12.9 Gbps.

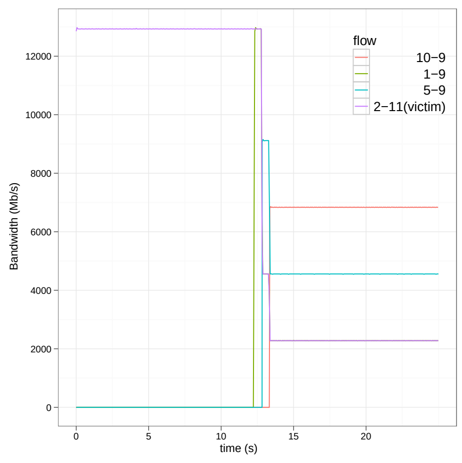
With dFtree, the victim flow (2-11) recovers to 12.9 Gbps after the congested flows are reassigned to the *'slow lanes'*. Furthermore, both congested flows 1-9 and 5-9 are transmitting at 3.4 Gbps because flow 2-11 is no longer sharing resources with the congested flow 1-9 after the *'slow lane'* assignment.

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

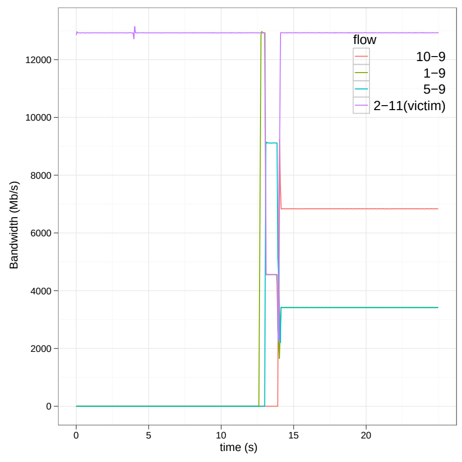
C. HPC Challenge Benchmark (HPCC)

In this experiment, we replaced the victim flows with the *HPC challenge* benchmark b_eff test suite [21]. The endpoint hot-spot is created using Perftest by running the traffic pattern presented in Fig. 3a for the non-oversubscribed topology and in Fig. 3b for the 2:1 oversubscribed topology. Simultaneously, we are running the HPCC benchmark in order to study the impact of congestion on the traffic generated by the HPCC benchmark. Even though the congested flows are still synthetically generated, this scenario resembles the network environment that an application could experience during congestion.

Table II shows the comparison of the HPCC b_eff results with and without the dFtree algorithm in the presence of congestion in a non-oversubscribed network. The most interesting observation is that the randomly ordered ring



(a) Per flow throughput without dFtree.



(b) Per flow throughput with dFtree.

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

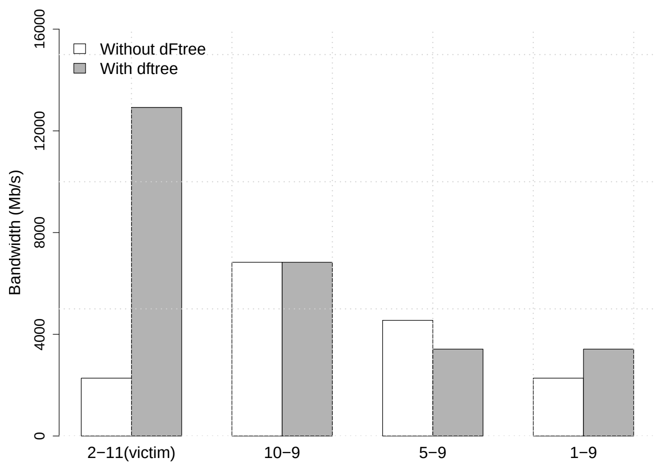


Figure 12. Per flow worst case bandwidth during congestion.

bandwidth increased by 49.34% with dFtree using only 2 VLs. We can see the improvement for all the latency and bandwidth tests, which is expected, as they correspond to the synthetic traffic patterns experiment that was carried out in the previous section. The results for the oversubscribed network are presented in Table III and the same trends are visible as for the non-oversubscribed network. These results clearly illustrate the performance gain with dFtree from the application traffic pattern's point of view.

D. *Simulation Results*

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

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

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

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

Table II
RESULTS FROM THE HPC CHALLENGE BENCHMARK WITH AND WITHOUT OUR DFTREE ALGORITHM FOR EXPERIMENT IN FIG. 3A.

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

Table III
RESULTS FROM THE HPC CHALLENGE BENCHMARK WITH AND WITHOUT OUR DFTREE FOR EXPERIMENT IN FIG. 3B.

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

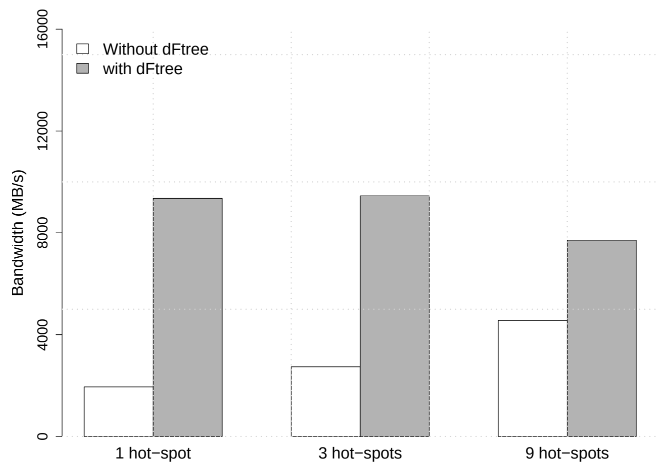


Figure 13. Simulation results for 648-switch.

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

VI. CONCLUSION AND FUTURE WORK

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

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

REFERENCES

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

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

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

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

[5] G. Rodriguez *et al.*, "Oblivious Routing Schemes in Extended Generalized Fat Tree Networks," *IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER '09.*, pp. 1–8, 2009. [Online]. Available: http://capinfo.e.ac.upc.edu/PDFs/dir09/file003460.pdf

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

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

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

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

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

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

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

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

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

[15] W. L. Guay, S.-A. Reinemo, O. Lysne, T. Skeie, B. D. Johnsen, and L. Holen, "Host side dynamic reconfiguration with infiniband," in *IEEE International Conference on Cluster Computing*, 2010, pp. 126–135.

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

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

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

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

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

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