sFtree: A Fully Connected and Deadlock-Free Switch-to-Switch Routing Algorithm for Fat-Trees

BARTOSZ BOGDANSKI, SVEN-ARNE REINEMO, FRANK OLAF SEM-JACOBSEN, and ERNST GUNNAR GRAN, Simula Research Laboratory

Existing fat-tree routing algorithms fully exploit the path diversity of a fat-tree topology in the context of compute node traffic, but they lack support for deadlock-free and fully connected switch-to-switch communication. Such support is crucial for efficient system management, for example, in InfiniBand (IB) systems. With the general increase in system management capabilities found in modern InfiniBand switches, the lack of deadlock-free switch-to-switch communication is a problem for fat-tree-based IB installations because management traffic might cause routing deadlocks that bring the whole system down. This lack of deadlock-free communication affects all system management and diagnostic tools using LID routing.

In this paper, we propose the sFtree routing algorithm that guarantees deadlock-free and fully connected switch-to-switch communication in fat-trees while maintaining the properties of the current fat-tree algorithm. We prove that the algorithm is deadlock free and we implement it in OpenSM for evaluation. We evaluate the performance of the sFtree algorithm experimentally on a small cluster and we do a large-scale evaluation through simulations. The results confirm that the sFtree routing algorithm is deadlock-free and show that the impact of switch-to-switch management traffic on the end-node traffic is negligible.

Categories and Subject Descriptors: C.2.2 [**Computer Communications Network**]: Network Protocol—*Routing protocols*

General Terms: Algorithms

Additional Key Words and Phrases: Routing, fat-trees, interconnection networks, InfiniBand, switches, deadlock

ACM Reference Format:
Bogdanski, B., Reinemo, S.-A., Sem-Jacobsen, F. O., and Gran, E. G. 2012. sFtree: A fully connected and deadlock-free switch-to-switch routing algorithm for fat-trees. ACM Trans. Architec. Code Optim. 8, 4, Article 55 (January 2012), 20 pages.
DOI = 10.1145/2086696.2086734 http://doi.acm.org/10.1145/2086696.2086734

1. INTRODUCTION

The fat-tree topology is one of the most common topologies for high performance computing (HPC) clusters today, and for clusters based on InfiniBand (IB) technology, the fat-tree is the dominating topology. This includes large installations such as the Roadrunner, Ranger, and JuRoPa [Top 500 2011]. There are three properties that make fat-trees the topology of choice for high performance interconnects: (a) deadlock freedom, the use of a tree structure makes it possible to route fat-trees without using virtual lanes for deadlock avoidance; (b) inherent fault-tolerance, the existence of multiple paths between individual source destination pairs makes it easier to handle network faults; (c) full bisection bandwidth, the network can sustain full speed communication between the two halves of the network.

Author's address: B. Bogdanski, Simula Research Laboratory, Norway; email: bartoszb@simula.no.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1544-3566/2012/01-ART55 $10.00
DOI 10.1145/2086696.2086734 http://doi.acm.org/10.1145/2086696.2086734

For fat-trees, as with most other topologies, the routing algorithm is crucial for efficient use of the network resources. The popularity of fat-trees in the last decade led to many efforts trying to improve the routing performance. This includes the current approach that the OpenFabrics Enterprise Distribution (OFED) [OpenFabrics Alliance 2011], the de facto standard for IB system software, is based on. That approach is presented in works by Gómez et al. [2007], Lin et al. [2004] and Zahavi et al. [2009]. Additionally, there exist several performance optimizations to this approach [Rodriguez et al. 2009; Bogdanski et al. 2010; Guay et al. 2011].

All the previous work, however, has one severe limitation when it comes to *switch-to-switch* communication. None of them support deadlock-free and fully connected switch-to-switch communication which is a requirement for efficient system management where all switches in the fabric can communicate with every other switch in a deadlock free manner. This is crucial because current IB switches support advanced capabilities for fabric management that rely on IP over IB (IPoIB). IPoIB relies on deadlock-free and fully connected IB routing tables. Without such support, features like the Simple Network Management Protocol (SNMP) for management and monitoring, Secure SHell (SSH) for arbitrary switch access, or any other type of IP traffic or applications using LID routing between the switches, will not work properly. The routing tables will only be fully connected and deadlock free from the point-of-view of the leaf switches.

There are algorithms that manage to obtain full connectivity on fat-tree topologies, but using them means sacrificing either performance or deadlock freedom. First of all, there is minhop that simply does shortest-path routing between all the nodes. As the default fallback algorithm implemented in the OpenSM subnet manager, it is not optimized for fat-tree topologies and, furthermore, it is not deadlock free. The alternatives include using a different routing algorithm, like layered shortest-path routing (LASH) [Skeie et al. 2002] or Up*/Down* [Schroeder et al. 1991]. LASH uses virtual lanes (VL) for deadlock avoidance and ensures full connectivity between every pair of nodes, but, like minhop, it is not optimized for fat-trees, which leads to suboptimal performance and longer route calculation times. Up*/Down* does not use VLs, but otherwise has the same drawbacks as LASH. Finally, there is a deadlock-free single-source shortest-path (DFSSSP) routing algorithm based on Dijkstra's algorithm [Domke et al. 2011]. However, it assumes that switch traffic will not cause a deadlock and uses VLs for deadlock avoidance for end-node traffic only.

In this paper we present, to the best of our knowledge, the first fat-tree routing algorithm that supports deadlock-free and fully connected switch-to-switch routing. Our approach retains all the performance characteristics of the algorithm presented by Zahavi [2009], and it is evaluated on a working prototype tested on commercially available IB technology. Our sFtree algorithm fully supports all types of single and multi-core fat-trees commonly encountered in commercial systems.

The rest of this paper is organized as follows. We introduce the InfiniBand Architecture in Section 2, followed by a description of fat-tree topologies and routing in Section 3. A description of the sFtree algorithm is given in Section 4 and in Section 5 we prove that it is deadlock free. Then we describe the setup of our experiments in Section 6, followed by a performance analysis of the result from the experiments and simulations in Section 7. Finally, we conclude in Section 8.

2. THE INFINIBAND ARCHITECTURE

InfiniBand is a lossless serial point-to-point full-duplex interconnect network technology that was first standardized in October 2000 [IBTA 2007]. The current trend is that IB is replacing proprietary or low-performance solutions in the high-performance computing domain [Top 500 2011], where high bandwidth and low latency are the key requirements.

The de facto system software for IB is OFED developed by dedicated professionals and maintained by the OpenFabrics Alliance [OpenFabrics Alliance 2011]. The sFtree algorithm that we propose in this paper was implemented and evaluated in a development version of OpenSM, which is the subnet manager distributed together with OFED.

2.1. Subnet Management

InfiniBand networks are referred to as *subnets*, where a subnet consists of a set of hosts interconnected using switches and point-to-point links. An IB fabric constitutes one or more subnets, which can be interconnected using routers. Hosts and switches within a subnet are addressed using local identifiers (LIDs) and a single subnet is limited to 49151 LIDs.

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

A major part of the SM's responsibility is to calculate routing tables that maintain full connectivity, deadlock freedom, and proper load balancing between all source and destination pairs. Routing tables must be calculated at network initialization time and this process must be repeated whenever the topology changes in order to update the routing tables and ensure optimal performance.

During normal operation the SM performs periodic *light sweeps* of the network to check for topology changes (e.g., a link goes down, a device is added, or a link is removed). If a change is discovered during a light sweep or if a message (trap) signaling a network change is received by the SM, it will reconfigure the network according to the changes discovered. This reconfiguration also includes the steps used during initialization.

IB is a lossless networking technology where flow-control is performed per *virtual lane* (VL) [Dally 1992]. VLs are logical channels on the same physical link, but with separate buffering, flow-control, and congestion management resources. The concept of VLs makes it possible to build virtual networks on top of a physical topology. These virtual networks, or layers, can be used for various purposes such as efficient routing, deadlock avoidance, fault-tolerance and service differentiation.

Our contribution in this paper will not make use of the service differentiation features in IB, but we will use VLs to show how switch-to-switch traffic influences end-node traffic when both traffic types are sharing one VL and when they are separated into different VLs. For more details about service differentiation mechanisms refer to IBTA [2007] and Reinemo et al. [2006].

3. FAT-TREE ROUTING

The fat-tree topology was introduced by Leiserson [1985] and has since become a common topology in HPC. The fat-tree is a layered network topology with equal link capacity at every tier (applies for balanced fat-trees) and is commonly implemented by building a tree with multiple roots, often following the m-port n-tree definition [Lin et al. 2004] or the k-ary n-tree definition [Petrini and Vanneschi 1995]. An XGFT notation is also used to describe fat-trees and was presented by Öhring [1995].

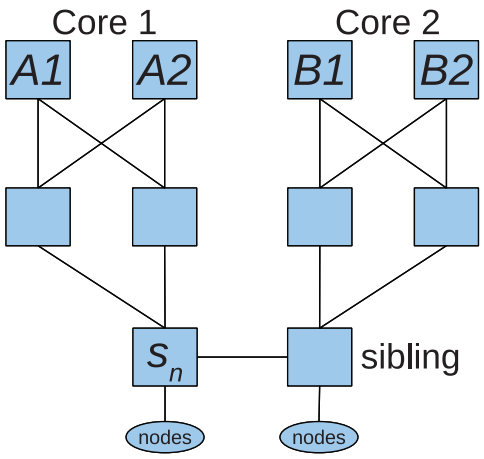


Fig. 1. A multi-core tree showing the s_n switch and its neighbor (sibling).

To construct larger topologies, the industry has found it to be more convenient to connect several fat-trees together rather than building a single large fat-tree. Such a fat-tree built from several single fat-trees is called a multi-core fat-tree. An example illustrating the concept is presented in Figure 1. Multi-core fat-trees may be interconnected through the leaf switches using horizontal links [Jülich Supercomputing Centre 2011] or by using an additional layer of switches at the bottom of the fat-tree and every such switch is connected to all the fat-trees composing the multi-core fat-tree [TACC 2011].

For fat-trees, as for most other network topologies, the routing algorithm is essential in order to exploit the available network resources. In fat-trees, the routing consists of two distinct phases: the upward phase in which the packet is forwarded from the source in the direction of one of the root switches and the downward phase when the packet is forwarded downwards to the destination. The transition between these two phases occurs at the lowest common ancestor, which is a switch that can reach both the source and the destination through its downward ports. Such an implementation ensures deadlock freedom, and the implementation presented in Zahavi et al. [2009] also ensures that every path towards the same destination converges at the same root (top) switch such that all packets toward that destination follow a single dedicated path in the downward direction. By having a dedicated downward path for every destination, contention in the downward phase is effectively removed (moved to the upward stage) so that packets for different destinations have to contend for output ports in only half of the switches on their path. In oversubscribed fat-trees, the downward path is not dedicated and is shared by several destinations.

3.1. Switch-to-Switch Routing

The fat-tree routing described in the previous section does not include switch-to-switch communication. In fact, the switch-to-switch communication has been ignored for a long time because the switches themselves lacked the necessary intelligence to be able to support advanced system management techniques, and originated very little traffic. In other words, IB switches were considered to be transparent devices that could be safely ignored from the point of view of the routing algorithm. Even today, switch-to-switch paths are treated as secondary paths (not balanced across ports), and are routed in the same manner as compute node paths. It means that to find a path between any two switches, the lowest common ancestor must be found, and the traffic is always forwarded through the first available port. Moreover, such a routing

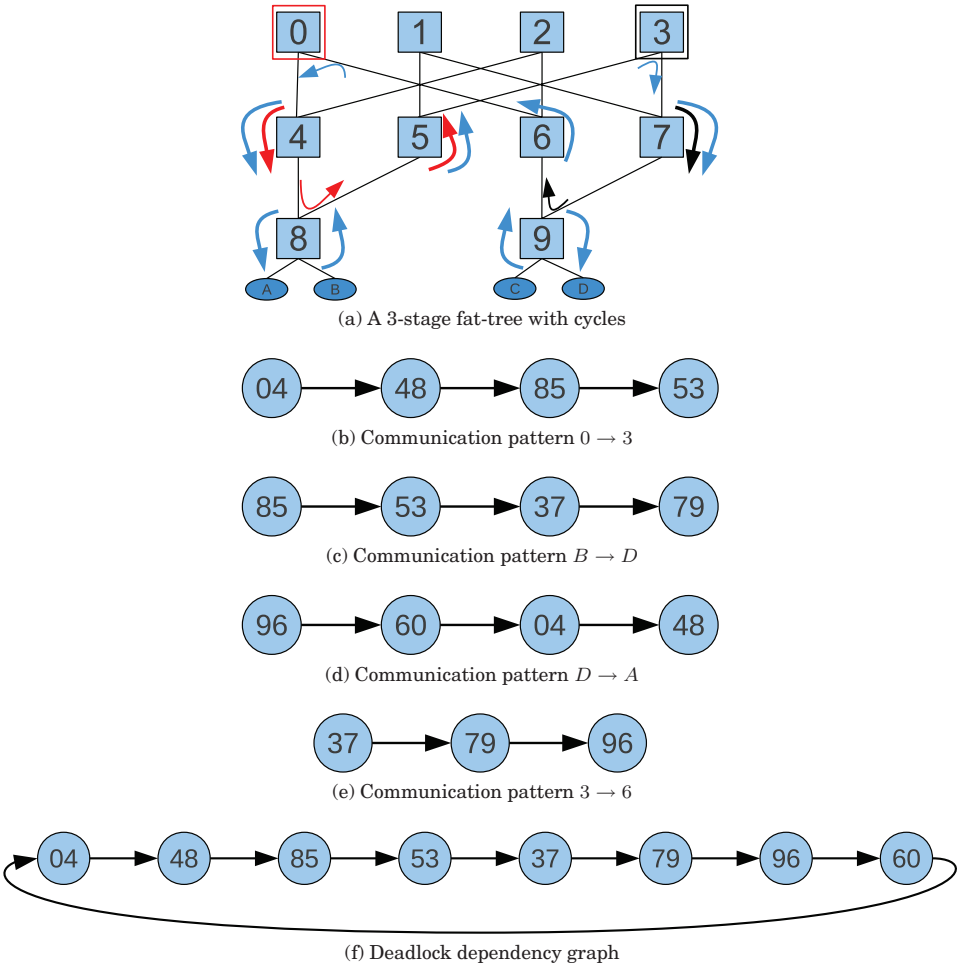
scheme does not provide connectivity between those switches which do not have a lowest common ancestor. These are usually the root switches in any fat-tree topology, but the problem can also manifest itself for all non-leaf switches in multi-core fat-trees or in an ordinary fat-tree with a rank (the number of stages in a tree) greater than two depending on the cabling as discussed in Section 4.1.

Today, full connectivity between all the nodes—both the end-nodes and the switches—is a requirement. IB diagnostic tools rely on LID routing and need full connectivity for basic fabric management and monitoring. More advanced tools like *perftest* used for benchmarking employ IPoIB. Moreover, IPoIB is also required by non-InfiniBand-aware management and monitoring protocols and applications like SNMP or SSH. Being an encapsulation method that allows running TCP/IP traffic over an IB network, IPoIB relies on the underlying LID routing. In the past, there was no requirement for connectivity between the switches because they lacked the capability to run many of the above mentioned tools and protocols. Today, however, switches are able to generate arbitrary traffic, can be accessed like any other end-node, and often contain an embedded SM. Therefore, the requirement for full connectivity between all the switches in a fabric is essential.

From the system-wide perspective of an interconnection network, deadlock freedom is a crucial requirement. Deadlocks occur because network resources such as buffers or channels are shared and because IB is a lossless network technology, i.e., packet drops are usually not allowed. The necessary condition for a deadlock to happen is the creation of a cyclic credit dependency. This does not mean that when a cyclic credit dependency is present, there will always be a deadlock, but it makes the deadlock occurrence possible. In this paper we demonstrate that when a deadlock occurs in an interconnection network, it prevents a part of the network from communicating at all. Therefore, the solution that provides full connectivity between all the nodes has to be deadlock free.

An example of a deadlock occurring in a fat-tree is presented on Figure 2(a). This is a simple 3-stage fat-tree topology that was routed without any consideration for deadlock freedom. We show that only four communication pairs are required to create a deadlock. The first pair, $0 \rightarrow 3$, is marked with red arrows, and the second pair, $3 \rightarrow 6$, is marked with black arrows. These two pairs are the switch-to-switch communication patterns. Two node-to-node pairs are also present and marked with blue arrows: $B \rightarrow D$ and $D \rightarrow A$. The deadlock that occurs with these four pairs is further illustrated using channel dependency subgraphs (see Definition 5.5 and Definition 5.6) that are shown on Figure 2(b) through Figure 2(e). A channel dependency graph is constructed by representing links in the network topology by vertices in the graph. Two vertices are connected by an edge if a packet in the network can hold one link while requesting the next one. On the figures, the number in each circle is the link between the two devices, for example, 04 is the link between the switch marked as 0 and the switch marked as 4 in Figure 2(a). Looking at these four channel dependency graphs, we are able to see that they in fact contain a cycle, which is shown on Figure 2(f). This example also shows that any 3-stage fat-tree can deadlock with node-to-node and switch-to-switch traffic present, if minhop is run on it. This is because minhop is unable to route a ring of 5 nodes or bigger in a deadlock-free manner [Guay et al. 2010], and almost any (apart from a single-root tree) 3-stage fat-tree will contain a 6-node ring. By having shown this deadlock example, we have demonstrated the need for an algorithm that will provide deadlock-free routing in fat-tree topologies when switch-to-switch communication between all switches is present.

Currently, the only deadlock-free routing algorithm for IB that complies with full connectivity requirement is LASH. It uses VLs to break the credit cycles in channel dependency graphs and provides full connectivity between all the nodes in the fabric.



(a) A 3-stage fat-tree with cycles

(b) Communication pattern $0 \rightarrow 3$

(c) Communication pattern $B \rightarrow D$

(d) Communication pattern $D \rightarrow A$

(e) Communication pattern $3 \rightarrow 6$

(f) Deadlock dependency graph

Fig. 2. An example of a deadlock occurring in a fat-tree.

However, for fat-tree topologies, LASH is a suboptimal choice because it does not exploit the properties of the fat-trees when assigning the paths, and thus gives worse network performance than ordinary fat-tree routing as shown in the performance evaluation section and also shown by Domke et al. [2011]. Additionally, route calculation with LASH is time consuming and due to the shortest-path requirement, it unnecessarily uses VL resources to provide deadlock freedom for a fat-tree topology. Another routing protocol that could theoretically support all-to-all switch-to-switch communication in a deadlock-free manner in fat-trees is DFSSSP. However, it has the same limitations as LASH when it comes to performance and route calculation time, and it does not break credit loops when they occur between non-HCA nodes, which means that switch-to-switch communication is not deadlock free.

Deadlocks can be avoided by using VLs that segment the available physical resources as LASH or DFSSSP do. We show, however, that for fat-trees, using VLs for deadlock avoidance is inefficient because the whole fabric can be routed in a deadlock-free manner by only using a single VL. The additional VLs can be used for other purposes like QoS.

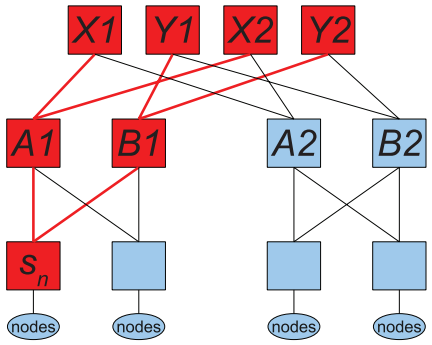


Fig. 3. A 3-stage fat-tree with a sample subtree converging to s_n.

4. THE SFTREE ALGORITHM

Our proposed design is based on the fat-tree routing algorithm presented by Zahavi [2009] and can be treated as a modular extension, which for every *switch source* takes all *switch destinations* and, if such a destination is marked as unreachable by the default fat-tree routing tables (meaning that it has no *up* $\rightarrow$ *down* path), it finds a deadlock-free path.

4.1. Design of the Algorithm

The current fat-tree routing algorithm proposed by Zahavi et al. [2009] is already capable of assigning switch-to-switch paths, but only those that follow the up/down turn model on which the fat-tree routing is based. For example, in Figure 3, the switches *A1* and *A2* would have connectivity by going up through switch *X1* or *X2*. However, switches *A1* and *B1* do not have a lowest common ancestor, so one possibility for them to achieve connectivity is to go through one of the common descendants (shortest-path method). The alternative method (suboptimal path or non-shortest-path) for them would be to go through any leaf switch in the topology, not necessarily located directly below them. This would require first going up, then down, until reaching a switch that has a path to both of the communicating switches, and then up again and down again until the destination. In this paper, we use the second approach, because it can be implemented without the need for additional VLs. The shortest-path method is only used for a specific set of switches (an example of such a set is marked in red on Figure 3). To guarantee deadlock freedom, the key to our approach is that all the switches in the topology collectively choose the same leaf switch for communication.

To achieve deadlock-free full connectivity between the switches, we propose using an upsidedown subtree whose root is one of the leaf switches (formally defined in Definition 5.7) and illustrated in Figure 3. The subtree concept allows us to localize all the down/up turns in a fabric to a single, deadlock-free tree. As we demonstrate in Section 5, accommodating all the prohibited turns in a subtree is deadlock free, and it permits full connectivity if a subtree root (s_n) is chosen properly. To illustrate the routing through a subtree concept, we can observe that the switches *X1* and *Y1* will communicate through a subtree root s_n whereas *X1* and *X2* will go through switch *A1*. Combining the description of the subtree with the discussion in the previous paragraphs, we can observe that switches external to the subtree must communicate through it to reach other switches previously marked as unreachable, which leads to usage of suboptimal (non-shortest) paths. In other words, any switch in the fabric that cannot reach any other switch using the traditional upward to downward scheme, must first send the packets to the subtree and the packets are passed down in the subtree until they reach a switch that

has a path to the destination switch. In this switch the packets make a U-turn, i.e., a down-to-up turn and follow a traditional up-to-down path towards the destination. The deadlock freedom of this approach relies on the fact that all the U-turns take place only within the subtree and that any up-to-down turn will leave the subtree. Traffic leaving the subtree will not be able to circulate back into it because, by design, there are no U-turns outside of the subtree. This will be further explained in the proof in Section 5. It is also important to note that this communication scheme does not change the existing upward to downward paths between the switches that can reach each other using the traditional fat-tree routing.

ALGORITHM 1: Select a subtree root function

Require: Routing table has been generated.
Ensure: A subtree root (sw_{root}) is selected.

```
1:  found = false
2:  for sw_leaf = 0 to max_leaf_sw do
3:    if found == true then
4:      break
5:    end if
6:    sw_root = sw_leaf
7:    found = true
8:    for dst = 1 to max_dst_addr do
9:      if sw_root.routing_table[dst] == no_path then
10:       found = false
11:       break
12:     end if
13:   end for
14: end for
15: if found = false then
16:   sw_root = get_leaf(0)
17: end if
```

In detail, the algorithm first finds a subtree by finding a subtree root. The pseudocode for this step is presented in Algorithm 1. The algorithm traverses all the leaf switches in the topology and, for each leaf switch, checks whether any of the switch destination addresses are marked as unreachable in its routing table. If all the switch destinations in the routing table are marked as reachable, the first encountered leaf switch is selected as a subtree root switch. There may be as many potential subtrees in a fat-tree as there are leaf switches having full connectivity, however, only one of those leaf switches will be selected as the subtree root and there will be only one subtree with a root in this particular leaf switch. Algorithm 2 is only called for those switch pairs that do not have a path established using the traditional fat-tree routing.

When the switch-to-switch routing function is called, the first step is to determine whether the routing table of the *subtree root* (sw_{root}) contains a path to the *destination switch* (sw_{dst}) as shown in Algorithm 2 line 1. If it does, then the path to the *destination switch* is inserted into the routing table of the *source switch* (sw_{src}), and the output port for the *destination switch* is the same as the output port for the path to the *subtree root* (lines 2-5). Because the switch-to-switch routing function is called for every switch, in the end, all the paths to each unreachable *destination switch* will converge to the subtree. In other words, the selected *subtree root* is the new target for all the unreachable *destination switches*. The down/up turn will take place at the first switch located in the subtree that has an upward path both to the source switch and the destination switch.

In a single-core fat-tree there will always be a path from the *source switch* to the *subtree root*, and the *subtree root* will have a path to every *destination switch*. However,

ALGORITHM 2: Switch-to-switch routing function

Require: Subtree root (sw_{root})
Ensure: Each sw_{src} reaches each sw_{dst} at worst by sw_{root}.
1: **if** *found* **or** $sw_{root}.routing\_table[dst] \neq no\_path$ **then**
2: $sw_{src}.routing\_table[dst] = sw_{src}.routing\_table[sw_{root}.addr]$
3: $get\_path\_length(sw_{src}, null, dst, sw_{root}, hops)$
4: $set\_hops(sw_{src}, hops)$
5: **return true**
6: **else if** $(sw_{sib} = get\_sibling\_sw(sw_{root})) \neq null$ **then**
7: **if** $sw_{src}.routing\_table[sw_{sib}.addr] \neq no\_path$ **then**
8: **if** $sw_{sib}.routing\_table[dst] \neq no\_path$ **then**
9: $sw_{root}.routing\_table[dst] = get\_port\_to\_sibling(sw_{sib})$
10: $hops = get\_hops(sw_{sib}, dst)$
11: $set\_hops(sw_{root}, hops + 1)$
12: $hops = 0$
13: $sw_{src}.routing\_table[dst] = sw_{src}.routing\_table[sw_{sib}.addr]$
14: $get\_path\_length(sw_{src}, null, dst, sw_{root}, hops)$
15: $set\_hops(sw_{src}, hops)$
16: **return true**
17: **end if**
18: **end if**
19: **end if**
20: **print** 'SW2SW failed for sw_{src} and sw_{dst}.'
21: **return false**

for complex multi-core or irregular fat-trees this may not always be the case, and the second part of the pseudocode presented in Algorithm 2 deals with cases where the best-effort approach is needed and the *subtree root* does not have paths to all the destinations. For best effort, it is necessary to check whether the subtree root has a direct neighbor (to which they are connected through horizontal links as shown in Figure 1). Next, a check is done to verify whether that neighbor, also called a sibling, has a path to the *destination switch*. This is done in lines 7 and 8 of the presented pseudocode. If the sibling exists and it has a path to the *destination switch*, a path is set both on the *subtree root* and the originating *source switch* to the *destination switch* through the sibling switch (lines 9-16). In other words, the target for the unreachable *destination switch* will still be the *subtree root*, but in this case, it will forward the packets to the *destination switch* to its sibling which in turn will forward them to the *destination switch*.

If both previous steps do not return from the function with a true value, the algorithm has failed to find a path between two switches. This occurs only for such topologies on which it is not recommended to run the fat-tree routing at all due to multiple link failures or very irregular connections between the nodes. An example of such a topology would be two fat-trees of different sizes connected to each other in an asymmetrical manner, for example, from a few middle-stage switches on the larger tree to roots on the smaller one. Using various command-line parameters, fat-tree routing in OpenSM can be forced to run on such a topology, however, it will give suboptimal routing not only when it comes to performance, but also when we consider connectivity.

4.2. OpenSM Implementation

OpenSM requires that every path be marked with a hop count to the destination. The older versions of the fat-tree routing algorithm (pre-3.2.5, current ones are 3.3.x) calculated the number of hops using the switch rankings in the tree. In the newer

versions, the counting is done using a simple counter in the main routing function. Because the switch-to-switch routing we perform is totally independent of the main routing done by the fat-tree algorithm, we could not use the counters stored there, and because of the possible zig-zag paths (i.e., paths not following the shortest hop path), counting using the switch ranks is unreliable. Therefore, we devised a simple recurrence function for hop calculation, called *get_path_length* only to be used during the switch-to-switch routing (it is called in line 14 of the pseudocode shown in Algorithm 2). This function iterates over the series of the switches that constitute the path, and when it reaches the one having a proper hop count towards the destination, by backtracking it writes the correct hop count into the routing tables of the switches on the whole path.

Moreover, one of the enhancements that we made is to choose the subtree root in such a way that the subnet manager node (the end-node on which the subnet manager is running) is not connected to the switch marked as the subtree root. This will draw the switch-to-switch traffic away from the subnet manager, thus, not creating a bottleneck in a critical location.

5. DEADLOCK FREEDOM PROOF

In this section we prove that the sFtree algorithm is deadlock free. The first part of the proof contains the definitions of the terms used later in the text.

Definition 5.1. By switch s_n we mean a switch with a rank n. Root switches have rank $n = 0$.

Definition 5.2. By a *downward channel* we mean a link between s_{n-1} and s_n where the traffic is flowing from s_{n-1} to s_n. By an *upward channel* we mean a link between s_n and s_{n-1} where the traffic is flowing from s_n to s_{n-1}.

Definition 5.3. A *path* is defined as a series of switches connected by channels. It begins with a source switch and ends with a destination switch.

Definition 5.4. A *U-turn* is a turn where a downward channel is followed by an upward channel.

Definition 5.5. A *channel dependency* between channel c_i and channel c_j occurs when a packet holding channel c_i requests the use of channel c_j.

Definition 5.6. A *channel dependency graph* $G = (V, E)$ is a directed graph where the vertices V are the channels of the network N, and the edges E are the pairs of channels (c_i, c_j) such that there exists a (channel) dependency from c_i to c_j.

Definition 5.7. By a *subtree* we mean a logical upside down tree structure within a fat-tree that converges to a single leaf switch s_n, which is the single root of the subtree. A subtree expands from its root and its leaves are all the top-level switches in the fat-tree. Any upward to downward turn will leave the subtree structure.

The following theorem states the sufficient condition for deadlock freedom of a routing function [Duato et al. 2003].

THEOREM 5.1. *A deterministic routing function R for network N is deadlock free if and only if there are no cycles in the channel dependency graph G.*

Next, we prove the necessary lemmas followed by the deadlock freedom theorem of the sFtree algorithm.

LEMMA 5.8. *There exists at least one subtree within a fat-tree that allows full switch-to-switch connectivity using only U-turns within that subtree.*

PROOF. In a fat-tree, from any leaf we can reach any other node (including all the switches) in the fabric using an up/down path or a horizontal sibling path. This is because every end-node can communicate with every other end-node. Because end-nodes are directly connected to the leaf switches, it follows that leaf switches are able to reach any other node in the topology. Consequently, if it contains no link faults, any leaf switch can act as the root of the subtree. □

LEMMA 5.9. *There can be an unlimited number of U-turns within a single subtree without introducing a deadlock.*

PROOF. A subtree is a logical tree structure and two types of turns can take place within it: U-turns, i.e., downward-to-upward turns, and ordinary upward-to-downward turns. The U-turns cannot take place at the top switches in a subtree (and fat-tree) because these switches have no output upward ports. Any upward to downward turn will leave the subtree according to Definition 5.7.

Because a subtree is a connected graph without any cycles, it is deadlock free by itself. Any deadlock must involve U-turns external to the subtree since a U-turn is followed by an upward to downward turn leaving the subtree, which is also shown on Figure 2(a). All the traffic going through an up/down turn originating in the subtree will leave the subtree when the turn is made and follow only the downward channels to the destination. Such a dependency can never enter the subtree again and can never reach any other U-turn, and so, cannot form any cycle. □

THEOREM 5.2. *The sFtree algorithm using the subtree method is deadlock free.*

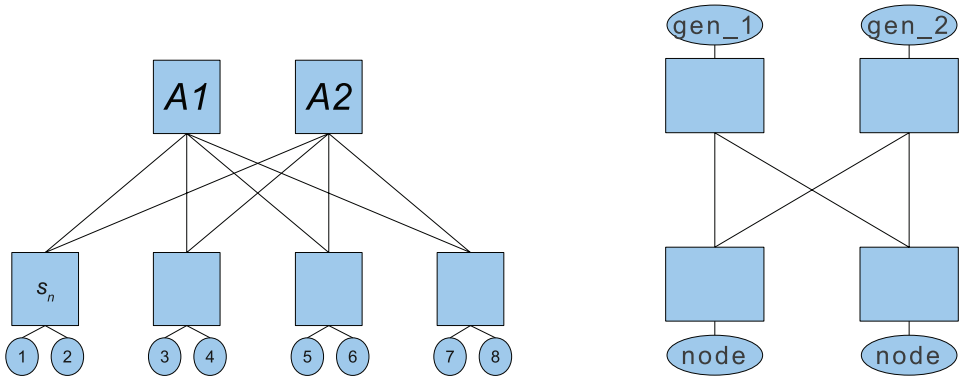
PROOF. Following Lemma 5.8 and Lemma 5.9, we observe that a deadlock in a fat-tree can only occur if there are U-turns outside the subtree. Such a situation cannot happen due to the design of the sFtree algorithm where all U-turns take place within a subtree. The traffic leaving a subtree will not circle back to it because once it leaves the subtree, it is forwarded to the destination through downward channels only, and it is consumed, thus, no cyclic credit dependencies will occur in the topology. □

6. EXPERIMENT SETUP

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

6.1. Experimental Test Bed

Our test bed consisted of eight nodes and six switches. Each node is a Sun Fire X2200 M2 server [Oracle Corporation 2006] 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 were: two 36-port Sun Datacenter InfiniBand Switch 36 [Oracle Corporation 2011a] QDR switches which acted as the fat-tree roots; two 36-port Mellanox Infiniscale-IV QDR switches [Mellanox Technologies 2009], and two 24-port SilverStorm 9024 DDR switches [Qlogic 2007], all of which acted as leaves with the nodes connected to them. The port speed between the QDR switches was configured to be 4x DDR, so the requirement for the constant bisectional bandwidth in the fat-tree was assured. The cluster was running the Rocks Cluster Distribution 5.3 with kernel version 2.6.18-164.6.1.el5-x86_64, and the IB subnet was managed using a modified version of OpenSM 3.2.5 with our sFtree implementation. The topology on which we performed the measurements is shown in Figure 4(a). Switches *A1* and *A2* are the Sun devices, which are able to produce and consume traffic. These two switches were used as follows. We established a connection between the switches, and used the sFtree



(a) The experimental cluster. (b) Deadlock scenario topology.

Fig. 4. The clusters used during the experiments.

routing algorithm to forward traffic in a deadlock-free manner between the devices through the chosen subtree root s_n.

The Sun switches are the only managed switches in the topology, and they are able to generate and consume arbitrary traffic. The firmware installed on the switches is 1.1.3-2, and the BIOS revision is NOW1R112. Those switches have an enhanced port 0 which is connected to the IB switching fabric using a 1x SDR link (signaling rate is 2.5 Gb/s, and the effective speed is 2 Gb/s). That port 0 is also connected to the internal host using a 1x PCI Express 1.1. The IPoIB interfaces on both switches were configured with an MTU size of 2044 octets, which is the maximum supported size in the IPoIB Datagram Mode (IPoIB-UD) [Chu and Kashyap 2006] (in most implementations). The MTU size on the switches cannot be increased because there is no support for the optional IPoIB Connected Mode (IPoIB-CM) [Kashyap 2006]. The link MTU provides a limit to the size of the payload that may be used. The 2044 octet MTU is in fact a 2048 octet MTU minus the 4-octet encapsulation overhead, which is done to reduce problems with fragmentation and path-MTU discovery.

Furthermore, we constructed a separate topology to show the effects of a deadlock in a small fat-tree when a suboptimal routing engine is chosen. The topology is shown in Figure 4(b). That fat-tree can be unfolded into a ring which, when routed improperly, will deadlock. Because the 1x SDR generators built into the Sun Datacenter InfiniBand Switches injected packets too slow to create a deadlock, we created an artificial scenario in which we connected ordinary end-nodes, namely, *gen_1* and *gen_2* to the root switches, and routed the topology with sFtree and minhop algorithms. We used the same settings for the hardware as in the previous scenario.

6.2. Simulation Test Bed

To perform large-scale evaluations and verify the scalability of our proposal, we use an InfiniBand model for the OMNeT++ simulator [Gran and Reinemo 2011]. The model contains an implementation of HCAs and switches with support for routing tables and virtual lanes. The network topology and the routing tables were generated using OpenSM and converted into OMNeT++ readable format in order to simulate real-world systems. The simulations were performed on a 648-port fat-tree topology, illustrated in Figure 5. This topology is the largest 2-stage fat-tree topology that can be constructed using 36-port switch elements. When fully populated, this topology consists of 18 root switches and 36 leaf switches. We chose the 648-port fabric because it is a common configuration used by switch vendors in their own 648-port systems [Oracle

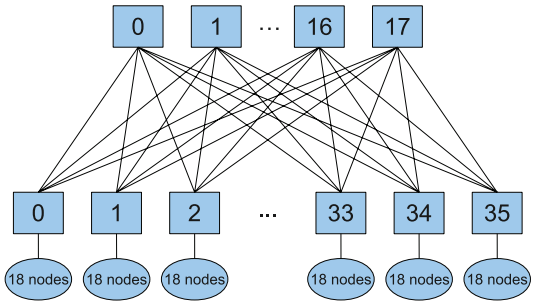


Fig. 5. A 648-port fat-tree topology.

Corporation 2011b; Voltaire 2011; Mellanox Technologies 2011]. Additionally, such switches are often connected together to form larger installations like the JuRoPa supercomputer [Jülich Supercomputing Centre 2011].

For the simulations we used a few different traffic patterns, but because of space limitations we are only presenting the results for the uniform traffic. All other simulations using nonuniform traffic exhibited the same trends. The uniform traffic pattern had a random destination distribution and the end-nodes were sending only to other end-nodes and switches only to other switches. Each simulation run was repeated eight times with different seeds and the average of all simulation runs was taken. The message size was 2 kB for every simulation, and the packet generators and sinks at the switches were configured to match those from the hardware test. These simulations were performed to verify whether the switch traffic influences the end-node traffic, and if so, to what degree for large-scale scenarios. We also performed simulations for the different network topologies listed in Table IV, and the results show the same trends as the results for the 648-port switch.

7. PERFORMANCE EVALUATION

Our performance evaluation consists of measurements on the experimental fat-tree cluster and simulations of large-scale topologies. Before running the performance evaluation, we confirmed that the sFtree algorithm works as expected by using the command-line tool *ibtracert* and, after setting up IPoIB, *ping* and *route*. For measurements on the cluster, we use the results from the HPCC benchmark [HPCC 2011] run under MVAPICH2-1.4.1 [MVAPICH2 2011] to show how the sFtree algorithm impacts application traffic. In the HPCC benchmark the number of ring patterns was increased from 30 to 50000 to provide a better average when comparing the results. For the deadlock scenario, we used *perftest* to initiate communication between the nodes. For the simulations, we use the achieved average throughput per end node or per switch as the metric for measuring the impact of switch traffic on the simulated 648-port network topology.

7.1. Experimental Results

In the HPCC experiments we were simultaneously running the HPCC benchmark on the compute nodes and generating the switch-to-switch traffic that consisted of sequential reads and writes of random blocks of data from the internal hard drive. The data was sent using the TCP connection between the IB interfaces on the switches. The average data throughput between the switches was 25 MB/s both ways. There are several reasons why the switches are not able to send with the full capacity of their SDR link. First, the only available transport service is the IPoIB-UD, which

Table I. Results from the HPC Challenge Benchmark with All Communication in VL0

Network latency and throughput	a) SW2SW off	b) SW2SW on	c) Difference
Min Ping Pong Lat. (ms)	0.001997	0.001997	0.0%
Avg Ping Pong Lat. (ms)	0.002267	0.002266	−0.044%
Max Ping Pong Lat. (ms)	0.002369	0.002369	0.0%
Naturally Ordered Ring Lat. (ms)	0.002193	0.002193	0.0%
Randomly Ordered Ring Lat. (ms)	0.002157	0.002165	0.371%
Min Ping Pong BW (MB/s)	1587.248	1586.048	−0.076%
Avg Ping Pong BW (MB/s)	1588.806	1588.379	−0.027%
Max Ping Pong BW (MB/s)	1590.559	1591.162	0.038%
Naturally Ordered Ring BW (MB/s)	1488.926	1495.895	0.468%
Randomly Ordered Ring BW (MB/s)	1226.432	1226.347	0.007%

Table II. Results from the HPC Challenge Benchmark with Node Communication in VL1 and Switch Communication in VL0

Network latency and throughput	a) SW2SW off	b) SW2SW on	c) Difference
Min Ping Pong Lat. (ms)	0.001997	0.001997	0.0%
Avg Ping Pong Lat. (ms)	0.002300	0.002291	−0.391%
Max Ping Pong Lat. (ms)	0.002429	0.002429	0.0%
Naturally Ordered Ring Lat. (ms)	0.002289	0.002313	1.048%
Randomly Ordered Ring Lat. (ms)	0.002197	0.002214	0.773%
Min Ping Pong BW (MB/s)	1586.048	1586.048	0.0%
Avg Ping Pong BW (MB/s)	1588.650	1588.486	−0.01%
Max Ping Pong BW (MB/s)	1591.766	1591.011	−0.047%
Naturally Ordered Ring BW (MB/s)	1505.4256	1487.8035	−1.171%
Randomly Ordered Ring BW (MB/s)	1228.2845	1228.3750	0.007%

has a limited MTU size. No direct application access to the IB interface is possible because there is no user space access for establishing a Queue Pair (QP), so all the user data has to be encapsulated within IP and then forwarded through IB. Second, the packet-processing seems to be limited by the CPU whose utilization was constantly above 90%. In addition, there is no support for RDMA Read and Write requests, which further increases the CPU usage.

During the tests, we used a few different network performance tools (qperf, netperf, NetPIPE). These tools also support UDP traffic benchmarking, but after saturating the internal link with UDP traffic, the responsiveness of the switches was severely limited due to the high CPU utilization. Furthermore, we were not able to see any difference in the experimental results, and for that reason, we decided to use TCP traffic.

The experiment was repeated twice: In the first scenario, we mapped both the switch and the application traffic (HPCC) to VL0 and, in the second scenario, the application traffic was running on VL1 and switch traffic on VL0. To establish a base reference, for each scenario, we also disabled the switch traffic and measured the application traffic only. The reference results are presented in the second column of Tables I and II.

7.1.1. First scenario. Table I shows the results for the first scenario. The most interesting observation is that the influence of the switch traffic on the application traffic is negligible. There are variations between the reference results and the scenario results, but they are very small as seen in the last column of Table I. The influence of the switch traffic on the end-node traffic is negligible because there are more nodes than switches in the fabric and the nodes are able to inject traffic 8 times faster using all the links in the fabric. In comparison, the switches are unable to send at their full capacity because of the technical limitations described in the previous paragraphs and they only use a single path to communicate.

7.1.2. Second scenario. The results for the second scenario are presented in Table II. The key observation here, as seen in the last column of Table II is that for small data

streams, which emulate the management traffic between the switches, there is almost no influence of switch traffic on the end-node traffic when compared with the reference results presented in the second column of Table II. However, the latencies for both the natural and random rings are higher when switch-to-switch traffic is present. The difference is only 1%, but this pattern also corresponds well with the simulation results where using an additional VL for traffic separation decreases the overall throughput (see Section 7.2.2).

To conclude, the results from both scenarios clearly illustrate that there is little or no performance overhead when using the sFtree algorithm. However, we can suspect that if the volume of the traffic generated by the switches was larger, we would observe a drop in performance when assigning different VLs to different types of traffic (as in the second scenario). Because our solution is deadlock free, there is no need to use additional VLs for deadlock-avoidance. Still, one of the limitations of the hardware tests is the fact that only two switches out of six were able to send and receive traffic. The other limitation was the topology size and the fact that a large part of the node-to-node communication was taking part in the crossbar switches. Therefore, in Section 7.2 we will show through simulations that the same trends hold for a larger number of switches.

7.1.3. Deadlock scenario. We also created an artificial deadlock scenario. For the topology shown on Figure 4(b), we run the perftest to measure the sent bandwidth over time. As mentioned in Section 6, the built in generators in the currently available switches are not able to create a deadlock due to their low bandwidth. Therefore, we substituted those generators with fully capable end-node generators and configured the routing to emulate the paths assigned by the sFtree and minhop routing algorithms. However, the implementation of minhop available in OpenSM will not deadlock on such a topology, so by modifying the routing table on the fly we managed to obtain the desired effect. The reason why minhop will not deadlock on a ring constructed out of four nodes is the fact that its implementation in OpenSM assigns symmetric paths for *source-destination* and *destination-source* pairs, thus, it breaks the credit loop necessary for a deadlock to occur. However, it is a well known fact that minhop deadlocks on at least a 5-node ring [Guay et al. 2010], and any 3-stage fat-tree that has more than one root contains such a ring. Therefore, almost any 3-stage or larger fat-tree routed with minhop will potentially deadlock, thus, it makes our experiment valid and practical.

In this experiment all the communication streams were launched during the first five seconds. By analyzing Figure 6, we observe that minhop routing deadlocks immediately after the last stream that closes the credit loop is added, and then that the average per node bandwidth drops to 0 MB/s. For the sFtree routing algorithm, we observe that the average per node network bandwidth stabilizes after the last stream is added because all the U-turns take place on one leaf switch, and no credit loop is created.

This experiment shows the effect that a deadlock has on a fat-tree topology which is routed improperly. The sFtree routing algorithm, however, is safe to use and will not deadlock on a fat-tree topology as proven in Section 5.

7.2. Simulation Results

An important question is how well the sFtree algorithm scales. Specifically, the purpose of the simulations was to show that the same trends that were observed in the experiments exist when the number of switches generating the traffic will be much larger and the network topology corresponds to real systems. Like the hardware measurements, we performed two different tests. In the first scenario, we mapped the switch-to-switch and node-to-node traffic both to VL0, and, for the second scenario, we separated the node-to-node traffic by mapping it to VL1. Furthermore, for every measurement the

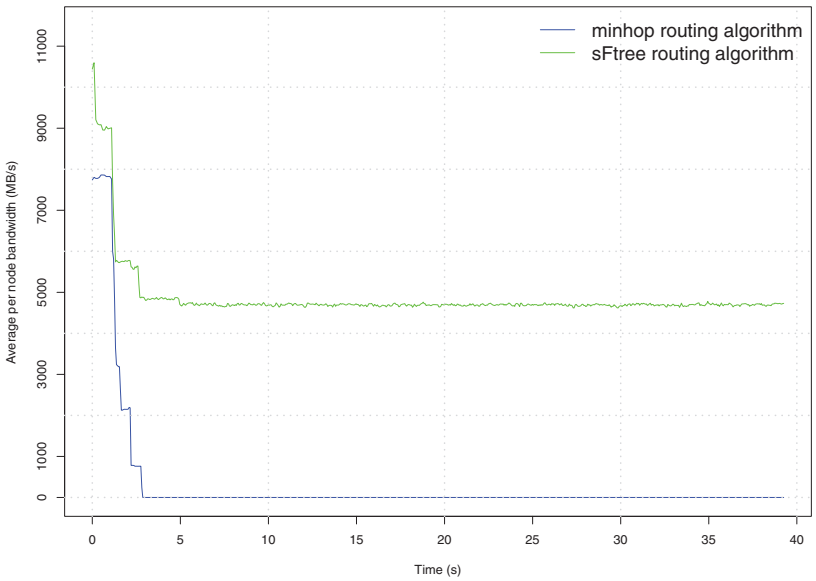
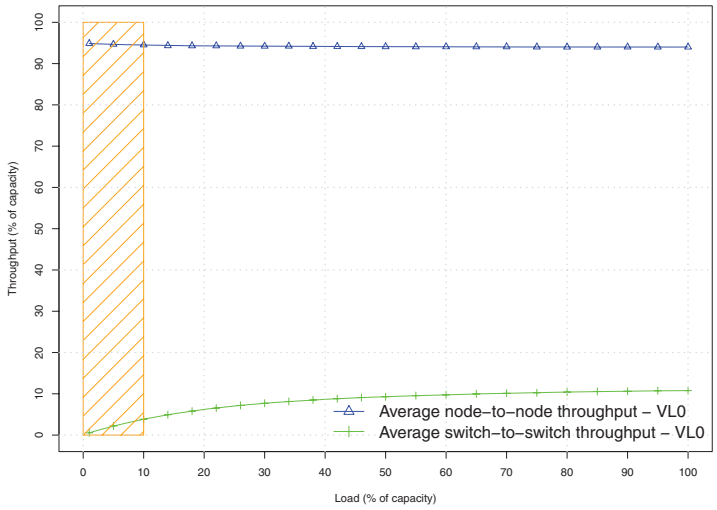


Fig. 6. Average per node network bandwidth comparison for minhop and sFtree.

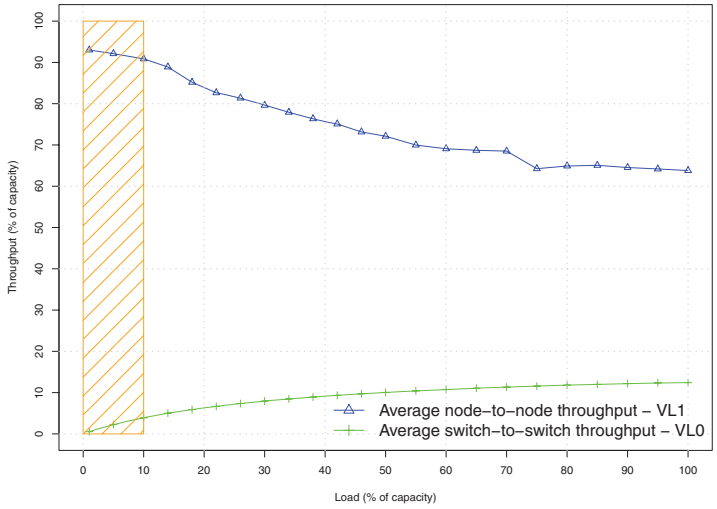
nodes were communicating at their full capacity, and the load of the switch traffic was gradually increased from 0% to 100%. It has to be noted that the scale on the Y-axis of the graphs presented in Figure 7 corresponds to the node-to-node sending and receiving capacity which is 4x DDR (20 Gb/s), and not to the 1x SDR (2.5 Gb/s) capacity of the switches. This means that all the results are normalized to 20 Gb/s and 12.5% of throughput achieved by the switches is in fact their full sending/receiving capability (1x SDR is 12.5% of 4x DDR). On all graphs, the throughput is presented as an average throughput per node. The second important detail is the shaded rectangles in Figure 7 that correspond to the hardware-obtainable results, i.e., the range of the results within those boxes can also be obtained with the available switches. The results beyond those boxes could not be obtained using the hardware due to limitations of the enhanced ports in real IB switches. With these two experiments we show the effect of using the same VL for both node-to-node traffic and switch-to-switch traffic and the effects of assigning these two types of traffic to different VLs.

7.2.1. First scenario. We observe that when both the node and switch traffic is mapped to the same VL, then the switch traffic does not influence the node traffic as shown in Figure 7(a), and the average achieved throughput per node decreases only by 1% when the switches send at their full capacity. There are two reasons why this happens. First, there are many more end-nodes in the network than there are switches (648 nodes and only 54 switches). The nodes are injecting more traffic into the fabric and the switch traffic suffers from contention (and possible congestion). Second, the switches have a limited number of paths they can choose from and the switch traffic is not balanced, always taking the first possible output port towards the destination, which leads to some congestion on switch-to-switch paths. Also, the subtree root is a potential bottleneck as a large part of the switch traffic is sent through it.

7.2.2. Second scenario. The situation changes when we separate the node traffic and map it to VL1 as shown in Figure 7(b). In this case, for higher loads of switch-to-switch traffic, the node traffic decreases to 64% of the capacity for the highest switch



(a) Simulation results for a 648-port fat-tree with end-node traffic in VL0.



(b) Simulation results for a 648-port fat-tree with end-node traffic in VL1.

Fig. 7. Simulation results for a 648-port switch built as fat-tree topology.

load. This happens because the traffic in the two VLs are given an equal share of the link bandwidth, artificially increasing the impact of the comparatively light switch-to-switch traffic. Clearly, care must be taken when separating management traffic in a separate VL to manage the VL arbitration weights for a minimal system impact.

7.2.3. Routing algorithm comparison. In Table III we compare various routing algorithms (sFtree, minhop, Up*/Down*, DFSSSP, LASH) at 100% load both for switch and node traffic on two different commercially available fabrics. The settings were the same as for other experiments and matched the hardware configuration. All the node-to-switch

Table III. Routing Algorithm Performance as Percentage of Throughput Per Node

Topology	sFtree	minhop	Up*/Down*	DFSSSP	LASH
648-port fat-tree switch (2-stage)	91.94%	66.98%	66.98%	85.49%	5.257%
648-port fat-tree with rack switches (3-stage)	92.93%	54.01%	54.01%	84.20%	0.8103%

and switch-to-switch links were 4x DDR and the switch generators were set to 1x SDR. We chose a 648-port fat-tree switch, which was also used for other simulations presented in this paper, and a 648-port fat-tree switch with an additional layer of rack switches attached at the bottom of it. In this case the rack switches are also interconnected horizontally to create two-switch units [Oracle Corporation 2010]. The results presented in Table III illustrate that only the sFtree algorithm is able to achieve high performance, and while DFSSSP may still be considered a viable choice, both minhop and Up*/Down* are not suitable for any of those two fat-tree topologies. Moreover, we observe that if the topology becomes more complex, the performance of all algorithms apart from sFtree deteriorates even more. It is worth noting that LASH is unsuitable for routing any type of fat-tree topology. The performance results are shown in Table III, being 5.257% and 0.8103% of average throughput per node for the 2-stage and 3-stage fat-trees, respectively. Such a low routing performance is explained by the fact that LASH does not balance the paths (like DFSSSP does) and selects the first possible shortest path towards the destination. Therefore, in the 648-port 2-stage fat-tree, out of 18 possible root switches, only one switch is used for forwarding the data packets from all 36 leaf switches and no traffic passes through the other 17 root switches. The same situations happens in the 3-stage fat-tree, but due to differences in cabling, the performance is even lower. In other words, when used for routing a fat-tree topology, LASH constructs a logical tree within the physical fat-tree fabric, which dramatically decreases the number of available links that could be used for forwarding the packets and leads to congestion. When it comes to VL usage, LASH only uses one VL for the 648-port fat-tree in Figure 5, but for a 3456-port fat-tree (3-stage fabric) LASH requires 6 VLs, and for the JuRoPa fat-tree it needs 8 VLs to ensure deadlock freedom. This means that using LASH not only leads to a decrease in routing performance, but also to a waste of valuable VL resources that could be used differently in fat-trees [Guay et al. 2011].

To summarize, the simulations confirmed what we observed in the hardware measurements. For the small loads (<10% of capacity) which are obtainable on the hardware, we see no influence of switch traffic on the end-node traffic. The simulations also confirmed that our solution is scalable and can be applied to larger topologies without negative impact on the end-node network performance. Apart from the 648-port topology presented in this paper, we simulated other topologies like a 3456-port switch (3456 nodes, 720 switches) and a JuRoPa-like supercomputer (5184 nodes, 864 switches) and the results obtained during these simulations exhibit the same trends as the results for the 648-port topology. Furthermore, for small loads, there is little difference when it comes to separating the node-to-node and switch-to-switch traffic. We also show that LASH despite having the desired properties of deadlock freedom and all-to-all communication, is not suitable for routing fat-tree topologies.

7.3. Execution Time

The execution time of the sFtree algorithm depends only on the number of switches in the network. For the largest single-core fat-tree shown in Table IV the execution overhead is around 0.5 seconds. For more complex topologies, like JuRoPa-like (864 switches) or Ranger-like (1440 switches) supercomputers, the overhead is 2.4 and 6.1 seconds respectively. For comparison, we also added the execution time results from LASH. Due to the fact that LASH does cycle search between all the possible pairs, routing takes much longer on all fabrics apart from the smallest ones. This is

Table IV. Execution Time of the OpenSM Routing in Seconds

Topology	no SW2SW	SW2SW	LASH
648-port fat-tree switch (s)	0.047	0.047	0.04
648-port fat-tree with rack switches (s)	0.104	0.112	0.27
3456-port fat-tree switch (s)	2.29	2.796	400.99
JuRoPA-like supercomputer (s)	6.43	8.858	–
Ranger-like supercomputer (s)	21.73	27.8	–

another reason why LASH should not be used for routing fat-trees because a possible rerouting takes a very long time. It is visible for a 3456-port fat-tree where LASH execution took almost 401 seconds. Unfortunately, we were not able to run LASH on topologies larger than the 3456-port fat-tree in a reasonable time.

8. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed and demonstrated the sFtree routing algorithm which provides full connectivity and deadlock-free switch-to-switch routing for fat-trees. The algorithm enables full connectivity between any of the switches in a fat-tree when using IPoIB, which is crucial for fabric management features such as the Simple Network Management Protocol (SNMP) for management and monitoring, Secure SHell (SSH) for arbitrary switch access, or generic web interface access.

We have implemented the sFtree algorithm in OpenSM for evaluation on a small IB cluster and studied the scalability of our algorithm through simulations. Through the evaluation we verified the correctness of the algorithm and showed that the overhead of the switch-to-switch communication had a negligible impact on the end-node traffic. Moreover, we were able to demonstrate that the algorithm is scalable and works well even with the largest fat-trees. Furthermore, we compared the sFtree algorithm to several topology agnostic algorithms and showed that sFtree was by far the most optimal solution for switch-to-switch connectivity in fat-trees.

REFERENCES

Bogdanski, B., Sem-Jacobsen, F. O., Reinemo, S.-A., Skeie, T., Holen, L., and Huse, L. P. 2010. Achieving predictable high performance in imbalanced fat trees. In *Proceedings of the 16th IEEE International Conference on Parallel and Distributed Systems*. X. Huang, Ed. IEEE Computer Society, 381–388.

Chu, J. and Kashyap, V. 2006. RFC 1633 transmission of IP over InfiniBand (IPoIB). IETF.

Dally, W. J. 1992. Virtual-channel flow control. *IEEE Trans. Parall. Distrib. Syst. 3,* 2, 194–205.

Domke, J., Hoefler, T., and Nagel, W. 2011. Deadlock-free oblivious routing for arbitrary topologies. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 613–624.

Duato, J., Yalamanchili, S., and Ni, L. 2003. *Interconnection Networks An Engineering Approach*. Morgan Kaufmann.

Gómez, C., Gilabert, F., Gómez, M. E., López, P., and Duato, J. 2007. Deterministic versus adaptive routing in fat-trees. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*.

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

Gran, E. G., Zahavi, E., Reinemo, S.-A., Skeie, T., Shainer, G., and Lysne, O. 2011. On the relation between congestion control, switch arbitration and fairness. In *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2011)*.

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

Guay, W. L., Reinemo, S.-A., Lysne, O., Skeie, T., Johnsen, B. D., and Holen, L. 2010. Host side dynamic reconfiguration with InfiniBand. In *Proceedings of the IEEE International Conference on Cluster Computing*. X. Gu and X. Ma, Eds. IEEE Computer Society, 126–135.

HPCC. 2011. HPC challenge benchmark. http://icl.cs.utk.edu/hpcc/.

IBTA. 2007. Infiniband architecture specification 1.2.1 Ed.

JÜLICH SUPERCOMPUTING CENTRE. 2011. FZJ-JSC JuRoPA. http://www.fz-juelich.de/jsc/juropa/configuration/.

KASHYAP, V. 2006. IP over InfiniBand: Connected mode. IETF.

LEISERSON, C. E. 1985. Fat-Trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Computers*.

LIN, X.-Y., CHUNG, Y.-C., AND HUANG, T.-Y. 2004. A multiple LID routing scheme for fat-tree-based Infiniband networks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposiums*.

MELLANOX TECHNOLOGIES. 2009. MTS3600 36-port 20 Gb/s and 40Gb/s InfiniBand switch system. Product brief. http://www.mellanox.com/related-docs/prod_ib_switch_systems/PB_MTS3600.pdf.

MELLANOX TECHNOLOGIES. 2011. IS5600—648-port InfiniBand chassis switch. http://www.mellanox.com/related-docs/prod_ib_switch_systems/IS5600.pdf.

MVAPICH2. 2011. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/overview/mvapich2/.

ÖHRING, S., IBEL, M., DAS, S., AND KUMAR, M. 1995. On generalized fat trees. In *Proceedings of the 9th International Parallel Processing Symposium*. 37–44.

OPENFABRICS ALLIANCE. 2011. The OpenFabrics Alliance. http://openfabrics.org/.

ORACLE CORPORATION. 2006. Sun Fire X2200 M2 server. http://www.sun.com/servers/x64/x2200/.

ORACLE CORPORATION. 2010. Sun Blade 6048 InfiniBand QDR switched NEM. http://www.oracle.com/us/products/servers-storage/servers/blades/031095.htm.

ORACLE CORPORATION. 2011a. Sun datacenter InfiniBand switch 36. http://www.oracle.com/us/products/servers-storage/networking/infiniband/036206.htm.

ORACLE CORPORATION. 2011b. Sun datacenter InfiniBand switch 648. http://www.oracle.com/us/products/servers-storage/networking/infiniband/034537.htm.

PETRINI, F. AND VANNESCHI, M. 1995. K-ary n-trees: High performance networks for massively parallel architectures. Tech. rep., Dipartimento di Informatica, Universita di Pisa.

QLOGIC. 2007. SilverStorm 9024 switch. http://www.qlogic.com/Resources/Documents/DataSheets/Switches/Edge_Fabric_Switches_datasheet.pdf.

REINEMO, S.-A., SKEIE, T., SØDRING, T., LYSNE, O., AND TØRUDBAKKEN, O. 2006. An overview of qos capabilities in infiniband, advanced switching interconnect, and ethernet. *IEEE Comm. Mag. 44,* 7, 32–38.

RODRIGUEZ, G., MINKENBERG, C., BEIVIDE, R., AND LUIJTEN, R. P. 2009. Oblivious routing schemes in extended generalized fat tree networks. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*.

SCHROEDER, M. D., BIRRELL, A. D., BURROWS, M., MURRAY, H., NEEDHAM, R. M., AND RODEHEFFER, T. L. 1991. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE J. Select. Areas Comm. 9,* 8.

SKEIE, T., LYSNE, O., AND THEISS, I. 2002. Layered shortest path (lash) routing in irregular system area networks. In *Proceedings of the Communication Architecture for Clusters Conference*.

TACC. 2011. Texas Advanced Computing Center. http://www.tacc.utexas.edu/.

TOP 500. 2011. Top 500 supercomputer sites. http://top500.org/.

VOLTAIRE. 2011. Voltaire QDR InfiniBand grid director 4700. http://www.voltaire.com/Products/InfiniBand/Grid_Director_Switches/Voltaire_Grid_Director_4700.

ZAHAVI, E., JOHNSON, G., KERBYSON, D. J., AND LANG, M. 2009. Optimized Infiniband fat-tree routing for shift all-to-all communication patterns. *Concurrency Computat. Pract. Exper*.

Received July 2011; revised October 2011, December 2011; accepted December 2011