Efficient Routing and Reconfiguration in Virtualized HPC Environments with vSwitch-enabled Lossless Networks

Evangelos Tasoulas*,[1] Feroz Zahid,[1,4] Ernst Gunnar Gran,[1] Kyrre Begnum,[3] Bjørn Dag Johnsen,[2] and Tor Skeie[1,4]

[1] *Simula Research Laboratory, Martin Linges Vei 25, 1364 Fornebu, Norway*
[2] *Oracle Norway, Olaf Helsets vei 6, 0694 Oslo, Norway*
[3] *Oslo and Akershus University College, Pilestredet 46, 0167 Oslo, Norway*
[4] *Department of Informatics, University of Oslo, P.O. Box 1072 Blindern, 0316 Oslo, Norway*

Correspondence: *Evangelos Tasoulas, Email: vangelis@tasoulas.net

Summary

To meet the demands of communication-intensive workloads in the cloud, virtual machines (VMs) should utilize low overhead network communication paradigms. In general, such paradigms enable VMs to directly communicate with the hardware by means of a passthrough technology like Single-Root I/O Virtualization (SR-IOV). However, when passthrough-based virtualization is coupled with lossless interconnection networks, live-migrations introduce scalability challenges due to the substantial network reconfiguration overhead. With these challenges in mind we proposed a virtual switch (vSwitch) SR-IOV architecture for InfiniBand in (33). In this paper, we first suggest solutions to rectify the space-domain scalability issues that are present in vSwitch-enabled subnets as a result of the VMs using dedicated layer-two addresses. Then we discuss routing strategies for virtualized environments using vSwitches, and present a routing algorithm for Fat-Trees. We also present a reconfiguration method that minimizes imposed reconfiguration overhead on Fat-Trees. We perform an extensive evaluation of our prototype algorithms, and as vSwitch-enabled hardware does not yet exist, we deduce from empirical observations by emulating vSwitches with existing hardware, as well as large-scale simulations. Our results show significant reduction in the reconfiguration times as route recalculations can be eliminated, and for certain scenarios, the number of reconfiguration subnet management packets sent to switches is reduced from several hundred thousand down to a single one without degrading the routing quality.

Keywords: Data centers, hardware virtualization, High Performance Computing (HPC), InfiniBand (IB), lossless networks, network reconfiguration, network routing, scalability, SR-IOV, vSwitch architecture.

1 Introduction

During the last decade, the prospect of virtualized High Performance Computing (HPC) environments has improved considerably as CPU overhead has been practically removed through hardware virtualization support; memory overhead has been significantly reduced by virtualizing the Memory Management Unit; storage overhead has been reduced by the use of fast SAN storages or distributed networked file systems; and network I/O overhead has been reduced by the use of device passthrough techniques like Single Root Input/Output Virtualization (SR-IOV) (16). It is now possible for clouds to accommodate virtual HPC (vHPC) clusters using high performance lossless interconnect solutions and deliver the necessary performance (14, 15, 21). However, when virtual machines (VMs) are coupled with lossless networks, such as InfiniBand (IB) (1), necessary cloud functionality like live migration of VMs remains an issue due to the complicated addressing and routing schemes used in lossless networks.

IB is an interconnection network technology offering high bandwidth and low latency, thus, is very well suited for HPC and other communication intensive workloads. IB accelerates 37.4% of the systems in the TOP500 supercomputers list as of November 2016 (32). The traditional approach for connecting IB devices to VMs is by utilizing SR-IOV with direct assignment, but achieving live migration of VMs assigned with IB Host Channel Adapters (HCAs) using SR-IOV has shown to be challenging (13, 14, 21, 34). The identified challenges can be grouped into two categories: challenges related to the extended migration downtime due to the direct device assignment, and challenges related to the scalability and operation of the underlying network. In this work we focus on the second group of challenges and address issues from the perspective of the network. Note that although IB and IB terminology is used throughout this work, the presented concepts and challenges arise from the lossless nature and operating principles of the underlying network and apply even in competing technologies[1].

Each IB connected node has three different addresses: LID, GUID and GID (further discussed in section 2). When a live migration happens, one or more of these addresses change. Other nodes communicating with the VM-in-migration lose connectivity and try to find the new address to reconnect to by sending Subnet Administration (SA) path record queries to the IB Subnet Manager (SM). In (34) we showed that multiple concurrent path record queries can challenge the SM, and by using *address caching*, one does not have to send repetitive SA queries to reconnect once a VM is live migrated. However, in order to allow a VM to be moved and benefit from such a caching mechanism, each VM should be bound to a dedicated set of IB addresses that can migrate together with the VM. With the current IB SR-IOV *Shared Port* implementation (22), the VMs running on the same *hypervisor* share one LID address and have dedicated GUID and GID addresses. When a VM with its associated LID is migrated, the connectivity will be broken for the rest of the VMs that share the same LID. In (33) we proposed two implementations of the *Virtual Switch (vSwitch)* architecture, that allow IB subnets to support transparent virtualization and migration of all IB addresses, accompanied with a scalable and topology-agnostic dynamic network reconfiguration method to make live migrations of VMs feasible in large vSwitch-based IB subnets. However, the vSwitches come at the cost of space-domain scalability issues due to the limited LID space, an issue which is a limitation imposed by the IB specification. The space-domain scalability issues were left unaddressed in (33). Moreover, the reconfiguration algorithm presented in (33) (we call it *ItRC* in the rest of the paper) made live migrations feasible, but it was suboptimal.

[1]Intel Omni-Path (5) and Bull eXascale Interconnect (8) are two such competing technologies that share similar concerns.

In this paper, we extend and complement our work in (33) by addressing the space-domain scalability challenges as well as optimizing the reconfiguration algorithm to reduce the number of reconfigured switches for any reconfiguration scenario. Furthermore, in this work, we present the first generalized *Fat-Tree* (17) topology-aware routing algorithm which has been designed for virtualized subnets that are based on vSwitches; a topic that we did not target in our previous work.

The rest of the paper is organized as follows: section 2 presents background information on the IB SR-IOV design, IB addressing schemes and Fat-Tree topologies, followed by the related work in section 3. In section 4 we give a brief of our previously proposed vSwitch architecture and topology-agnostic dynamic reconfiguration mechanism (*ItRC*). In the same section we extend our vSwitch architecture by proposing space-domain scalability improvements. A discussion of routing strategies for vSwitch-based subnets, and the presentation of a Fat-Tree topology-aware routing algorithm is following in section 5, and a Fat-Tree topology-aware method to minimize the path distribution phase of our reconfiguration mechanism is presented in section 6. We evaluate our prototype implementations in section 7, before we conclude in section 8.

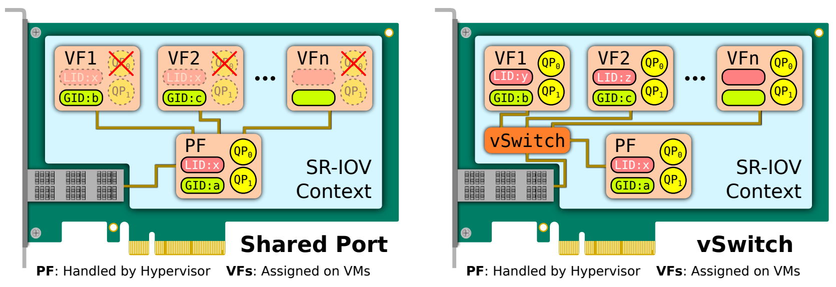
2 Background

2.1 The Relationship of HPC, SR-IOV and Live VM Migration

Direct assignment, or *device passthrough*, provides near to native performance with minimum overhead on VMs. The physical device bypasses the hypervisor and is directly attached to the VM. Currently, this is the only option for using a physical interconnect in a virtualized HPC environment as opposed to software emulation, which does not offer the same level of performance. The downside is limited scalability, as there is no sharing; one physical network card is coupled with one VM.

SR-IOV allows a physical device to appear through hardware virtualization as multiple independent lightweight instances of the same device. These instances can be assigned to VMs as passthrough devices, and accessed as Virtual Functions (VFs) (16). The hypervisor accesses the device through a unique, per device, fully featured Physical Function (PF). SR-IOV eases the scalability issue of pure direct assignment. At the moment, there is no efficient way to live-migrate VMs without a network downtime in the order of seconds when using direct device assignment (9). With SR-IOV VFs, the complete internal state of the network interface cannot be copied as it is tied to the hardware (21). The VFs assigned to a VM will need to be detached, the live migration will run, and a new VF will be attached at the destination.

In the case of IB and SR-IOV, this process will introduce downtime in the order of seconds as discussed by Guay et al. (11–13), and is one aspect of the existing issues for enabling efficient live-migration of VMs coupled with lossless networks. Another aspect is that with the currently implemented SR-IOV *Shared Port* model the addresses of the VM will change after the migration, causing additional overhead in the SM and a negative impact on the scalable management and performance of the network fabric (34). The latter aspect takes the perspective of the interconnect fabric, and is the primary focus of this paper.



(a) (b)

Fig. 1: Proposed InfiniBand SR-IOV architectures.

2.2 InfiniBand Addressing Schemes

IB uses three different types of addresses (1, 13, 26). First is the 16 bits Local Identifier (LID). At least one unique LID is assigned to each HCA port and each switch by the SM. The LIDs are used to route traffic within a subnet. Since the LID is 16 bits long, 65536 unique address combinations can be made, of which only 49151 (0x0001-0xBFFF) can be used as unicast addresses. Consequently, the number of available unicast addresses defines the maximum size of an IB subnet. Second is the 64 bits Global Unique Identifier (GUID) assigned by the manufacturer to each device (e.g. HCAs and switches) and each HCA port. The SM may assign additional subnet unique GUIDs to an HCA port, which is particularly useful when SR-IOV is used. Third is the 128 bits Global Identifier (GID). The GID is a valid IPv6 unicast address, and at least one is assigned to each HCA port and each switch. The GID is formed by combining a globally unique 64 bits prefix assigned by the fabric administrator, and the GUID address of each HCA port.

2.3 InfiniBand SR-IOV Architectures

The *Shared Port* architecture, illustrated in Fig. 1(a), is currently the only SR-IOV implementation for IB (22). The Shared Port architecture comes with a shared LID and Queue Pair (QP)[2] space resulting in two main shortcomings: 1) VFs are not configurable by the SM, meaning that features such as knowing the port state of the VFs or configuring individual routes per VF are unavailable. 2) The inability to provide transparent live migration, hindering the potential for flexible VM placement. As each LID is associated with a specific hypervisor and shared among all VMs residing on the hypervisor, a migrating VM will have its LID changed to the LID of the destination hypervisor. As shown at (34), this LID change can seriously challenge the SM and the scalability of the subnet in the performance-domain.

The *vSwitch* architecture was suggested by Liss in 2010 (18) with certain benefits with regards to live migration when compared to the *Shared Port*. In the vSwitch architecture (Fig. 1(b)) each VF is a complete vHCA, meaning that the VM is assigned a complete set of the three IB addresses and a dedicated QP space in the hardware. For the rest of the network and the SM, the HCA looks like a switch with additional nodes connected to it; the hypervisor uses the PF and the VMs use the VFs, as shown in Fig. 1(b). The vSwitch architecture solves the two shortcomings of the Shared Port architecture and provides transparent

[2]A QP is a virtual communication port used by IB apps to communicate (1)

virtualization, but at the cost of consuming additional LID addresses. When many LID addresses are in use, more communication paths have to be handled by the SM and more Subnet Management Packets (SMPs) have to be sent to the switches in order to update their Linear Forwarding Tables (LFTs). In particular, the computation of the communication paths might take several minutes in large networks as we show in the results section 7.1. Moreover, as each VM, physical node, and switch occupies one LID each, the number of physical nodes and switches in the network limits the number of active VMs, and vice versa, leading to potential scalability challenges in the space-domain. Recall that the IB LID space is limited to 49151 unicast LIDs.

An alternative *vPort* concept has been standardized in 2016 (2). The vPort is loosely defined in order to give freedom of implementation to vendors (e.g. the definition does not rule that the implementation has to be SR-IOV specific), and the goal of the vPort is to standardize the way VMs should be handled in IB subnets. With the vPort concept both SR-IOV *Shared-Port*-like and *vSwitch*-like architectures or a combination of both, that can be more scalable in both the space and performance domains, can be defined. A vPort supports optional LIDs, and unlike the *Shared-Port*, the SM is aware of all the vPorts available in a subnet even if a vPort is not using a dedicated LID.

2.4 The Fat-Tree Topology

The Fat-Tree is a hierarchical network topology that has been shown to scale with the available network resources (23, 25). Moreover, Fat-Trees are easy to build using commodity switches placed on different levels of the hierarchy (3). The idea behind Fat-Trees is to employ *fatter* links between nodes, with more available bandwidth towards the roots of the topology. The fatter links help to avoid congestion in the upper-level switches and the bisection bandwidth is maintained. Different variations of Fat-Trees are presented in the literature, including *k*-ary-*n*-trees (25), Extended Generalized Fat-Trees (XGFTs) (23), Parallel Ports Generalized Fat-Trees (PGFTs) and Real Life Fat-Trees (RLFTs) (37).

A *k*-ary-*n*-tree (25) is an n level Fat-Tree with k^n end nodes and $n \cdot k^{n-1}$ switches, each with $2k$ ports. Each switch has an equal number of up and down connections in the tree. XGFT Fat-Tree extends *k*-ary-*n*-trees by allowing both different number of up and down connections for the switches, and different number of connections at each level in the tree. The PGFT definition further broadens the XGFT topologies and permits multiple connections between switches. A large variety of topologies can be defined using XGFTs and PGFTs. However, for practical purposes, RLFT, which is a restricted version of PGFT, is introduced to define Fat-Trees commonly found in today's HPC clusters (38). An RLFT uses the same port-count switches at all levels in the Fat-Tree.

3 Related Work

Different works in the context of virtualized HPC clouds with lossless interconnects are mostly limited to pointing out existing issues and demonstrating that the technology has matured enough and virtualization overheads sufficiently reduced to make such clouds feasible (14, 15, 21, 27). Others have focused in cloud tenant isolation in the link level (40, 43) while Ranadive, in his doctoral dissertation dedicated his efforts on virtualized resource management with lossless networks (28). Our work focuses in the SM scalability and the limitations imposed by lossless network architectures, and in particular the IB architecture, in the context of virtualized environments that support VM live migrations.

In the context of live VM migration, Guay et al. (13) demonstrate VM migration with IB VFs. The vGUID of the SR-IOV VF is migrated together with the VM, but the LID changes. The main goal is to reestablish broken communications after a VM has been migrated and the LID has changed, with the intention to avoid reconfiguring the network. However, additional signaling and overhead towards the SM is required due to the LID change. In (34) we migrate VMs with IB VFs while keeping all three addresses. A caching mechanism is used to reestablish connectivity and demonstrate that as long as all three addresses can be migrated, no additional communication overhead, such as sending SA PathRecord queries, is needed. This is a key finding that drove our design of the IB vSwitch architecture (33), an architecture that we complement with the work presented in the current paper. In (12) the authors present VM migration with IB VFs and reestablish connectivity without interrupting the communications. The scope of (12) is to reduce migration downtime, unlike the works in (13, 33, 34) as well as the work in the current paper, that are focused in the reconfiguration of the network and scalability of the SM.

In general, when a lossless network is reconfigured, routes have to be recalculated and distributed to all switches, while avoiding deadlocks. Note that the coexistence of two deadlock free routing functions, the R_{old} and R_{new}, during the transition phase from the old to the new one, might not be deadlock free (10). Zafar et al. (36) discusses the tools and applicable methods on IB architecture (IBA), that would allow the implementation of the *Double Scheme* (24) reconfiguration method. The *Double Scheme* is using Virtual Lanes (VLs) to separate the new and the old routing functions. Lysne et al. (20) use a token that is propagated through the network to mark a reconfiguration event. Before the token arrives on a switch, traffic is routed with the old routing algorithm. After the token arrives and forwarded through the output ports of the switch, the traffic is flowing with the new routing algorithm. The *Skyline* by Lysne et al. (19), speeds up the reconfiguration process by providing a method for identifying the minimum part of the network that needs to be reconfigured. Sem-Jacobsen et al. (31) use the channel dependency graph to create a channel list that is rearranged when traffic needs to be rerouted. The rearranging is happening in such a way, that no deadlocks can occur. Robles-Gómez et al. (29) use close up*/down* graphs to compute a new routing algorithm which is close to the old one, and guarantees that the combination of old and new routing during transition do not allow deadlocks to be introduced. Bermúdez et al. (4) are concerned with the long time it takes to compute optimal routing tables in large networks, that consequently delays the IB subnet from becoming operational. They use some quickly calculated but not optimal provisional routes, and calculate the optimal routes *offline*. Since the provisional and the optimal routes are calculated based on the same acyclic graph, deadlock freedom is guaranteed. In our previous work at (41), we presented a metabase-driven fast network reconfiguration scheme for Fat-Trees. In the proposed method, routing is divided into two stages; In the first stage, the routes in the topology are calculated, which are later assigned to the actual destinations in the second stage. The reconfiguration method, however, targets only performance-driven routing updates where no change in the topology is assumed during reconfigurations. The work in (35) is concerned about the dynamic nature of clouds where workloads constantly change or VMs live migrate and

	SR-IOV vSwitch Implementation	
	Prepopulated LIDs	**Dynamic LID Assignment**
Cost at network Initialization (Lower is better)	***Medium-High***: Takes more time to configure the network when booting since paths have to be computed and distributed even if no VMs are running.	***Medium***: Takes into account only the physical infrastructure and the number of active VMs.
Cost when booting VMs (Lower is better)	***None***: Paths towards the newly booted VMs already exist in the LFTs of the switches	***Medium***: Each time a VM is booted the paths of all the LFTs of all the physical switches have to be updated.
Flexibility with regards to the LID space (Higher is better)	***None***: The LID space rules the max number of available VFs in the subnet. When operating close to the unicast LID limit, we have less live-migration options for optimization.	***High***: VMs can be migrated anywhere in the network where VFs are available even when the network operates close to the unicast LID limit.
Reconfiguration cost for a Live Migration (Lower is better)	***Low-Medium***: If the migrated VM connects with a VF for which the LID is contained in the same LFT block as the one that the LID currently assigned to the VM is contained into, one LFT update SMP has to be sent to the switches chosen to be reconfigured. Otherwise, twice as many SMPs will be sent.	***Low***: Only one LFT update SMP has to be sent to the switches chosen to be reconfigured.
Reconfiguration cost formulas (33)	$vSwitch\_RC_t = n' \cdot m' \cdot k$ ($m' = 2$ if the two LID entries are not located in the same LFT block when the LIDs are prepopulated, otherwise $m' = 1$)	$vSwitch\_RC_t = n' \cdot k$

Table 1: Summary of the Proposed vSwitch architecture. Refer to (33) for more details and formula explanations.

the infrastructure is in continuous need for performance driven reconfigurations. A solution that reduces the reconfiguration overhead by allowing for fast partial-reconfigurations is presented. In the current paper we present a reconfiguration algorithm for vSwitch-enabled subnets, that focuses on quick reconfiguration of the network when VMs migrate. The intention of our reconfiguration mechanism is to reestablish VM connectivity as quickly as possible while imposing minimal overhead and disturbance on the SM and the network by reconfiguring as few switches as possible.

4 Proposed vSwitch Architecture

In (33), we proposed two alternative implementations of the vSwitch architecture with different scalability characteristics, the *vSwitch with Prepopulated LIDs* and the *vSwitch with Dynamic LID assignment*. Furthermore, we provided an iterative reconfiguration method, *ItRC*[3], for scalable and topology-agnostic dynamic reconfiguration as VMs are live migrated. In this section we provide an brief overview of the vSwitch architecture.

The *vSwitch with Prepopulated LIDs* initializes all available VFs with LIDs, even those VFs that are not currently used by any VM. In a vSwitch architecture with prepopulated LIDs, communication paths are computed and distributed for all the LIDs once, when the network is booted. When a new VM needs to be started the system does not have to perform additional bookkeeping, hence, there is no additional overhead. On the negative side, the initial handling of the paths will require more time than what it would need without the prepopulation of all LIDs, and if all of the VFs are occupied by running VMs, there is no option for optimizations by using live migrations.

The *vSwitch with Dynamic LID assignment* initializes the VFs when a new VM is booted. Since we know that all the VFs in a hypervisor share the same uplink with the PF, there is no need to compute a new set of routes, which is the most time-consuming

[3]The iterative reconfiguration method has not been given a name in paper (33) where it is originally discussed, but we now call it *ItRC*, in order to differentiate it from *FTreeMinRC* that we introduce in section 6.

step in a reconfiguration phase. It is only needed to iterate through the LFTs of all the physical switches in the network, copy the forwarding port from the LID entry that belongs to the PF of the hypervisor —where the VM is created— to the newly added LID, and send a single SMP to update the corresponding LFT block of the particular switch. This scheme is more flexible, especially when operating close to the LID space limit, at the cost of a slight overhead when a new VM is booting.

When a VM is migrated and carries its addresses to the destination, a network reconfiguration is necessary. However, the migration of the LID is not trivial, because the routes have to be recalculated and the LFTs of the physical switches reconfigured. In general, recalculation of routes needs a considerable amount of time in the order of minutes on large subnets, posing scalability challenges that may render VM migrations impractical. The vSwitch, however, has the property that all the VFs of an HCA share the link with the PF. Our topology-agnostic dynamic reconfiguration mechanism, *ItRC*, utilizes this property and copies forwarding entries in a novel way to make the reconfiguration highly efficient in dynamic environments. The LID reconfiguration time is minimized by eliminating the path computation, and drastically reducing the path distribution phase.

A summary of these two alternative architectures and the reconfiguration cost is presented in Table 1, and readers are referred to (33) for additional details and formula explanations.

4.1 vSwitch Scalability

A main concern when using the IB vSwitch architecture is the limited LID space that defines that max number of nodes as explained in section 2.2, and when VMs are used, the number of nodes can escalate quickly. To overcome the LID space-domain scalability issues, we propose three alternatives that can be used independently or combined:

Use multiple IB subnets The LID is a layer-2 address, and has to be unique within a subnet. When the IB topology spans on multiple subnets, the LID is not a limitation anymore, but if a VM needs to be migrated to a different subnet its LID address will have to change since the address may already be in use in the new subnet. Spanning on multiple subnets can solve the LID limitation of a single subnet topology, but it also means that the layer-3 GID address must be used for inter-subnet routing, adding additional overhead and latency to the routing process since the layer-2 headers will have to be altered by the router that is located at the edge of the subnet. Also, under the current hardware, software implementations, and loose IBA specifications, the SM of an individual subnet cannot be aware of the global topology in order to provide optimized routing paths for clusters that span on multiple subnets. In a typical cloud offering scenario the size of a single subnet should be enough to host VMs from a single tenant in most cases, and tenant consolidation within a subnet should be engaged. IB subnetting and routing has been discussed by Bogdański et al. (6), but the exploration in this field is still at early stages and there is plenty of ground for future work. The reconfiguration and routing methods that are discussed and analyzed in the next sections in this paper are only taking into account single-subnet live migration scenarios.

Introduce a backwards compatible LID space extension in IBA Increasing the scarce LID space by increasing the number of LID bits to e.g. 24-bits or 32-bits has already been suggested (6, 11). By doing so, backwards compatibility will break as the

IB Local Route Header (LRH) will have to be overhauled, and legacy hardware will not be able to work with the new standard. What we propose is to extend the LID space in such a way that backwards compatibility is still kept, but allow new hardware to take advantage of the enhancements. Currently, the LRH has seven reserved bits that are transmitted as *zero*, and ignored on the receiver (1). By utilizing two of these reserved bits in the LRH for the Source LID (SLID) and two bits for the Destination LID (DLID), utilizing 4/7 reserved bits in total[4], we can extend the LID space to 18 bits (quadrupling the LID space) and create a scheme with *physical LIDs (pLID)* that are assigned to the physical equipment, and *virtual LIDs (vLID)* that are assigned to VMs. When the two additional bits are transmitted as zero, the LID is used as defined currently in IBA (48K unicast LIDs and 16K multicast LIDs), and the switches lookup their primary LFT for the forwarding of the packets. Otherwise, the LID is a vLID that belongs to a VM only and it has to be forwarded based on the secondary LFT that has a size of 192K. No multicast LIDs are reserved in the vLID space, thus, with the suggested scheme a maximum of 192K vSwitch-enabled VMs per subnet are allowed. Note that the vLID space is independent of the pLID space, and the vLIDs can be allocated to any physical machine. Since the vLIDs belong to VMs and VMs share the uplink with a physical node that has a pLID, the vLIDs can be excluded from the path computation phase when (re-)configuring the network, and the secondary LFT table in the switches should be updated as described earlier in the *vSwitch with dynamic LID assignment* scheme. When the SM boots and discovers the network, it should identify if all of the hardware supports the extended LID space. If not, it should fallback in a *legacy compatibility* mode and VMs should occupy LIDs from the pLID space. In the legacy compatibility mode, the number of VMs is limited as explained in (33).

Use a hybrid vSwitch architecture to form a lightweight-vSwitch A vSwitch architecture that is able to migrate the LID together with the migrated VMs scales well with respect to the subnet management, as there is no requirement for additional signaling in order to re-establish connectivity with the peers after the migration (34) as opposed to a shared-LID scheme where the LID will change (13). On the other hand, the shared-LID schemes scale well with respect to the LID space. As a result, we propose a hybrid *vSwitch+Shared-vPort* model that the SM is aware of all the available SR-IOV VFs in the subnet, but certain VFs get a dedicated LID while others are routed in a shared-LID fashion based on their GID. Then, with some knowledge of the VM-node role, *popular* VMs with many peers (usually servers) can be assigned dedicated LIDs (e.g. in order to be considered separately while calculating routes and performing load balancing in the network), while other VMs that are not interacting with many peers or run stateless services (and do not need to be migrated, but can be re-spawned) can share the LID.

5 Routing Strategies for vSwitch-based Subnets

To obtain optimal performance, the routing algorithm should consider the vSwitch architecture when calculating routes. The vSwitches can be identified in the topology discovery process by the distinct property of having only one upward link to the corresponding switch. Once the vSwitches have been identified, a routing function can generate LFTs for all the switches such

[4]A maximum of three additional bits could be used for the SLID and DLID (occupying 6/7 reserved bits), but we propose to use two bits in order to keep some reserved bits for potential future use. If three additional bits are used, then the total LID space (pLIDs + vLIDs) would become 512K.

that the traffic can find its path towards all VMs in the network. In our proposed vSwitch architecture each VM has its own address, thus, technically each VM can be routed independently of other VMs attached to the same vSwitch. One drawback of this approach is that when the VM distribution is not uniform among vSwitches, the vSwitches with more VMs are potentially assigned more network resources. However, the single upward link from the vSwitch to the corresponding switch still remains the bottleneck link shared by all the VMs attached to a particular vSwitch. As a result, sub-optimal network utilization may be obtained. The simplest and fastest routing strategy is to generate paths between all vSwitch-vSwitch pairs, and route VMs with the same paths as assigned to the corresponding vSwitches. With both prepopulated and dynamic LID assignment schemes, as described in section 4, each vSwitch has a LID defined by the SR-IOV PF. These PF LIDs can be used to generate LFTs in the first phase of the routing, while in the second phase the LIDs of the VMs can be added to the generated LFTs. In the prepopulated LIDs scheme, the entries to the VF LIDs are added by just copying the output port of the corresponding vSwitch. Similarly, in the case of dynamic LID assignment when a new VM is booted, a new entry with the LID of the VM and the output port determined by the corresponding vSwitch is added in all LFTs. The problem with this strategy is that VMs belonging to different tenants that happen to share a vSwitch will have intrinsic interference among them, due to the sharing of the same complete path in the network. To solve the aforementioned issues while still keeping high network utilization, we propose a *weighted* routing scheme for virtualized subnets.

5.1 Weighted Routing for Virtualized Subnets

In our proposed weighted routing scheme for vSwitch-based virtualized subnets, each VM on a vSwitch is assigned a *weight* parameter to be considered for balancing when calculating routes. The value of the *weight* reflects the proportion of the vSwitch to leaf switch link capacity allocated to a VM. For example, a simple configuration could assign each VM a weight equals to $1/num\_vms$, where num_vms is the number of booted VMs on the corresponding vSwitch hypervisor. Another possible implementation could be to assign higher proportion of the vSwitch capacity to most critical VMs for prioritizing the traffic towards them. However, the cumulative weight of VMs per vSwitch will be equal on all vSwitches, so the links in the topology can be balanced without being affected by the actual VM distribution. At the same time, the scheme enables multipath routing where each VM can be independently routed in the network, eliminating interference between same vSwitch VMs at the intermediate links in the topology. The scheme can be combined with per VM rate limits enforcement on each vSwitch to ensure that a VM is not allowed to exceed its allocated capacity. In addition, in the presence of multiple tenant groups in the network, techniques like tenant-aware routing can be integrated with the proposed routing scheme to provide network-wide isolation among tenants (40, 43).

5.2 Weighted Fat-Tree routing algorithm for vSwitch

We now present *vSwitchFatTree*, an implementation of our proposed weighted routing for virtualized vSwitch-based Fat-Tree topologies. vSwitchFatTree is based on the Fat-Tree routing algorithm found in OpenSM (39). As the original Fat-Tree routing

Algorithm 1 vSwitchFatTree routing algorithm

```
procedure RouteVirtualizedNodes
    for all $s \in leafSwitches[]$ do
        sort vSwitches in the increasing order of connected virtual machines
        for all $v \in vSwitches[]$ do
            $num\_vms \leftarrow$ GetTotalVMs($v$)
            $vm\_weight \leftarrow 1/num\_vms$
            for all $vm \in vSwitches[]$ do
                $vm.weight \leftarrow vm\_weight$
                $s.LFT[vm.LID] \leftarrow v.port$
                RouteDowngoingByGoingUp($s$,$vm$)
            end for
        end for
    end for
end procedure
procedure RouteDowngoingByGoingUp($s$, $vm$)
    $p \leftarrow$ GetLeastLoadedPort($s.UpGroups[]$)
    $rSwitch \leftarrow p.Switch$
    $rSwitch.LFT[vm.LID] \leftarrow p$
    $p.Dwn$ += $vm.weight$
    RouteUpgoingByGoingDown($s$,$vm$)
    RouteDowngoingByGoingUp($rSwitch$,$vm$)
end procedure
procedure RouteUpgoingByGoingDown($s$, $vm$)
    for all $g \in s.DownGroups[]$ do
        skip $g$ if the LFT($vm.LID$) is part of this group
        $p \leftarrow$ GetLeastLoadedPort($g$)
        $rSwitch \leftarrow p.Switch$
        $rSwitch.LFT[vm.LID] \leftarrow p$
        $p.Up$ += $vm.weight$
        RouteUpgoingByGoingDown($rSwitch$, $vm$)
    end for
end procedure
```

algorithm, vSwitchFatTree recursively traverses the topology to set up LFTs in all switches for the LIDs associated with each VM in the subnet. The algorithm is deterministic and supports destination-based routing in which all routes are calculated backwards starting at the destination nodes.

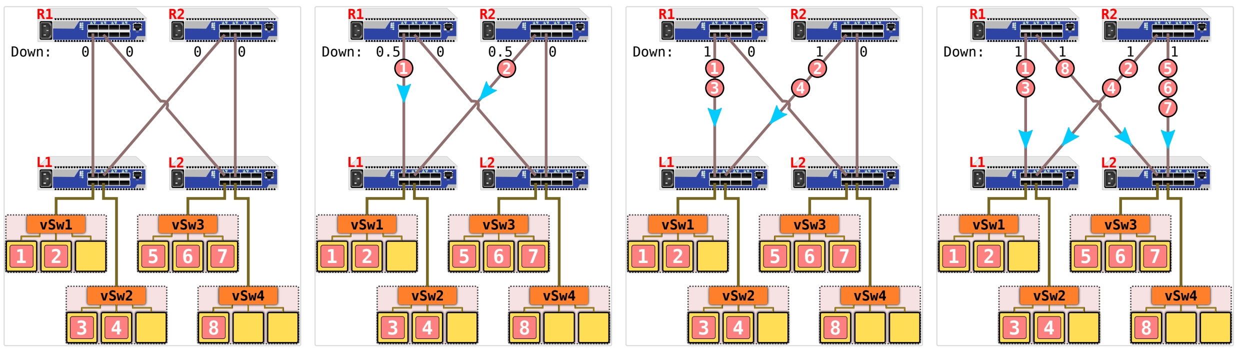
The vSwitchFatTree routing algorithm works as follows. Each VM is assigned a proportional $weight$ that is calculated by dividing the weight of a vSwitch node (taken as constant 1) with the total number of running VMs on it. Different weighting schemes can also be implemented. For instance, an implementation can choose to assign weights based on VM types. However, for the sake of brevity we only discuss the proportional weighting scheme. The pseudo-code of vSwitchFatTree is shown in Algorithm 1. For each leaf switch, the routing algorithm sorts the connected vSwitches in increasing order based on the number of connected VMs (line 3). The order is to ensure that VMs with higher weights are routed first, so that the routes assigned to the links can be balanced. The algorithm passes through all the leaf switches and their corresponding vSwitches, traversing up in the tree from each VM to allocate the path towards the VM in the tree recursively, by calling

ROUTEDOWNGOINGBYGOINGUP (line 10). The down-going port at each switch is selected based on the least-accumulated downward weight among all of the available up-going port groups (ROUTEDOWNGOINGBYGOINGUP, line 16). When a down-going port is selected, the algorithm increases the accumulated downward weight for the corresponding port by the $weight$ of the VM being routed (ROUTEDOWNGOINGBYGOINGUP, line 19). After a down-going port is set, the algorithm assigns upward ports for routes towards the VM (and updates the corresponding upward weights for the ports) on all the connected downward switches by descending down the tree (ROUTEUPGOINGBYGOINGDOWN, line 20). The process is then repeated by moving up to the next level in the tree. When all VMs have been routed, the algorithm also routes the physical LIDs of the vSwitches the same way as the VMs, albeit with equal weights to balance vSwitch to vSwitch paths in the topology (not shown in Algorithm 1). This is necessary to provide better balancing when our proposed minimum reconfiguration method (presented in section 6) is used in the context of live migrations. Also, the routing path on the base physical LIDs of the vSwitches can be used as a *pre-determined path* to deploy new VMs quickly without the need for a reconfiguration. However, over a period of time overall routing performance will be slightly decreased over original vSwitchFatTree routing. To limit performance degradation, a reconfiguration based on vSwitchFatTree could take place offline when a certain performance threshold is crossed.

The vSwitchFatTree routing algorithm enables weighted routing for virtualized Fat-Tree topologies using three important improvements:

(i) Unlike the original Fat-Tree routing algorithm which does not consider the vSwitches or VMs in the topology, vSwitchFatTree marks vSwitches, and routes each VM independently of the other VMs connected to a vSwitch.
(ii) To cater non-uniform VM distribution among the vSwitches, each VM is assigned a weight that corresponds to the proportion of the link it is allocated on the vSwitch. The weight is used in maintaining port counters for balancing path distribution in the Fat-Tree.
(iii) The scheme also enables generalized weighted Fat-Tree routing where each VM can be assigned a weight based on its traffic profile or role priority in the network (42).

Consider a virtualized Fat-Tree topology with four end nodes (vSwitches), as shown in Fig. 2(a). Each of the vSwitches connected to the leaf switch *L1*, *vSw1* and *vSw2*, have two running VMs shown with their identifier in the figure. The second leaf switch, *L2*, has *vSw3* with three VMs, while one VM is running on the host vSwitch *vSw4*. Each leaf switch is connected to both root switches, *R1* and *R2*, so there are two alternative paths available to set up routes towards each VM through the roots. Routing for the VMs connected to *vSw1* is shown in Fig. 2(b) using circles showing the selected downward path from the root switches. VM *1* is routed using $R1 \rightarrow L1$, while VM *2* is routed from $R2 \rightarrow L1$. The corresponding downward load counters are updated on the selected links, adding 0.5 for each VM. Similarly, as shown in Fig. 2(c), after adding routes for *vSw2*, VMs *3* and *4* are routed through links $R1 \rightarrow L1$ and $R2 \rightarrow L1$, respectively. Note that after routing all the VMs connected to *L1*, the total downward *load* on both links is equal, even though the VMs are routed individually. The VM distribution on the vSwitches connected to the leaf switch *L2* is different, so the vSwitch with one VM, *vSw4*, will be routed first. The route $R1 \rightarrow L2$ will



(a)
Begin routing. Load counters = 0

(b)
Routing VMs connected to vSw1

(c)
Routing VMs connected to vSw2

(d)
Final routing

Fig. 2: Port selection routing phases in vSwitchFatTree routing.

be allocated to VM 8, while all three VMs connected to *vSw3* will be routed from $R2 \rightarrow L2$ to keep the accumulated load on both downgoing links balanced. The final routing, shown in Fig. 2(d), has balanced load on each of the links together with independent routes towards VMs wherever possible, given the VM distribution in the topology.

6 Minimum Overhead Reconfiguration on Virtual Machine Live Migrations

The dynamic reconfiguration method discussed in section 4, *ItRC*, iterates through all of the switches and updates the routes if necessary when a VM is migrated. However, depending on the existing LFTs, only a subset of the switches actually needs to be updated. For instance, a special case is the migration of a VM within a leaf switch where regardless of the network topology, only the corresponding leaf switch needs an LFT update. Refer to Fig. 3, when *VM3* migrates from *Hypervisor 1* to *Hypervisor 2* only the leaf switch *1* needs to be updated because both hypervisors are connected to the same leaf switch, and the local changes will not affect the rest of the network. Consider an initial routing algorithm determining that the traffic from *Hypervisor 4*, towards *Hypervisor 1* follows path $P1$ ($12 \rightarrow 9 \rightarrow 5 \rightarrow 3 \rightarrow 1$), and towards *Hypervisor 2* follows path $P2$ ($12 \rightarrow 10 \rightarrow 6 \rightarrow 4 \rightarrow 1$). When *VM3* is migrated and ItRC is used to reconfigure the network, traffic towards *VM3* follows $P1$ towards *Hypervisor 1* before the migration, and would follow $P2$ towards *Hypervisor 2* after the migration. In this case, ItRC will update half of the total switches ($6/12$) given that the Fat-Tree routing algorithm was used for the initial routing, however, only the single leaf switch needs to be updated to keep the migrated VM connected. By limiting the number of switch updates on VM migration, the network can be reconfigured quicker and the routing update overhead reduced. In this section, we present a topology-aware reconfiguration method for supporting VM migrations on Fat-Trees, *FTreeMinRC*, based on the skyline technique described in (19).

6.1 Sub-Trees and Switch Tuples in Fat-Trees

In the following, we describe our minimum overhead network reconfiguration method, FTreeMinRC, using XGFTs as an example Fat-Tree network. However, the concepts presented here are also valid for PGFTs and RLFTs (38). We start with the definition of $XGFT$ as given by (23), but we use a slightly different notation to match the tuple assignment notation in OpenSM where

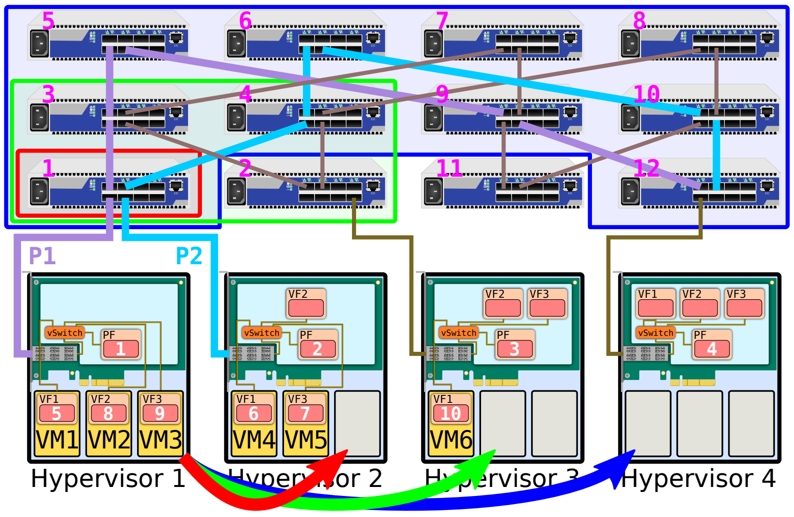


Fig. 3: vSwitch - LFTs Update on Limited Switches. Matching color of arrow and box highlighting switches, illustrates how many switches need to be updated on a minimum reconfiguration based on the skyline technique when a VM is live migrated in different parts of the network.

the compute nodes are marked at level n and the root nodes are marked at level 0. An $XGFT(n; m_1, ..., m_n; w_1, ..., w_n)$ is a Fat-Tree with $n+1$ level of nodes. Levels are denoted from 0 to n, with compute nodes at level n, and switches at all other levels. Except for the compute nodes which do not have children, all nodes at level i, $0 \leq i \leq n-1$, have m_i child nodes. Similarly, except for the root switches which do not have parents, all other nodes at level i, $1 \leq i \leq n$, have w_{i+1} parent nodes.

An $XGFT(n+1; m_1, ..., m_{n+1}; w_1, ..., w_{n+1})$ is constructed recursively by connecting m_n distinct copies of the $XGFT(n; m_1, ..., m_n; w_1, ..., w_n)$ with $\prod_{i=0}^{n+1} w_i$ additional switches at the new top level. By using this definition we can identify the following two properties and one definition:

Property 1. *For* $\mathrm{n} > 0$*, each XGFT with* $\mathrm{n}+1$ *levels is made up of* $\mathrm{m_n}$ *sub-trees. The same statement from a different angle is that for each sub-tree with* n *levels in an XGFT of* l *levels,* $\mathrm{l} > \mathrm{n}$*, there is one immediate super-tree with* $\mathrm{n}+1$ *levels, that connects* $\mathrm{m_n}$ n*-level sub-trees.*

Property 2. *From a network connectivity perspective, each sub-tree in an XGFT can be considered as a distinct XGFT, and the top-level switches in the sub-tree defines its skyline towards its immediate super-tree.*

Definition 1. *Each switch in an XGFT with* $\mathrm{n}+1$ *levels can be denoted by a unique n-tuple,* $(\mathrm{l}, \mathrm{x_1}, \mathrm{x_2}, ..., \mathrm{x_n})$*. The left most tuple value,* l*, denotes the level at which the tree is located, while the rest of the values,* $\mathrm{x_1}, \mathrm{x_2}, ..., \mathrm{x_n}$*, represent the location of the switch in the tree corresponding to the other switches. In particular, a switch A at level* l*,* $(\mathrm{l}, \mathrm{a_1}, ..., \mathrm{a_l}, ..., \mathrm{a_n})$*, is connected to a switch B at level* $\mathrm{l}+1$*,* $(\mathrm{l}+1, \mathrm{b_1}, ..., \mathrm{b_l}, \mathrm{b_{l+1}} ..., \mathrm{b_n})$ *if and only if* $\mathrm{a_i} = \mathrm{b_i}$ *for all the values except for* $\mathrm{i} = \mathrm{l}+1$.

The switch tuples as allocated by the OpenSM's Fat-Tree routing algorithm implemented for an example Fat-Tree, $XGFT(4; 2, 2, 2, 2; 2, 2, 2, 1)$, are shown in Fig. 4. As the Fat-Tree has $n = 4$ switch levels (marked as row 0 at the root level, until row 3 at the leaf level), this Fat-Tree is composed of $m_1 = 2$ *first-level* sub-trees with $n' = n - 1 = 3$ switch levels each (highlighted using two boxes that enclose switches from levels 1 until 3). Each of those first-level sub-trees is composed of $m_2 = 2$ second-level sub-trees with $n'' = n' - 1 = 2$ switch levels each (highlighted using four smaller boxes that enclose switches from levels 2 until 3), above the leaf switches. Similarly, each of the leaf switches can also be considered as a sub-tree, as shown in the figure highlighted using the eight smallest boxes that enclose switches only at level 3. The colored four-number

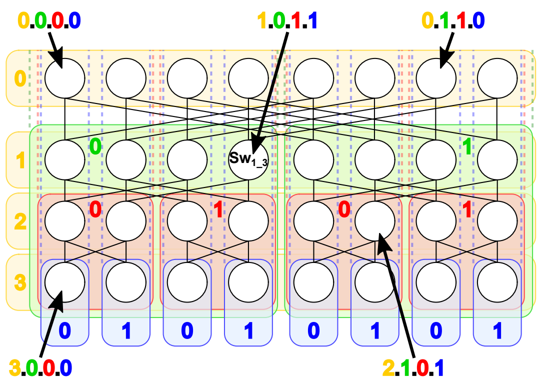


Fig. 4: $XGFT(4; 2, 2, 2, 2; 2, 2, 2, 1)$ switch tuples assignment. Nodes that are located at level 4 are omitted from this figure.

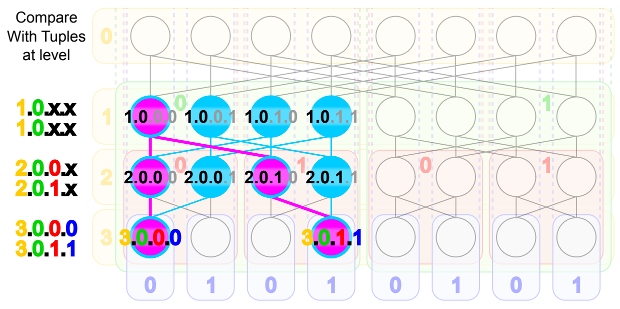


Fig. 5: FTreeMinRC: Minimum reconfiguration on Fat-Trees based on switch tuples for the case of Live Migration.

tuples in the figure are assigned to the pointed switches, and the colors indicate the specific sub-tree correspondence for the position of each value in the tuple. For example, the Switch Sw_{1_3} is assigned with tuple $1.0.1.1$, representing its location at level 1 and $0th$ first-level sub-tree.

6.2 Fat-Tree-aware Minimum Reconfiguration with FTreeMinRC in the Context of Live Migration

As shown above, the switch tuples encode information about the location of the switch in correspondence to the sub-trees in the topology. FTreeMinRC uses this information to enable quick reconfiguration in the case of live VM migration. The tuple information is used to find the skyline with the least number of switches that needs to be reconfigured by the SM when a VM is migrated. In particular, when a VM is migrated between two hypervisors in a Fat-Tree topology, the skyline representing the minimum number of switches that needs to be updated, is formed by all top-level switches of all the sub-trees that are involved in the migration.

To further elaborate, when a VM is live migrated, our switch-marking algorithm starts from both leaf switches where the source and the destination hypervisors are connected, and compares the tuples of the switches. If the tuples match then we know that the VM is being migrated within the leaf switch. Thus, only the corresponding leaf switch is marked for reconfiguration. However, when tuples do not match, the upward links from both the source and the destination leaf switches are traced. The switches that are located one level up are the top-level switches of the immediate super-tree that the leaf-level sub-trees are connected to (Property 1), and the only possible hops before reaching the leaf-switches when traversing the tree downwards. We then compare the source and destination leaf switches tuple with the newly traced switches, after adjusting the tuple value to reflect the current level and the values that correspond to the sub-tree(s) of the current tree are wild-carded. Again, the traced switches (that are top-level switches for a corresponding sub-tree) are marked for updating, and if the comparisons from both the source and destination switch tuples match the tuples of all the traced switches, the tracing stops. Otherwise, the same procedure is repeated until we find the common ancestor switches from both ends. In the worst case, we stop after reaching the root switches of the Fat-Tree topology. Since all the upward-paths are traced starting from the leaf level, and the skyline switches of the consecutive sub-trees are marked (Property 2), when we reach the topmost sub-tree that is affected by the migration we have already selected on the way all the switches that are potential *traffic gateways* towards the lower level switches, and the hypervisors that participate in the live migration. Thus, we have marked all the switches that form the skyline of the affected part of the network due to the live migration.

The switch marking algorithm finds the minimum number of switches that needs to be updated from a physical connectivity perspective. However, not all of these switches contain active paths calculated by the routing algorithm towards the LIDs affected by the reconfiguration. Thus, the switches that contain the active routes can be prioritized in the updating procedure.

The Fat-Tree routing algorithm always routes traffic to a given destination through the same root switch. As only a single path between a root switch and an end node exists in the topology[5], once we locate the root switch that has been selected to represent the given end node we can find the intermediate switches that are used to route traffic to the end node. In order to find the active routes, we trace the path from the source to the destination LID of the participating hypervisors and vice versa. We mark the switches that are a subset of the switches already selected for reconfiguration, and prioritize the LFT updates of those switches. Later, to keep all the LFTs valid, we update the rest of the selected switches.

The overall reconfiguration procedure is shown in Fig. 5, where a VM is migrated between two hypervisors that are connected to the leaf switches with tuples $3.0.0.0$ and $3.0.1.1$. These two tuples are used as the basis for the comparison as we trace the paths upwards from the selected leaf switches. In this example, the common ancestor switches are found on level 1. Level 0 is the root level, and level 3 is the leaf level. The highlighted links are the links that were traced throughout the execution of the algorithm, and the highlighted switches are all marked for update. The five of the switches and four of the links that are highlighted in different color (purple), represent the active routes and their LFT update will be prioritized.

With FTreeMinRC we minimize the number of LFT updates that need to be sent to the switches, in order to provide rapid connectivity with a minimum overhead in a virtualized data center that supports live migrations. Note, that the resulting routing after such a reconfiguration may not be optimal, since the initial routing takes into account the complete topology for the balancing, while FTreeMinRC will alter only a sub-tree of the Fat-Tree. Paths that are contained internally to the sub-tree are fully independent from other sub-trees[6], so if needed a local path re-computation can take place and re-balance the given sub-tree, a task that is in orders of magnitude faster than re-balancing the whole network. For example, it takes more than a minute for the Fat-Tree routing algorithm to calculate the paths for a 3-levels Fat-Tree with 11664 nodes and 1620 36-port switches, while calculating the paths for one of its 2-levels sub-trees with 324 nodes can take a few milliseconds (see section 7). It is noteworthy that live migrations within a leaf switch do not affect the balancing and performance of the network in any way, since the leaf switches are non-blocking and both before and after the migration the packets will reach to the same switch before getting forwarded on a different port towards the final destination.

[5]In PGFTs we can have multiple parallel paths from a root switch to an end node, but not multiple disjoint switches.
[6]Paths that enter/leave the sub-tree must be considered specially.

Nodes	Switches	LIDs	Min LFT Blocks/Switch	Min SMPs Full RC	Min SMPs LID Swap/Copy	Max SMPs LID Swap/Copy
324	36	360	6	216	1	72
648	54	702	11	594	1	108
5832	972	6804	107	104004	1	1944
11664	1620	13284	208	336960	1	3240

Table 2: Number of required SMPs to update LFTs of all switches for the Fat-Tree topologies used in Fig. 6.

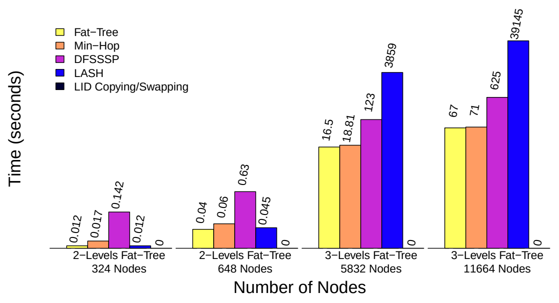


Fig. 6: IBSim Path computation results for different routing algorithms on a Fat-Tree topology with a varied number of nodes.

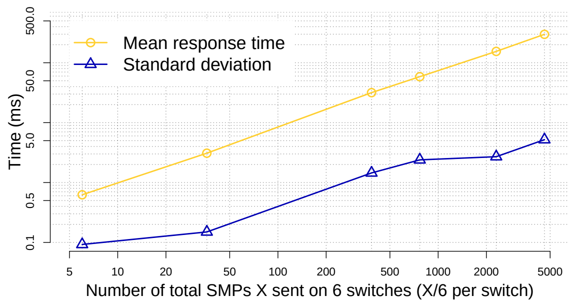


Fig. 7: Measured time to send X LFT-update SMPs on six switches. Time is measured from the moment the first SMP was sent until the last SMP ACK was received for each case.

7 Evaluation

For our results, we implemented the *ItRC* reconfiguration algorithm in OpenSM 3.3.16, and in order to test it we emulated the vSwitch architecture with prepopulated LIDs on hardware. Due to the *Shared Port* support in current hardware all VMs in a hypervisor share the same LID, so we had to limit the number of VMs running in a hypervisor to one for the emulation. We further extended our implementation to include the *vSwitchFatTree* routing algorithm, and the Fat-Tree-aware minimum overhead reconfiguration, *FTreeMinRC*, described in sections 5 and 6.2 respectively. vSwitchFatTree is based on the Fat-Tree routing implementation of OpenSM. In particular, if only one VM is running on each of the hypervisors, vSwitchFatTree will generate the same routing tables as the original Fat-Tree routing algorithm. However, when multiple VMs are running, vSwitchFatTree will independently route each VM while still keeping an even distribution of the assigned routes over the available links in the topology. The routing algorithm works in the same way for both prepopulated LIDs and dynamic LID assignment schemes. Since our testbed is not large (6 IB switches and 8 IB-enabled nodes), we developed a simulator, *vSwitchMigrationSim*, that allowed us to simulate large subnets with the ability to migrate and reconfigure the routes in the network based on both ItRC and FTreeMinRC. vSwitchMigrationSim loads a topology and the corresponding routing tables extracted from OpenSM, performs live migrations, reconfigures the network as necessary and extracts the updated routing tables as well as information such as how many switches were reconfigured after each migration. The Oblivious Routing Congestion Simulator (ORCS) (30) was then used to study the impact of FTreeMinRC on the routing quality and compare it with that of vSwitchFatTree. ORCS is capable of simulating a variety of communication patterns on statically routed networks. Furthermore, we used *ibsim*, a tool that is distributed with the OFED software stack, to emulate physical topologies in order to generate routing tables using OpenSM.

7.1 Dynamic Reconfiguration Evaluation

Ibsim was used to emulate different IB subnets in OpenSM in order to calculate the time it takes for different routing algorithms to compute the routing tables. The path computations were executed on an *HP ProLiant DL360p Gen8* server with 8 CPU cores and 32GB RAM. The results are presented in Fig. 6. We simulated four regular Fat-Tree topologies based on 36-port switches and as one can see, when the network grows larger, the path calculation time corresponding to path computation skyrockets. The path computation time is polynomially increasing with the size of the subnet. It takes 0.012s for the Fat-Tree routing algorithm to compute the routing for 324 nodes, while for a 36 times larger subnet with 11664 nodes and 1620 switches, it takes 67 seconds; ~5583 times longer. DFSSSP, a topology-agnostic routing algorithm needs 0.142 seconds for 324 nodes, while it needs 625 seconds for the subnet with 11664 nodes; ~4401 times longer. LASH needed 39145 seconds for the big subnet with 11664 nodes. With a traditional reconfiguration method, the path computation has to be repeated each time a live-migration is happening and LIDs are changing position in the network. Our proposed reconfiguration methods, ItRC and FTreeMinRC, eliminate this step. For any topology, and independent of the routing algorithm utilized for the initial path computation, zero time is spent in path recalculation (bar with legend *LID Copying/Swapping* in Fig. 6). Thus, live-migration is made possible in vSwitch-enabled IB subnets regardless of the size.

In addition, a full reconfiguration will have to update the complete LFT on each switch and as the network is growing, more switches have to be updated and more SMPs per switch are needed to be sent. For the same four networks that we simulated in Fig. 6, one can see how many LIDs are consumed and the minimum number of SMPs needed for a full reconfiguration in Table 2. Note that the amount of consumed LIDs in a subnet defines the minimum amount of LFT blocks needed to be used on each switch, but not the maximum. Consider the example where we use only three LIDs in a network with one switch and two nodes. If one of the nodes uses the topmost unicast LID, which is 49151, then the $LinearFDBCap$ (1) will get the max value of 49152 and the whole LFT table on the switch will have to be populated, meaning that 768 SMPs will need to be sent on the single switch — instead of the one that would need to be sent if the two nodes and the switch were using LIDs in the range 1-3. Our presented reconfiguration methods, depending on how far a VM is migrated from an interconnection perspective, will need to send a minimum of only one SMP if the VM is migrated within the same leaf switch, or a maximum of $2 * NumberOfSwitches$ SMPs in the extreme case that all of the switches will need to be updated with two SMPs each, as shown in the formulas in Table 1, in the case that ItRC is used. In particular, for the subnet with 324 nodes in Table 2, a full reconfiguration would have to send at least 216 SMPs, while a worst case scenario with our reconfiguration methods will send a max of 72 SMPs, 33.3% of the min number of SMPs required for a full reconfiguration, or 66.7% reduction. For the subnet with 11664 nodes, a full reconfiguration would have to send at least 336960 SMPs, while a worst case scenario with our reconfiguration methods will send a max of 3240 SMPs, 0.96% of the min number of SMPs required for a full reconfiguration, or 99.04% reduction. The best case scenario for our reconfiguration mechanisms is subnet size-agnostic, and will only send one SMP. In Fat-Tree topologies when using FTreeMinRC, even less SMPs when compared to ItRC will be sent on average as we show in section 7.2

To quantify the time needed to send X number of SMPs we ran an experiment with real IB switches, and present the results in Fig. 7. In our testbed we have six IB QDR 40GbE switches. Four of them are the Mellanox IS5025 QDR switches and two of them are the SUN DCS 36 QDR switches, while all six of them are based on the same chipset, the Mellanox InfiniScale® IV. For this experiment we sent $X' = X/6$ LFT update SMPs on each switch and measured the time interval between the first SMP sent, and until the last SMP ACK was received. To increase the number of SMPs X' sent per switch, we increased the $LinearFDBCap$. As one can see in Fig. 7 the time it takes for X SMPs to be sent and their ACK received on six switches grows linearly, and when $X = 4608$, ~300ms are required for the reconfiguration of the switches. When looking at the very worst case scenario of our reconfiguration methods for the largest network in Table 2, up to 3240 SMPs will have to be sent, placing the time on the linear line of Fig. 7 at ~210ms. One more observation that we had during these experiments, is that more switches can handle X SMPs faster, meaning that in a large network with 1620 switches handling 3240 SMPs would take even less time than the one presented here. Even if it took 210ms, this time is significantly less than the VM downtime of a VM with an IB SR-IOV VF, due to the stop-and-copy phase of a live migration (12).

7.2 Reconfiguration with Minimum Overhead on Fat-Trees

In this section FTreeMinRC is evaluated in detail. The FTreeMinRC algorithm needs to identify and mark the switches that needs to be updated in a Fat-Tree topology. The switch-marking execution overhead of FTreeMinRC for three large-scale Fat-Tree topologies can be seen in Table 3. For two very large tangible networks, as one can see in Table 3, the execution overhead is 3ms and 3.4ms respectively. This time can be spent before the migration starts or during the pre-copy phase of the live migration while the VM connectivity is not yet suspended. Thus, the switch-marking algorithm adds a negligible overhead in the whole migration process when compared to the total migration overhead. It is worth mentioning that the times presented in Table 3 represent a worst case scenario where a VM is migrated between the edges of the Fat-Tree, meaning that the highest possible number of switches will have to be traversed and marked for reconfiguration. The network in the third line of Table 3 is not a topology that somebody would choose with today's standards, but it has been included as advancements in hardware and the introduction of new lossless network interconnects with larger *LID* space will make efficient 4-level Fat-Trees a reality in the near future. The intention here is to simulate such a topology in order to see how the execution of the algorithm performs when a very large number of switches distributed in several levels of a Fat-Tree needs to be traversed. As one can see, even in a topology with 6912 switches the switch-marking algorithm of FTreeMinRC executes in 79ms. Still, not a very sizable amount when compared to the total migration overhead. In the rest of this section we present the simulations we ran to evaluate the routing quality resulting from minimally reconfiguring the network with FTreeMinRC.

Simulations

For the simulations we ran experiments for all the possible combinations of *Topologies*, *Initial VM Distribution*, *Migration Pattern* and *VM Population per Hypervisor* shown in Table 4. In total, we ran 21 simulations per topology, for six different

Fat-Tree topology description	Algorithm execution speed
3 Levels Fat-Tree: 972 Switches/5832 Nodes	3ms
3 Levels Fat-Tree: 1620 Switches/11664 Nodes	3.4ms
4 Levels Fat-Tree: 6912 Switches/20736 Nodes	79ms

Table 3: Time needed for FTreeMinRC to find the min number of switches that needs to be reconfigured when a VM is migrated between the edges of a Fat-Tree (worst case scenario).

Fat-Tree topology	Topology Description	Num Nodes /Switches	Initial VM Distribution	Migration Pattern	VM Population per Hypervisor
XGFT(2;8,4;4,1)	4-ary-2-full	32/12	Random Uniform	Random Consolidate Grouping To-Uniform (*To-Uniform* pattern is only used for the random initial VM distribution)	25%, 50%, 75% of max capacity (Each hypervisor has 4 VFs so the max (100%) capacity $m_c = 4 \cdot ftree\_num\_nodes$)
XGFT(2;8,8;8,1)	8-ary-2	64/16			
XGFT(3;8,4,4;4,4,1)	4-ary-3-full	128/80			
XGFT(2;16,16;16,1)	16-ary-2	256/32			
XGFT(3;8,8,8;8,8,1)	8-ary-3	512/192			
XGFT(3;16,8,8;8,8,1)	8-ary-3-full	1024/320			

Table 4: Simulation parameters. Recall from section 6.1 that our XGFT definition uses a slightly different notation.

topologies, with the largest of the simulations when using the *8-ary-3-full* topology and having the *VM population per hypervisor* set to 75% counting as many as 3072 VMs. The initial VM distribution is chosen to be either *uniform* or *random*. In the uniform VM distribution, all the vSwitches start with an equal number of connected VMs, while VMs are distributed to the vSwitches randomly in the random distribution. Once the initial topologies have been loaded using ibsim, OpenSM builds the initial routing tables with the vSwitchFatTree routing algorithm. Consequently, we load these topologies and routes in the *vSwitchMigrationSim* simulator. Then we perform VM migrations and: 1) Extract information about the number of switches that have been reconfigured after every single migration; 2) Output minimally reconfigured LFTs every 10% of the total migrations for each pattern running on each topology[7]. Four migration patterns are used: *random*, *consolidate*, *grouping*, and *to-uniform*. In the *random* migration pattern, $2 \cdot num\_nodes\_in\_topology$ random migrations are performed for each of the simulated topologies. The random migration scenario is used to simulate a scenario where a network would be in a relatively random state after several standalone optimizations without a global scope may have taken place in a data center. The *consolidate* migration pattern packs all the VMs to the left side of the Fat-Tree and uses the least number of hosts possible that can accommodate the number of booted VMs. Consolidation is a typical power saving scenario with a global data center scope. In the *grouping* pattern, two randomly chosen VM groups are consolidated and isolated on neighboring hypervisors. The grouping pattern simulates a case where two different tenants share fragmented resources and they need to be consolidated independently. Last, *to-uniform* migration pattern achieves a uniform VM distribution on the vSwitches in the network, starting with a random initial VM distribution. The intention of the *to-uniform* migration pattern is to simulate a scenario where the cloud provider tries to improve VM performance by spreading out VMs evenly in the data center[8].

For each topology dump from our *vSwitchMigrationSim* simulator, we re-ran OpenSM with vSwitchFatTree to generate fresh routing tables for the given snapshot (recall that every snapshot is every 10% of the total migrations for each scenario).

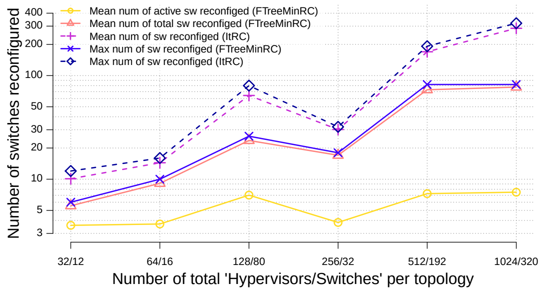
[7]Our choice to extract the minimally reconfigured LFTs every 10% of the total migrations is practical. We wanted to keep the number of ORCS simulations (as explained in the next paragraph) in a sensible number so that all the simulations could complete within a few weeks. As can be seen from the simulation results in Fig. 8 the 10% step is good enough to show the performance trends over time (percentage of total migrations) for the different simulation scenarios.

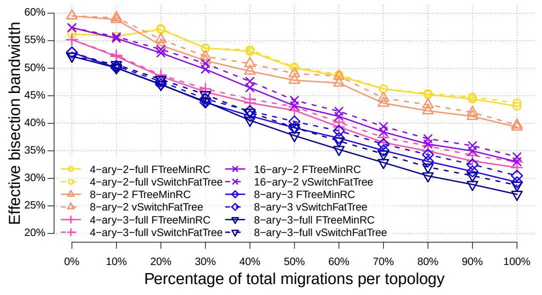
[8]The migration patterns are used to show the efficiency of our reconfiguration methods and the vSwitchFatTree routing algorithm, and are not optimized to perform the minimum number of migrations.

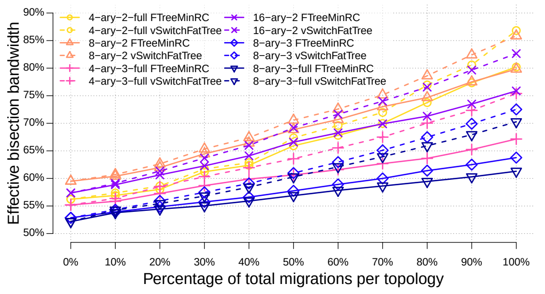
The quality of the routing tables from OpenSM and *vSwitchMigrationSim*, that performs minimum reconfiguration, are then compared using the ORCS *bisect* communication pattern. The reason for comparing the routing quality from a routing algorithm (*vSwitchMigrationSim*) and a reconfiguration algorithm (*FTreeMinRC*), is in order to study the impact of the latter after several migrations have taken place in a subnet. The scope of *FTreeMinRC* is to reconfigure the network quickly and make live migrations possible with a minimal number of switches reconfigured, thus, we hypothesize that after several migrations the routing quality would be degraded. Note that we do not compare vSwitchFatTree with any other routing algorithm, as no other routing algorithm exists yet that takes into account the nature of the vSwitches in the topology. Any such comparison would be unfair and meaningless. The ORCS *bisect* pattern emulates effective bisection bandwidth, where nodes in a partition are split into two equally sized halves. Each node in the first half sends a message to a node in the second half. To obtain statistically significant results, all simulations were executed 10000 times with randomly chosen nodes in both partitions to eliminate the affect of node selection in the bisect pattern of the ORCS simulator. As we obtained too many results and it is not possible to present everything, the most representative sample is shown in Fig. 8.

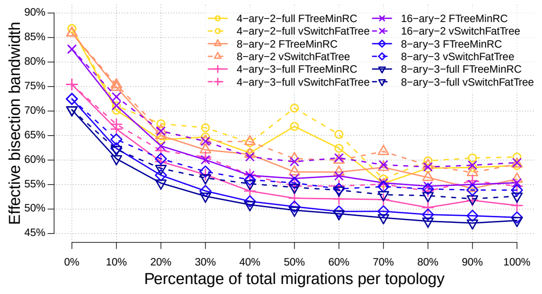
In Fig. 8(a) we present the average number of switches that were reconfigured for each of the Fat-Tree topologies in Table 4, for each individual migration. We compare ItRC and FTreeMinRC. As expected, the topology-aware method, FTreeMinRC, performs significantly better, and both the mean number of total switches reconfigured as well as the max number is lower. In addition, FTreeMinRC prioritizes the updates of the switches containing active routes, meaning that the connectivity can be re-established in a matter of single-digit milliseconds according to our empirical results shown in Fig. 7. This is less than the reported downtime attributed to the stop-and-copy phase in a live migration for both IB (12) and the more mature Ethernet (7). For the rest of the figures in Fig. 8 the achieved bisection bandwidth and routing quality is compared between FTreeMinRC and vSwitchFatTree. In particular, in Fig. 8(b), 8(c), and 8(d), we present the *random* to *grouping*, *random* to *uniform*, and *uniform* to *random* migration scenarios respectively, when 25% of the data center VM capacity is being utilized for each topology. In Fig. 8(e) and Fig. 8(f), we present the *random* to *consolidate* and *uniform* to *consolidate* migration scenarios respectively, when 50% of the data center VM capacity is being utilized for each topology. Last, in Fig. 8(g) and Fig. 8(h), we present the *random* to *uniform* and *uniform* to *random* migration scenarios respectively, when 75% of the data center VM capacity is being utilized for each topology.

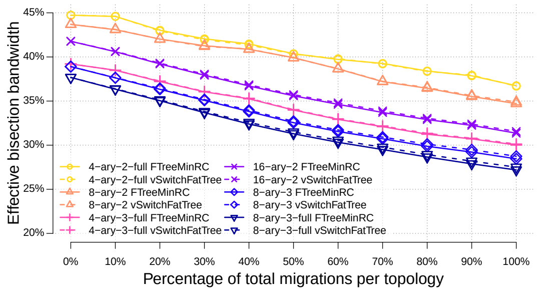
Firstly, a remark that applies to all Fig. 8(b)-(h) is that the bisection bandwidth becomes worse as the total VMs in the data center increase from 25% to 50% and 75% of the maximum data center capacity. This is natural, and of course, when more VMs exist there is higher contention for the available resources. One more global observation is that there is no case in Fig. 8(b)-(h) where the routing resulted from FTreeMinRC delivers better performance when compared to the routing resulted from vSwitchFatTree. In particular, in Fig. 8(c), Fig. 8(d), Fig. 8(g), and Fig. 8(h), when we migrate from a uniform to a random distribution of VMs and vice versa, we see a clear relative degradation of the performance as the number of the migrations progress and the network is reconfigured with FTreeMinRC. In some cases we can see almost 10% of relative performance improvement when the network is re-balanced with vSwitchFatTree. This observation is aligned with our earlier hypothesis

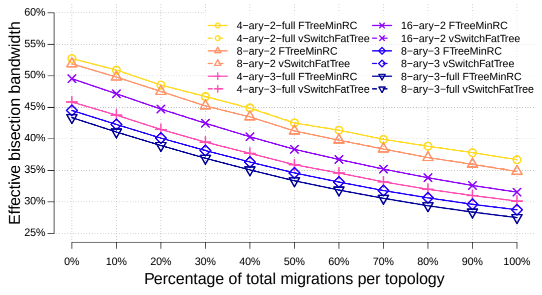

(a) Number of switches updated with FTreeMinRC vs the number of switches updated with ItRC when performed live migrations on the Fat-Tree topologies presented in Table 4 (lower number is better).

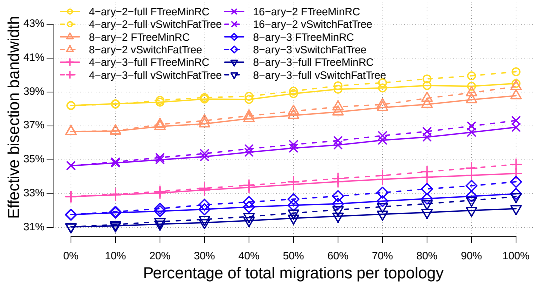

(b) Bisect bandwidth for 25% occupancy of the available VFs, *random* initial distribution, *grouping* migration pattern.


(c) Bisect bandwidth for 25% occupancy of the available VFs, *random* initial distribution, *to-uniform* migration pattern.


(d) Bisect bandwidth for 25% occupancy of the available VFs, *uniform* initial distribution, *random* migration pattern.


(e) Bisect bandwidth for 50% occupancy of the available VFs, *random* initial distribution, *consolidate* migration pattern.


(f) Bisect bandwidth for 50% occupancy of the available VFs, *uniform* initial distribution, *consolidate* migration pattern.


(g) Bisect bandwidth for 75% occupancy of the available VFs, *random* initial distribution, *to-uniform* migration pattern.

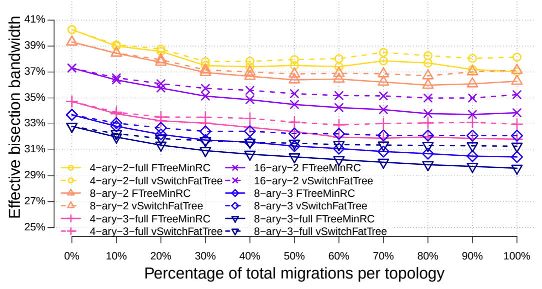

(h) Bisect bandwidth for 75% occupancy of the available VFs, *uniform* initial distribution, *random* migration pattern.

Fig. 8: Simulation results (Higher values in all subfigures except (a) indicate better performance).

that after several migrations and reconfigurations with FTreeMinRC the routing quality would be degraded. The reason why vSwitchFatTree provides better performance, is because vSwitchFatTree re-balances the routes for all of the VMs. FTreeMinRC only performs local reconfiguration without taking into account the rest of the network after each migration. The trade-off, of course, is that vSwitchFatTree goes through the expensive path computation phase that FTreeMinRC aims to eliminate in order to make live migrations feasible. Based on that observation, the general scheme that we suggest to utilize in virtualized environments with vSwitches is to use FTreeMinRC for migrations (ItRC for topology agnostic reconfiguration), and from time to time call a routing algorithm such as the vSwitchFatTree to optimize the current state of the network.

Probably the most interesting observation that can be made when one focuses in Fig. 8(b), Fig. 8(e), and Fig. 8(f), is that whenever we have a consolidation scenario (including grouping, which is basically consolidation by taking into account different tenants), the resulting routing quality of vSwitchFatTree and FTreeMinRC produces equal results, or a very tiny difference in the order of ∼1% exists. This can be explained because the initial routing, that has been provided by vSwitchFatTree, balances the PF of the vSwitches as well. When FTreeMinRC reconfigures the network, it is using the routes of the PFs to route the migrated VMs. As such, in consolidation scenarios where all of the VFs will be occupied in a hypervisor, and all neighboring hypervisors will end up having the same number of VMs (If the hypervisors have similar capabilities with the same number of VFs), all of the occupied hypervisors will have the same weight in the network. Since the vSwitches have already been balanced, the resulting routing from FTreeMinRC and a re-balancing with vSwitchFatTree, will give almost the same routing as the one that was calculated by vSwitchFatTree before the migrations started. This interesting finding indicates that in power saving scenarios where VMs are consolidated, further routing optimizations might not be necessary once all of the migrations complete.

Some other general observations in our results include: 1) In the cases where we migrate from a random to a uniform distribution the performance overall is improving. Again, this is expected since a uniform distribution will utilize the network resources evenly (i.e. there will be no cases where VMs are overpacked in hypervisors, while parts of the data center are idle). Exactly the opposite happens in the migration scenarios from uniform to random. 2) The results from the smallest topology tested (*4-ary-2-full*) exhibit more fluctuations when compared to the rest of the results, and the results of the largest topology (*8-ary-3-full*) are the most stable. This behavior is observed due to the number of nodes and VMs in each topology. When 25% of the maximum VM capacity is used, the *4-ary-2-full* topology only has 32 VMs. With such few VMs, even a small number of migrations may be enough to alter the topology significantly in order to observe these fluctuations.

8 Conclusion

In this paper, we revisited and complemented our proposed *vSwitch* SR-IOV architecture for lossless interconnection networks (33). We used IB to demonstrate our concept prototypes and suggested different options that can help to overcome the scalability issues related to the layer-two LID space address limitations of IB. Routing strategies for vSwitch-based subnets were discussed and a Fat-Tree routing algorithm was presented, *vSwitchFatTree*, that takes into account the shared connection of VMs residing at the same hypervisor. Furthermore, we presented *FTreeMinRC*, a Fat-Tree topology-aware reconfiguration algorithm for vSwitch-powered subnets. *FTreeMinRC* considers switch locations in the topology to identify and reconfigure the minimum number of switches when a VM is migrating, resulting in reduced overhead when compared to our previously proposed topology-agnostic reconfiguration algorithm, *ItRC*. Our results show that we are able to significantly reduce network reconfiguration time by eliminating the path computation phase when VMs live migrate. Moreover, in certain scenarios where VMs migrate within a leaf-switch in a Fat-Tree topology, the number of required reconfiguration management packets sent to switches can be reduced down to a single one. Based on empirical measurements and simulations, we show that the total execution time of a reconfiguration with our devised methods is not a limiting factor for live migrations anymore. Thus, live migration can become a practical option from the subnet management perspective in virtualized environments that are based on lossless network technologies. We have also shown that depending on the migration scenario, it can be sufficient to rely on minimum overhead reconfigurations with *FTreeMinRC* and not needed to re-balance the network traffic with a complete performance-driven re-routing, as we observed only a marginal degradation in the routing quality.

Acknowledgements

We would like to acknowledge Mellanox Technologies for providing us with IB hardware, and the Norwegian research council for funding the ERAC project (nr. 213283/O70).

References

1. *InfiniBand Architecture General Specifications 1.3*: InfiniBand Trade Association; 2015
2. *InfiniBand Architecture General Specifications 1.3, Annex A18: Virtualization*: InfiniBand Trade Association; 2016
3. Al-Fares M., Loukissas A., Vahdat A. (). *A Scalable, Commodity Data Center Network Architecture*, Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, pp. 63–74.
4. Bermúdez A., Casado R., Quiles F. J., Duato J. (). *Use of Provisional Routes to Speed-up Change Assimilation in InfiniBand Networks*, 18th International Parallel and Distributed Processing Symposium, pp. 186–193.
5. Birrittella M. S., Debbage M., Huggahalli R., Kunz J., Lovett T., Rimmer T., Underwood K. D., Zak R. C. (). *Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics*, IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 1–9.
6. Bogdański B., Johnsen B. D., Reinemo S., Flich J. Making the Network Scalable: Inter-subnet Routing in InfiniBand,In Wolf Felix, Mohr Bernd, an Mey Dieter (eds.) 2013 (pp. 685–698)., Euro-Par 2013 Parallel Processing: Springer Berlin Heidelberg (English).
7. Clark C., Fraser K., Hand S., Hansen J. G., Jul E., Limpach C., Pratt I., Warfield A. (). *Live Migration of Virtual Machines*, Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2, pp. 273–286.
8. Derradji S., Palfer-Sollier T., Panziera J. P., Poudes A., Atos F. W. (). *The BXI Interconnect Architecture*, IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 18–25.
9. Dong Y., Yang X., Li J., Liao G., Tian K., Guan H. High Performance Network Virtualization with SR-IOV. *Journal of Parallel and Distributed Computing*. 2012;72(11):1471 –1480 Communication Architectures for Scalable Systems.
10. Duato J. A Necessary and Sufficient Condition for Deadlock-Free Routing in Cut-Through and Store-and-Forward Networks. *IEEE Transactions on Parallel and Distributed Systems*. 1996Aug;7(8):841–854.
11. Guay W. L. *Dynamic Reconfiguration in Interconnection Networks*: University of Oslo. Ph.D. Thesis; 2014
12. Guay W. L., Reinemo S., Johnsen B., Yen C., Skeie T., Lysne O., Tørudbakken O. Early Experiences with Live Migration of SR-IOV Enabled InfiniBand. *Journal of Parallel and Distributed Computing*. 2015;78:39 –52.
13. Guay W. L., Reinemo S., Johnsen B. D., Skeie T., Torudbakken O. (). *A Scalable Signalling Mechanism for VM Migration with SR-IOV over InfiniBand*, IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS), pp. 384–391.
14. Hillenbrand N., Mauch V., Stoess J., Miller K., Bellosa F. (). *Virtual InfiniBand Clusters for HPC Clouds*, Proceedings of the 2nd International Workshop on Cloud Computing Platforms, pp. 9.
15. Jose J., Li M., Lu X., Kandalla K. C., Arnold M. D., Panda D. K. (). *SR-IOV Support for Virtualization on InfiniBand Clusters: Early Experience*, 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pp. 385–392.

16. Kutch P. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. *Application Note*. 2011.

17. Leiserson C. E. Fat-trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*. 1985;100(10):892–901.

18. Liss L. (). *Infiniband and RoCEE Virtualization with SR-IOV*, 2010 OFA International Workshop.

19. Lysne O., Duato J. (). *Fast Dynamic Reconfiguration in Irregular Networks*, Proceedings of the International Conference on Parallel Processing., pp. 449–458.

20. Lysne O., Montañana J. M., Pinkston T. M., Duato J., Skeie T., Flich J. Simple Deadlock-Free Dynamic Network Reconfiguration 2005 (pp. 504–515)., High Performance Computing-HiPC 2004: Springer.

21. Mauch V., Kunze M., Hillenbrand M. High Performance Cloud Computing. *Future Generation Computer Systems*. 2013;29(6):1408–1416.

22. Morgenstein J. *Add SRIOV support for IB interfaces*. https://web.archive.org/web/20170518120225/http://www.mail-archive.com/linux-rdma@vger.kernel.org/msg11956.html; 2012

23. Öhring S. R., Ibel M., Das S. K., Kumar M. J. (). *On Generalized Fat Trees*, Proceedings of the 9th International Symposium on Parallel Processing, pp. 37–.

24. Pang R., Pinkston T. M., Duato J. (). *The Double Scheme: Deadlock-free Dynamic Reconfiguration of Cut-Through Networks*, Proceedings of the International Conference on Parallel Processing., pp. 439–448.

25. Petrini F., Vanneschi M. (). *k-ary n-trees: High Performance Networks for Massively Parallel Architectures*, Proceedings 11th International Parallel Processing Symposium, pp. 87–93.

26. Pfister G. F. An Introduction to the InfiniBand Architecture. *High Performance Mass Storage and Parallel I/O*. 2001;42:617–632.

27. Rad P., Boppana R. V., Lama P., Berman G., Jamshidi M. (). *Low-latency Software Defined Network for High Performance Clouds*, 10th System of Systems Engineering Conference (SoSE), pp. 486–491.

28. Ranadive A. U. *Virtualized Resource Management in High Performance Fabric Clusters*: Georgia Institute of Technology. Ph.D. Thesis; 2015

29. Robles-Gómez A., Bermúdez A., Casado R., Solheim Å. G. Deadlock-Free Dynamic Network Reconfiguration Based on Close Up*/Down* Graphs 2008 (pp. 940–949)., Euro-Par 2008–Parallel Processing: Springer.

30. Schneider T., Hoefler T., Lumsdaine A. ORCS: An Oblivious Routing Congestion Simulator. *Indiana University Technical Report, TR-675*. 2009.

31. Sem-Jacobsen F. O., Lysne O. (). *Topology Agnostic Dynamic Quick Reconfiguration for Large-Scale Interconnection Networks*, Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012), pp. 228–235.

32. Strohmaier E., Dongarra J., Simon H., Meuer M. *Top500.org*. http://www.top500.org/; 2017

33. Tasoulas E., Gran E. G., Johnsen B. D., Begnum K., Skeie T. (). *Towards the InfiniBand SR-IOV vSwitch Architecture*, IEEE International Conference on Cluster Computing (CLUSTER), 2015, pp. 371–380.

34. Tasoulas E., Gran E. G., Johnsen B. D., Skeie T. (). *A Novel Query Caching Scheme for Dynamic InfiniBand Subnets*, 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).

35. Tasoulas E., Gran E. G., Skeie T., Johnsen B. D. (). *Fast Hybrid Network Reconfiguration for Large-Scale Lossless Interconnection Networks*, IEEE 15th International Symposium on Network Computing and Applications (NCA), pp. 101–108.

36. Zafar B., Pinkston T. M., Bermúdez A., Duato J. Deadlock-Free Dynamic Reconfiguration Over InfiniBand™ Networks. *Parallel Algorithms and Applications*. 2004;19(2-3):127–143.

37. Zahavi E. D-Mod-K Routing Providing Non-Blocking Traffic for Shift Permutations on Real Life Fat Trees. *CCIT Report 776, Technion*. 2010.

38. Zahavi E. Fat-tree Routing and Node Ordering Providing Contention Free Traffic for MPI Global Collectives. *Journal of Parallel and Distributed Computing*. 2012;72(11):1423–1432.

39. Zahavi E., Johnson G., Kerbyson D. J., Lang M. Optimized InfiniBand Fat-Tree Routing for Shift All-to-All Communication Patterns. *Concurrency and Computation: Practice and Experience*. 2010;22(2):217–231.

40. Zahavi E., Shpiner A., Rottenstreich O., Kolodny A., Keslassy I. (). *Links As a Service (LaaS): Guaranteed Tenant Isolation in the Shared Cloud*, Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems, pp. 87–98.

41. Zahid F., Gran E. G., Bogdański B., Johnsen B. D., Skeie T., Tasoulas E. Compact Network Reconfiguration in Fat-Trees. *The Journal of Supercomputing*. 2016;72(12):4438–4467.

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

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