# Links as a Service (LaaS): Guaranteed Tenant Isolation in the Shared Cloud

Eitan Zahavi, Alexander Shpiner, Ori Rottenstreich, Avinoam Kolodny, and Isaac Keslassy

***Abstract*— The most demanding tenants of shared clouds require complete isolation from their neighbors, in order to guarantee that their application performance is not affected by other tenants. Unfortunately, while shared clouds can offer an option, whereby tenants obtain dedicated servers, they do not offer any network provisioning service, which would shield these tenants from network interference. In this paper, we introduce links as a service (LaaS), a new abstraction for cloud service that provides isolation of network links. Each tenant gets an exclusive set of links forming a virtual fat-tree, and is guaranteed to receive the exact same bandwidth and delay as if it were alone in the shared cloud. Consequently, each tenant can use the forwarding method that best fits its application. Under simple assumptions, using bipartite graph properties and pigeonhole-based analysis, we derive theoretical conditions for enabling the LaaS without capacity over-provisioning in fat-trees. New tenants are only admitted in the network, when they can be allocated hosts and links that maintain these conditions. We also provide new results on the numbers of tenants and hosts that can fit while guaranteeing network isolation. The LaaS is implementable with common network gear, tested to scale to large networks, and provides full tenant isolation at the cost of a limited reduction in the cloud utilization.**

***Index Terms*— Communication networks, software defined networking, computer networks, computer network management.**

## I. Introduction

### *A. Problem Background (The Network-Softwarization Predictability Hurdle in Shared Clouds):*

Many companies that own private data centers would like to move to a shared multi-tenant cloud, which can offer a significantly reduced cost of ownership and better fault-tolerance. It is vital for some of these companies that their applications will not be affected by other tenants, and will keep exhibiting the same performance[1] [10], [37], [38]. For example, a banking application may need to roll-up all accounts data overnight, and a weather prediction software should similarly complete within a highly predictable time. For such tenants, run-time predictability is a key requirement.

Unfortunately, distributed applications often suffer from unpredictable performance when run on a shared cloud [11], [27]. This unpredictable performance is mainly caused by two factors: *server sharing* and *network sharing* [14], [17], [21], [25], [35], [36], [39], [41], [51]. The first factor, *server sharing*, is easily addressed by using bare-metal provisioning of servers, such that each server is allocated to a single tenant [4]. However, the second factor, *network sharing*, is much more difficult to address. When datacenter network links are shared by several tenants, network contention can significantly worsen the application performance if other tenant applications consume more network resources, *e.g.,* if they simply want to benchmark their network or run a heavy backup [32]. This can of course prove even worse when other tenants purposely generate adversarial traffic for DoS or side-channel attacks [46].

In addition, shared clouds seemingly form a barrier to network softwarization. They cannot currently allow tenants to apply the network algorithms that they use in their private data centers, such as private routing, load-balancing, traffic engineering, self-management, self-optimization, and NFV (network function virtualization) forwarding algorithms. Allowing different forwarding and management algorithms for different tenants may result in a significantly increased unfairness between aggressive and non-aggressive tenants in a way that is hard to predict. For companies that are thinking of moving to the shared cloud, this inability to have access to their own fully-programmable, optimized, and automated network slice forms a major barrier.

As detailed in the Related Work (Section II), current solutions for providing guarantees either (a) require tenants to provide and adhere to a specific traffic matrix declared in advance, which often proves impractical [14]; (b) follow the hose model by providing enough throughput for any set of admissible traffic matrices [11], [22], but also significantly reduce the link bandwidth and burst size that can be allocated to each VM; or (c) attempt to track the current traffic matrix,

Manuscript received October 10, 2018; revised January 30, 2019 and March 10, 2019; accepted March 13, 2019. Date of current version April 16, 2019. This work was supported in part by the Technion Hiroshi Fujiwara Cyber Security Research Center, in part by Israel Cyber Bureau, in part by the Technion Funds for Security Research, in part by the Intel ICRI-CI Center, in part by the Hasso Plattner Institute Research School, in part by the Gordon Fund for Systems Engineering, in part by the Israel Ministry of Science and Technology, and in part by the Shillman, Erteschik, and Greenberg Research Funds. The work of O. Rottenstreich was supported by the Taub Family Foundation. *(Corresponding author: Ori Rottenstreich.)*

E. Zahavi is with Mellanox Technologies, Sunnyvale, 94085 CA USA (e-mail: eitan@mellanox.com).

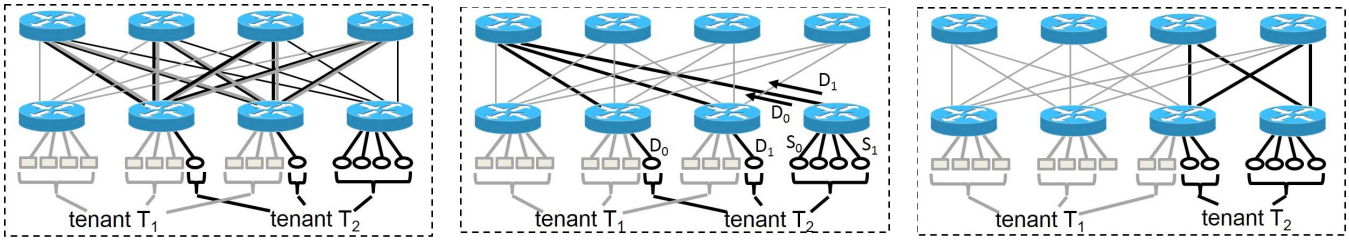
A. Shpiner is with Lightbits Labs., Inc., Kfar Saba 4464313, Israel (e-mail: alex.shpiner@gmail.com).

O. Rottenstreich, A. Kolodny, and I. Keslassy are with the Technion-Israel Institute of Technology, Haifa 3200003, Israel (e-mail: or@cs.technion.ac.il; kolodny@ee.technion.ac.il; isaac@ee.technion.ac.il).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2019.2906747

[1]By *performance*, we refer to the inverse of either the total application run-time, including both the computation and communication times, or of the response time of online services.




(a) No LaaS: Shared links (b) No LaaS: Bandwidth loss (c) LaaS: Full isolation

Fig. 1. Illustration of cloud resource allocation for two tenants. (a) The traffic of the two tenants interferes on many shared links. (b) There are no shared links, but the second tenant cannot service an admissible traffic from $S_0$ and $S_1$ to $D_0$ and $D_1$. (c) Under LaaS conditions of tenant placement and link allocation, a link is assigned to at most one of the tenants and the network can service any admissible tenant traffic demands.

but cannot guarantee constant performance [25], [36], [51]. Furthermore, none of the current cloud solutions allow multiple programmable forwarding and management algorithms to co-exist on the same network without impacting performance.

### *B. Problem Statement*

We want to find conditions under which we can *guarantee a full network isolation* between different tenants of a shared cloud, such that each tenant can obtain the exact same bandwidth and delay independently of the number and bandwidth of other tenants.

### *C. Contributions*

In this paper, we introduce *Links as a Service (LaaS)*, a novel shared-cloud architecture model that guarantees a full isolation between all tenants. Keeping with the notion that good fences make good neighbors, the main idea of LaaS is that *each tenant that requests isolation should receive its own dedicated sub-fat-tree network,* i.e., *an* exclusive access to a subset of the data center links. As a result, the main novelty of the LaaS model consists in fully guaranteeing that this tenant can obtain the exact same bandwidth and delay as if it were alone in the shared cloud, *i.e.,* its performance cannot be affected by any other tenant. In addition, LaaS allows each tenant to use a private fully-programmable network forwarding and management algorithm that is optimized for its own application. Finally, we show that allocation of links to tenants is cost-effective and implementable by using common hardware.

While the LaaS abstraction is attractive, Figure 1 illustrates why it can be a challenge to provide it given any arbitrary set of tenants. First, Fig. 1(a) illustrates a bare-metal allocation of distinct hosts (servers) to two tenants that does not satisfy the LaaS abstraction, since the tenants share common links. Likewise, the allocation of hosts and links in Fig. 1(b) also does not satisfy LaaS, even though no links are shared between tenants. This is because, regardless of the packet forwarding algorithm, internal traffic of the second tenant from the two hosts $S_0$ and $S_1$ in the right leaf switch to hosts $D_0$ and $D_1$ would need to share a common link, and so some admissible traffic patterns would not be able to obtain full bandwidth. Interestingly, for this host placement, we find that there is in fact no link allocation that can provide full bandwidth to all the admissible traffic patterns of both tenants. Finally, Fig. 1(c) fully satisfies the LaaS conditions. All tenants obtain dedicated hosts and links, and can service any admissible traffic demands between their nodes, independently of the traffic of other tenants. This illustrates the main practical novelty of this paper.

The main theoretical novelty of this paper is to generalize the above examples and analyze the fundamental requirements for providing LaaS guarantees to tenants in 2- and 3-level homogeneous fat-trees, *i.e.,* guarantee that the delay and bandwidth provided to a given tenant is completely independent of the other tenants. Specifically, we first provide necessary conditions on host placement (Theorem 2), then provide necessary and sufficient conditions on link allocation (Theorem 1). We also prove that in small fat trees, when the required host number fits the fat-tree size, an assignment satisfying LaaS always exists for any number of tenants and for any combination of the tenant sizes (Theorem 3). We further prove that LaaS also holds in large fat trees, whenever there are at most two large tenants (Theorem 4).

Furthermore, while the above results guarantee LaaS isolation and performance with 100% host utilization, we also want to guarantee LaaS with a limited reduction in utilization (*e.g.,* 90%) in a more general set of cases. Theorems 5 and 6 below intuitively guarantee that as long as there is a relatively large proportion of small tenants, then the utilization will be relatively high, *i.e.,* most of the cloud servers will be utilized while maintaining our strong goal of a full LaaS isolation and performance guarantee. For instance, if small tenants request 80% of the hosts and large tenants request 10% of the hosts, then we are guaranteed that all tenants will fully fit and satisfy LaaS. Note that our results rely on a range of methodological tools, including bipartite graph properties and pigeonhole-based analysis.

The above results greatly reduce the complexity of our online allocation algorithm, even in the case of a 3-level fat tree, as we detail in Section V. In addition, in the evaluations section (Section VI), we implement a standalone LaaS scheduler that automates tenant placement on top of OpenStack, as well as configures an InfiniBand SDN controller to provide forwarding without interference. Our open-source code is made available online [1]. We show that using this code, our LaaS algorithm responds to tenant requests within a few milliseconds, even on a cloud of 11K nodes, *i.e.,* several orders of magnitude faster than the time it takes to

provisioning a new virtual machine. In addition, when the average tenant size is smaller than a quarter of the cloud size, we find that our LaaS algorithm achieves a cloud utilization of about 90%, for various tenant-size distributions. For larger tenant sizes, our LaaS allocation converges to the maximal utilization obtained by a bare-metal scheduler that packs tenants without constraints. Finally, to demonstrate LaaS strength, we show performance improvements of 50%–200% for highly-correlated tenant traffic generated by a Bulk Synchronous Parallel (BSP) application relying on data exchanges along a virtual three-dimensional axis system. The performance improvement exceeds the utilization cost for such applications, uncovering an economic potential. Thus, our evaluations show that LaaS is practical and efficient, and completely avoids inter-tenant performance dependence.

For the sake of presentation, we present all proofs and further discussion of our results in an extended online version [55].

## II. Related Work

### A. Application Variability

Several studies about the variability of cloud services and HPC application performance were presented by [11], [27], [32]. They show significant variability for such applications which strengthens LaaS motivation.

### B. Network Isolation

Specific high-dimensional tori super-computers like IBM BlueGene, Cray XE6, and the Fujitsu K-computer provide scheduling techniques to isolate tenants [5], [42]. However, they all rely on forming an isolated cube on 3 out of the 5- or 6-dimensional torus space, and thus cannot be used in clouds with fat-tree topologies. They also exhibit a significantly lower cluster utilization, measured as the amount of servers used over time, than the 90% utilization obtained by LaaS on fat-trees. Another approach reduces the interference between jobs running on the same fat-tree but does not guarantee job isolation from each other [33]. Likewise, pFTree is a fat-tree routing algorithm that aims to provide network isolation in fat trees [56], [57]. However, it does not provide any guarantee of network isolation nor of full bisection bandwidth, which are the aims of this paper. It also does not consider the improved guarantees that result from alternative placement options.

### C. Packet Forwarding

Many architectures rely on Equal Cost Multiple Path (ECMP) [26] to spread the allocated tenant traffic and avoid the need to allocate exact bandwidth on each of the used physical links [11], [30]. However, while ECMP load-balancing is able to balance the average bandwidth of many small bandwidth flows, it suffers from a heavy tail of the load distribution. When traffic contains a relatively small number of large flows, ECMP is known to provide poor load-balancing. Thus, other tenants will affect the application performance.

Silo [29] provides guaranteed latency, bandwidth and burst size to multiple tenants for a worst-case traffic pattern, by applying accurate rate- and burst-size moderation to enforce centrally-calculated values obtained from network calculus. Unlike LaaS, in Silo multiple hosts of the same tenant might need to share link bandwidth where the link has to be used based on the existing forwarding rules.

DRF [23] and HUG [13] provide max-main fair link sharing in a multi-resource setting with correlated demands. DRF considers static and fixed demands. HUG also allows for elastic demands.

### D. Time Separation

Some systems like Cicade [35] accept the need for handling the varying nature of tenant traffic instead of relying only on the average demand. Alternatively, scheduling the MapReduce shuffle stages was proposed by Orchestra [16]. This approach was generalized to allow a tenant to describe its changing communication needs using coflows [15], [52], [59]. In particular, Utopia [52] considers the tradeoff between minimizing the average coflow completion time and providing optimal isolation between contending coflows. However, since these schemes propose a fair-share network bandwidth to the current set of applications, they actually change the performance of a tenant when new tenants are introduced and can increase tenant performance variability.

### E. Tenant Resource Allocation

Cloud network performance has received significant attention over the last few years. An overview of the different proposals to allocate tenant network resources is provided by [39]. In addition, a survey [18] specifically focuses on network isolation solutions for multi-tenant data centers.

Virtual Network Embedding maps tenants' requested topologies and traffic matrix over arbitrary clusters [14]. However, these schemes require that tenants know in advance their exact traffic demands and cand suffer from scalability challenges due to relying on solutions for linear programs.

Other proposals, such as Topology Switching and Oktopus [11], propose an abstraction for the topology and traffic demands to be allocated to the tenants. They are similar to the hose model proposed for Virtual Private Networks in the WAN context [7]. In addition, [9] aims to provide a feedback-based fair-share bandwidth using edge-based rate-limiting. However, to guarantee tenant latency predictability and isolation, such solutions need strict packet time-pacing, small limits on allowed VM bandwidth and burst-size allocation [29].

Another approach for isolation may rely on distributed rate limiting like NetShare [36], SecondNet [25] and Seawall [51]. Employing that at the network edge requires tenant-wide coordination to avoid bottlenecks due to load-imbalance. This coordination leads to response time in the order of milliseconds [30], while the life time of a traffic pattern for high-demanding applications may be 2 to 3 orders of magnitude shorter.

### F. Application-Based Routing

The above schemes for network resource allocation ignore the fact that each tenant application may perform best with

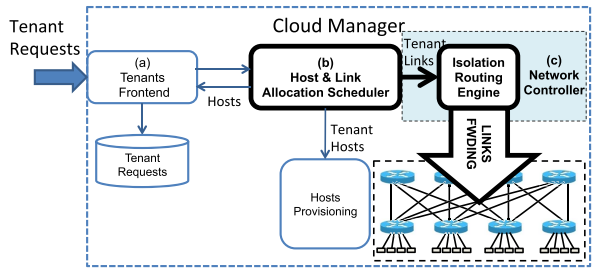



Fig. 2. Cloud management system architecture, with LaaS extensions in bold. First, in (a) a front-end interface collects tenant requests with information on the desired host number. Then, in (b) a scheduler allocates hosts and links for the tenants. Last, in (c) a network controller performs the network setup for a tenant based on its allocated hosts and links.

a different routing scheme. Routing algorithm types span a wide range. Some are completely static and optimized for MPI applications [24]. Others rely on traffic spraying as in RPS or DeTail [19], [58], on per-packet synchronized schemes like FastPass [44], on sampling [53], on programmable data planes [12], or on the virtual edge [34]. Additional algorithms can offer intra-application and potentially inter-application contention-free routing, but without any guarantees as in LaaS [47]. LaaS isolates the sub-topology of each tenant, and therefore allows each tenant to use the routing that maximizes its application performance. Without link isolation, routing engines must continuously coordinate the actual bandwidth each one of them utilize from each link.

### G. Fog and Edge Computing

Smaller clouds in the emerging fog and edge computing paradigms [48], [50] can similarly adopt LaaS models for isolation and performance guarantees. In fact, some of our theoretical guarantees specifically apply to small fat trees only.

## III. LaaS Architecture

A typical cloud architecture depicted in Fig. 2 consists of (a) a *front-end interface* for tenants to register their requests, (b) a *scheduler* that decides when and how to service these requests and can allocate hosts to tenants (*e.g.,* an OpenStack Nova scheduler and a Heat application setup), and (c) a *network controller* that performs the network setup (*e.g.,* an OpenStack Neutron and an SDN back-end). In this section, we introduce a *LaaS cloud architecture* that enhances this architecture by enabling the allocation of tenant-exclusive hosts and links.

Specifically, we propose to extend the *scheduler* with link allocation functionality (on top of the host allocation), and enhance the *network controller* by adding network routing rules to enforce the link allocation. Fig. 2 emphasizes these two extensions by bold lines on an abstract cloud management software architecture.

### A. Scheduler

We require the scheduler to provide each new tenant with an exclusive set of *dedicated hosts* and *dedicated links*. As in bare-metal allocation, a tenant may request a given number of *dedicated hosts*, which may be further refined by requirements of memory, accelerators or number of cores. In our implementation, we assume homogeneous hosts. In addition, the scheduler provides each new tenant with a set of *dedicated links* that form a tenant sub-topology, which will guarantee full bandwidth for any admissible traffic matrix of the tenant, i.e. will provide the tenant with the same bandwidth as in its own private data center.

In the LaaS architecture, we assume that the scheduler employs an online algorithm, by successively processing one new tenant request at a time. Each new tenant may be either accepted to the cloud, or denied due to the unavailability of a sub-network that can provide enough dedicated hosts and links. In any case, the scheduler does not migrate already-running tenants. This could be relaxed if we want to allow global optimization of host placements, by running tenants over virtual machines (VMs) and allowing migrations [31], [54]. But then, tenant run-times would be impacted by the arrival of new tenants, which is precisely what we want to avoid.

### B. Network Controller

As depicted in Fig. 2, we require the information of the allocated links to be provided by the scheduler to the network devices. This information should be used to adjust the network forwarding and routing to provide tenant isolation. This task fits SDN networks, but may also be implemented in other network architectures like TRILL [43]. There are several different ways to implement such an isolation-aware network controller. At one extreme, which requires switch-virtualization hardware support, a master controller may configure the underlying switches to be split into multiple virtual switches [49]. Then each tenant may incorporate its own SDN controller, which can then only discover its own isolated sub-topology. Another approach is to let a single SDN controller do all the work and enhance all the routing engines to work on sub-topologies. We rely in our implementation on an off-the-shelf InfiniBand SDN controller with a capability of defining sub-topologies and routing packets in an isolated manner (L2 forwarding). This feature, known as Routing Chains, is described in [2]. This isolated-routing feature could also be implemented by Ethernet SDN controllers like OpenDaylight.

## IV. Analyzing LaaS Requirements

In this section we introduce the analytical basis for providing LaaS. We first describe necessary and sufficient conditions on the assignment of hosts and links to the various tenants that can guarantee the LaaS properties. Later in this section, we also examine the possibility to satisfy these conditions in scenarios of low resource redundancy.

In this section, we describe online algorithms for *tenant placement* and *link allocation* in the LaaS scheduler. Online placement algorithms require the existing tenant placement to be maintained when a new job is placed, and therefore do not move existing tenants. Similarly we provide online link-allocation algorithms to avoid any traffic interruption when a new tenant is introduced. The algorithm we describe provably guarantees that a tenant will obtain a dedicated set of hosts and links, with the same bandwidth as in its own private

data center. The algorithm relies on the required properties of the placement to trim the solution space and achieve fast results.

We first study 2-level fat-trees, and then generalize the results to 3 levels. We first present a *Simple* heuristic algorithm, and then extend it with a *LaaS* algorithm that achieves a better cloud utilization.

### A. Isolation for 2-Level Fat Trees

Consider a 2-level full-bisectional-bandwidth fat-tree topology, i.e. a Full Bipartite Graph between leaf switches and spine switches, as in Fig. 1 above. For brevity we denote Full Bipartite Graphs that make the fat-tree connections between switches at levels $lvl_i$ and $lvl_{i+1}$: $FBG_i$. It is composed of $r$ leaf switches, denoted $L_i$ for each $i \in [1, r]$, and $m$ spine switches. Each leaf switch is connected to $n \leq m$ hosts as required to meet the rearrangeably non-blocking condition for fat-trees [28].

*1) Problem Definition:* Given a pre-allocation of tenants (with pre-assigned links and hosts), when a new tenant arrives with a request for $N$ hosts, we need to find:
*(i) Host Placement:* Find which free hosts to allocate to the new tenant, i.e. allocate $N_i$ free hosts in each leaf $i$ such that $N = \sum_{i=1}^{r} N_i$.
*(ii) Link Allocation:* Find how to support the tenant traffic, i.e. allocate a set $S_i$ of spines for each leaf $i$, such that the hosts of the new tenant in leaf $i$ can exclusively use the links to $S_i$, and the resulting allocation can fully service any admissible traffic matrix.

We want to fit as many arriving tenants as possible into the cloud such that their host placement and link allocation obey the above requirements, and without changing pre-existing tenant allocations.

*2) Simple Heuristic Algorithm:* We first introduce a *Simple* heuristic algorithm, as basis for the discussion of our algorithm. It relies on a property of fat-trees and minimum-hop routing: if a single tenant is placed within a subtree, then traffic from other tenants will not be routed through that subtree. Note that for 2-level fat-trees a subtree is a leaf switch.

Let $N$ denote the number of tenant hosts, and $n$ the number of hosts per leaf. The *Simple* heuristic simply computes the minimal number $s$ of leaf switches required for the tenant: $s = \lceil N/n \rceil$. Then, it finds $s$ empty leaf switches to place the tenant hosts in. Finally, if $s > 1$, it allocates all the up-links leaving the $s$ leaf switches; else, no such links are needed.

In the general case, any placement obtained by *Simple* supports any admissible traffic pattern. This is because the dedicated sub-network of the tenant is a single leaf switch if $s = 1$, and a 2-level fat-tree if $s > 1$, which is a folded-Clos network with $m \geq n$. It is well known that such a topology supports any admissible traffic pattern, because it meets the rearrangeable non-blocking criteria and the Birkhoff-von Neumann doubly-stochastic matrix-decomposition theorem [28].

*3) LaaS Placement Analysis:* This section describes a required condition on placement and sufficient condition on link allocation that are key to make the LaaS algorithm correct and efficient. The placement condition requires the allocation of $N$ tenant hosts as $Q$ leaves of $D$ hosts and optionally additional leaf of $R \mid R < D$ hosts such that $N = QD + R$. The sufficient link allocation condition requires the links of $R$ spines connecting to the $Q$ leaves and the optional single leaf of $R$ hosts. A subset of size $D - R$ of these spines should connect just to the $Q$ leaves.

Consider a single leaf $i$ with $N_i$ tenant hosts. In the analysis below, we make the following simplifying assumption: on every leaf switch, the number of leaf-to-spine links (and the corresponding number of spines) allocated to a tenant equals the number of its allocated hosts:

$$|S_i| = N_i. \tag{1}$$

Our simplifying assumption is based on the following intuition. On the one hand, for tenants occupying several leaves, if $|S_i| < N_i$, we may not be able to service all admissible traffic demands (since we may have up to $N_i$ flows that need to exit leaf $i$, but only $|S_i|$ links to service them). On the other hand, allocating $|S_i| > N_i$, is wasteful, because the number of remaining spine switches would then be less than the number of available hosts, and therefore future tenants spanning more than one leaf may not be able to obtain enough links to connect their hosts.

Without loss of generality, we also make a notational assumption that the $N_i$'s are sorted such that $0 < N_1 \leq N_2 \leq \cdots \leq N_t$, where $t$ is the number of leaves connected to hosts allocated to the tenant.

We will now see that our assumptions lead (by a sequence of lemmas) to a simple rule that greatly simplifies the possible placements that need to be evaluated by our LaaS scheduling algorithm.

*Lemma 1:* The number of common spines that connect two leaves must at least equal their minimal number of allocated hosts: $\forall i < j \in [1, t] : N_i = \min(N_i, N_j) \leq |S_i \cap S_j|$.

*Lemma 2:* The number of common spines that connect two leaves to a third must at least equal the minimal number of allocated hosts, either in the union of the first two leaves or in the third, i.e. $\forall i, j, k \in [1, t] : \min(N_i + N_j, N_k) \leq |S_i \cup S_j|$.

*Lemma 3:* The number of allocated hosts in any leaf cannot exceed the number in the union of any two other leaves, i.e. $\forall i \neq j \neq k \in [1, t] : N_i, N_j, N_k > 0 \rightarrow N_i + N_j \geq N_k$

*4) Necessary Host Placement:* We will now provide two theorems showing necessary and sufficient conditions to get the LaaS conditions of tenant traffic isolation and support for any admissible traffic matrix. Interestingly, the first theorem requires *necessary conditions on the host placement*, while the second theorem provides *sufficient conditions on the link allocation*. We continue to assume throughout the rest of the paper that $|S_i| = N_i$ for all $i$, and $N_1 \leq N_2 \leq \cdots \leq N_t$.

*5) Sufficient Link Allocation:* The following theorem suggests sufficient conditions on the link allocation to satisfy LaaS.

*Theorem 1:* A necessary and sufficient condition for LaaS is that the link allocation satisfies $\forall i \in [1, Q] : S_i = S^D$ and if $R > 0 : S^R \subset S^D$, i.e. all the allocated leaf up-links of a given tenant go to the exact same set of spine switches (or a subset of it for the remainder leaf).
In addition, we obtain a necessary condition:

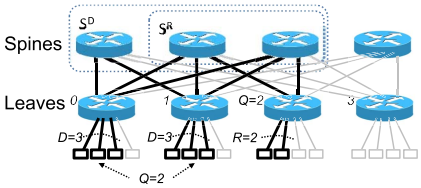



Fig. 3. Host allocation constraints illustration. To implement LaaS, for each tenant $i$, there must be $Q_i$ leaves of $D_i$ hosts and optionally one leaf of $R_i < D_i$ hosts such that the equality $N_i = Q_i \cdot D_i + R_i$ holds. A tenant of $N = Q \cdot D + R = 2 \cdot 3 + 2 = 8$ hosts appears in black.

*Theorem 2:* A necessary condition for LaaS is

$$N_1 \leq N_2 = N_3 = \cdots = N_t, \tag{2}$$

implying that all leaf switches of a tenant should hold the exact same number of hosts except for a potential smaller one.

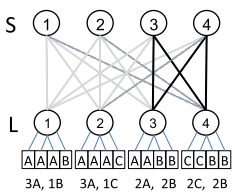
Given Theorem 2, the tenant placement should follow the form: $N = Q \cdot D + R$, where $Q$ is the number of repeated leaves with $D$ hosts each, and we optionally add one unique leaf with a smaller number of hosts $R$. This notation follows the Divisor, Quotient and Remainder of $N$. This result is useful because it greatly simplifies the solution of the host placement problem defined above.

Fig. 3 demonstrates this result. It shows $Q$ leaf switches of $D$ hosts each, and optionally another leaf switch of $R < D$ hosts. We denote by $S^D$ the set of spines connected by allocated links to the $Q$ leaves of $D$ hosts, and by $S^R$ those that connect via allocated links to the optional leaf of $R$ hosts.

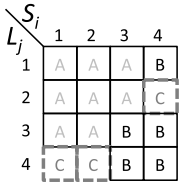
*Proof:* If $R = 0$, there is a group of $D$ spine switches that connect to all leaf switches. Thus the tenant sub-topology reduces to a *Full Bipartite Graph* $(FBG)$ with $D$ spine switches and the same number $D$ hosts per leaf. Such a topology is rearrangeable non-blocking known to support any admissible traffic matrix. If there is leaf $L_{j_R}$ of $R > 0$ hosts, we provide a constructive method for routing arbitrary permutations. We consider the $FBG$ sub-topology formed by the tenant hosts and links, where $L_{j_R}$ connects to all $S^D$ spines. For this topology $D$ spines, $D$ hosts per leaf and $Q+1$ leaves. Again, every full permutation of $D \cdot (Q+1)$ hosts is routeable. Consider a traffic permutation in the original topology. We can add to it traffic between $D - R$ dummy hosts in a single leaf. The traffic between the dummy hosts must be routed through $D - R$ distinct spines. The other spines are equivalent to $S^R$ and by symmetry the given traffic permutation could be routed in the partial topology. The necessity of the condition immediately follows the necessity of the condition in Theorem 2 with the result of Lemma 1. ■

*6) Counter-Example:* We now show that there is indeed a gap between the necessary conditions on host placement in Theorem 2 and the necessary and sufficient conditions on link allocation in Theorem 1. Namely, we exhibit a counter-example that satisfies the conditions of Theorem 2 but still does not guarantee LaaS.

*Lemma 4:* There are host placements that meet Theorem 2 for which there does not exist a link allocation satisfying LaaS.




(a) Placement




(b) Link Allocation

Fig. 4. Satisfying host allocation constraints is not enough. A joint host placement and link allocation is necessary for LaaS. (a) All tenants satisfy the host placement necessary conditions, *e.g.,* C includes $3 = Q \cdot D + R = 1 \cdot 2 + 1$ hosts. A and B support any admissible traffic matrix by the sufficient link allocation conditions. (b) However, the link allocation for C is impossible. There is no way to find a common set of spines with free ports.

### *B. Achieving LaaS Without Any Reduction in Utilization*

As shown in Lemma 4, some tenant requests may be denied when the scheduler cannot find a proper host placement and link allocation that satisfy the LaaS conditions. Indeed, sometimes a tenant can be denied even when the fat-tree has the number of available hosts as required by the tenant, i.e., when a host placement can be found but not a corresponding link allocation. A particularly challenging assignment task is when the total number of hosts required by the tenants exactly equals the number of hosts in the fat-tree. Based on the fat-tree size, the number of tenants and their requests, we demonstrate that are there some scenarios for which the existence of an allocation is guaranteed and there is no reduction in utilization for achieving that.

To simplify the presentation, we describe a host and a link assignment through a matrix of size $r \times m$ where $r$ and $m$ are the number of leaf and spines switches, respectively. A value in the matrix describes the tenant assigned to a link between a leaf switch and a spine. A matrix row shows the tenants assigned to one leaf switch. An example that follows the hosts assignment from Fig. 4(a) is described in Fig. 4(b). We also consider a family of two-level fat-trees, where we define the *subtree of parameter* $r \in \mathbb{N}^+$ as a fat-tree with $r$ switch leaves, fully connected to $m = r$ spines, such that $r$ hosts can be assigned to each leaf. We also define the *fat-tree size* as its number of hosts $r^2$. For instance, the fat-tree of parameter $r = 4$ is illustrated in Fig. 3.

The next theorem claims that for $r \leq 5$, when the required host number fits the fat-tree size, an assignment satisfying LaaS always exists *for any number of tenants and for any combination of the tenant sizes*.

*Theorem 3:* Consider a fat-tree of a parameter $r \leq 5$. For any tenant demands with a total of at most $r^2$ required hosts, there exists an assignment that satisfies LaaS.

While for $r \geq 6$ we do not provide the equivalent guarantee, we show that this can be provided when an assumption on the size of the tenants holds. Intuitively, when $r$ increases, it is harder to bound the number of possibly large tenants. We see the question whether Theorem 3 holds for any $r \in \mathbb{N}^+$ as an open problem.

*Theorem 4:* Consider a fat-tree of an arbitrary parameter $r$. For any tenant demand with a total number of required hosts

not larger than the fat-tree size, where at most two tenants have a demand larger than $r+1$, there exists an assignment that satisfies LaaS.

While satisfying the above conditions guarantees the existence of a LaaS assignment without any utilization reduction, they do not necessarily hold in the general case. Accordingly, in Section VI we show that in practical scenarios the implied utilization reduction is often small.

### C. Achieving LaaS With a Bounded Reduction in Utilization

We now consider a more general scenario in which utilization reduction might be a necessity. We describe guarantees to bound the reduction as a function of the tenant sizes. Intuitively, we show that a higher utilization can be guaranteed in each of the two following cases: *(i)* A large portion of the input tenants are of small size (lower than some threshold), and/or *(ii)* Among the large tenants (of at least the threshold), there are some of size much larger than the threshold.

As in the above we refer to a fat-tree of parameter $r$ with $r^2$ hosts. The largest set of tenants in a legal LaaS assignment includes a total of $r^2$ hosts. We refer to a tenant as *small* if its requested host number is at most $r+1$ and as *large* otherwise. We first describe a guarantee to find an assignment for a demand composed of a dominant portion of small tenants.

*Theorem 5:* Consider a set of requested tenants composed of: *(i)* A proportion $\alpha$ of hosts for small tenants, *i.e.,* a total of $\alpha \cdot r^2$ requested hosts that belong to small tenants, each of at most $r+1$ hosts. *(ii)* Additional $(1-\alpha)/2 \cdot r^2$ requested hosts that belong to large tenants. Then, a legal LaaS assignment exists for the tenants within a fat-tree of parameter $r$ with $r^2$ hosts. The assignment utilizes a proportion $\alpha + (1-\alpha)/2 = (1+\alpha)/2$ of the fat-tree hosts.

The above guarantee makes no assumption on the size of the request for the large tenants (beyond the fact they are indeed large). In the following, we explain that a stronger guarantee can be derived as a function of the properties of the large demands.

*Theorem 6:* Consider a set of requested tenants composed of: *(i)* A total of $\alpha \cdot r^2$ requested hosts that belong to small tenants, each of at most $r+1$ hosts. *(ii)* Additional $\left\lfloor (1-\alpha) \cdot \frac{\beta}{\beta+1} \cdot r^2 \right\rfloor - (r-1)$ requested hosts that belong to tenants, each of size at least $\beta \cdot r$ for some $\beta \in \mathbb{N}, \beta \geq 2$. Then, a legal LaaS assignment exists for the tenants within a fat-tree of parameter $r$ with $r^2$ hosts.

## V. Isolation for 3-Level Fat Trees and Algorithms

So far we have discussed the LaaS allocation for 2-level fat-trees. We now extend the results to 3-level fat-trees, which form the most common cloud topology [6], [8]. We use the notation of Extended Generalized Fat Trees (XGFT) [40], which defines fat-trees of $h$ levels and the number of subtrees at each level: $m_1, m_2, \ldots, m_h$ and the number of parent switches at each level: $w_1, w_2, \ldots, w_h$.

According to Lemma 4, some tenant requests may be denied because the scheduler cannot find a proper link allocation. Thus any LaaS algorithm has to validate the feasibility of a link allocation for each legal host placement.

We consider three approaches to this problem: a *Simple* heuristic, a *Hierarchical* decomposition, and an *Approximated* scheme. We conclude with describing the final *LaaS* algorithm that we implemented, relying on the *Approximated* scheme.

### A. Simple Heuristic for 3-Level Fat-Trees

The *Simple* algorithm described in sub-section 'Simple heuristic algorithm' is easily extended to any fat-tree size. For an arbitrary XGFT, first define the number of hosts $R_l$ under a subtree of level $l$: $R_0 = 0$, and $R_l = \prod_{i=1}^{l} m_i$. Given a tenant request for $N$ hosts, *Simple* first determines the minimum level $l_{min}$ of the tree that can contain all $N$ tenant hosts:

$$l_{min} = \min \{l | (R_l \geq N)\} \tag{3}$$

and the number $s$ of required subtrees of level $l_{min}$: $s = \lceil N/R_{l_{min}-1} \rceil$. Then, it places the tenant hosts in $s$ free subtrees of level $l_{min}$. It also allocates to the tenant all the links internal to these $s$ subtrees; and if $s > 1$, it allocates as well all the links connecting the subtrees to the upper level.

It is clear that the *Simple* heuristic algorithm, by rounding up the number of nodes, trades off cluster utilization for simplicity, non-fragmentation, and greater locality with lower hop distances. As we show in the evaluation section, the utilization obtained by this algorithm is low, making it potentially unacceptable to cloud vendors, so we keep looking for a better one.

### B. Hierarchical Decomposition

In this section we describe how LaaS can be provided to a 3-level fat-tree using a hierarchical decomposition approach following the recursive description of fat-trees in [45].

As we showed in the previous sections, since the tenant traffic pattern may be completely contained within each 2-level tree, host allocation in each 2-level tree must adhere to Theorem 2. So the number of tenant hosts within the 2-level subtree $j$ must be of the form $N_j = Q_j \cdot D_j + R_j$. Note that an allocation that fits in a single leaf switch also follows this scheme with $Q_j = 1$.

When we consider the conditions required for the highlighted $FBG_2$ to support any admissible traffic pattern, it is strikingly similar to the analysis we provided for the 2-level fat-tree. For the 2-level tree we already proved that in order to support any admissible traffic pattern, the sequence of $U_j$ values must meet the rule $U_1 \leq U_2 = U_3 = \ldots = U_{m_3}$. Applying the same to the 3-level tree we obtain a requirement for the assignments of $U_j$ on each of the $FBG_2$. However, each one of the $FBG_1$ (there are $m_3$ such 2-level subtrees) could select a different set of $S_j^D$ and $S_j^R$. This means that a solution could allow each 2-level subtree to select a different set of $FBG_2$ to carry its flows, as long as the above rule is maintained for each $FBG_2$.

Fig. 5 shows an example of 3-level fat-tree. We denote the switches on the tree by their levels (from bottom up) $lvl_1$, $lvl_2$ and $lvl_3$. For a LaaS link allocation to be feasible, the condition of Theorem 2 needs to hold not only for

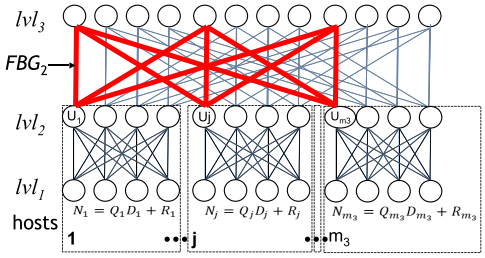



Fig. 5. 3-level fat-trees with levels $lvl_1$, $lvl_2$, $lvl_3$. Constraints on the allocated hosts similar to those of a 2-level fat-tree appear between $lvl_1$ - $lvl_2$ as well as between $lvl_2$ - $lvl_3$. In each of the $m_3$ subtrees of $lvl_1$ - $lvl_2$, a tenant is allocated a number of hosts of the form $N_j = Q_j \cdot D_j + R_j$. Similarly, in $lvl_2$ - $lvl_3$ the implied full bipartite graphs (such as the graph illustrated in red) should satisfy similar properties based on the number $U_i$, the number of flows of the tenant connected to some node in $lvl_2$.

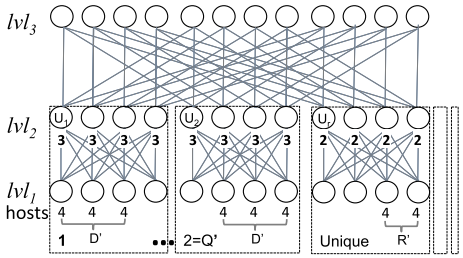



Fig. 6. An example of host placement with $N = 32$ hosts on a 3-level fat-tree using the *approximated* method. Using a notation similar to the 2-level fat-tree, this allocation is of the form: $Q' = 2$, $D' = 3$ and $R' = 2$. Here, flows of three hosts are connected each $U_1$ and $U_2$ and two to $U_r$ satisfying $U_1 = U_2 > U_r$.

each $lvl_1$-$lvl_2$ 2-level subtree but also for each $lvl_2$-$lvl_3$ Full Bipartite Graph ($FBG_2$) at the top of the tree. One of these $FBG$s is highlighted in Fig. 5 and the values of $U_j$ should satisfy $U_1 \leq U_2 = U_3 = \cdots = U_{m_3}$ for some order of them.

Fig. 5 depicts a Theorem 2-compliant host allocation within each of the 2-level $lvl_1$-$lvl_2$ subtrees. Within each subtree $j \in [1, \ldots, m_3]$, the number of allocated hosts follows the form: $N_j = Q_j \cdot D_j + R_j | j \in \{1 \ldots m_3\}$. Within that subtree, these hosts are connected to some spines in $lvl_2$. Moreover, the link assignment within the 2-level $lvl_1$-$lvl_2$ subtrees must also adhere to Theorem 1 such that $S_j^R \subset S_j^D$. Consequently, the maximum number $U_j$ of flows leaving the 2-level subtree from switch $s$ can be either 0 in case $s \notin S_j^D$, $Q_j$ in case $s \in S_j^D \backslash S_j^R$, or $Q_j + 1$ if $s \in S_j^R$. Accordingly, the constraint on the values $U_1, \ldots, U_{m_3}$ implies a constraint on the possible values of hosts in each of the $m_3$ $lvl_1$-$lvl_2$ subtrees.

Unfortunately the above rule still allows a vast amount of legal tenant-placement and link-allocation possibilities, which make the full 3-level fat-tree LaaS problem too hard to be solved optimally in practical time even on high-end processors. Luckily, our target is to show that there is a simple enough algorithm that would be able to handle the online LaaS problem in reasonable time and with reasonable success rate such that the cluster utilization remains high and LaaS is guaranteed. We do that by applying a restriction on the solution space of the hierarchical decomposition.

### C. Approximated Algorithm

We provide a simpler algorithm that compromises cluster utilization in favor of reduction of the solution search space. Our approximation requires the allocation to be symmetrical with respect to all the $FBG_2$, i.e. that the allocation on all the $FBG_2$ is identical and thus calculated just once. So the solution must use the same number of flows $U_j$ leaving any one of the $lvl_2$ switches in the same 2-level subtree. Note that any allocation where the number of tenant hosts $N_i$ connected to leaf switch $i$ does not include all the hosts on that leaf switch $N_i < m_1$, will not utilize all the links from that switch to the upper-level switches. So only a subset of the $lvl_2$ switches in the same $FBG_1$ is going to pass traffic of that tenant. Thus if we now consider the $lvl_2$ to $lvl_3$ traffic, not all $FBG_2$ will see the same $U_j$. To avoid this we require that $D$ is either 0 or $m_1$ for all 2-level subtrees, except where the tenant fits within the same 2-level fat-tree and thus $U_j = 0$. As a consequence, if a tenant cannot fit within a single subtree, we round up its size to a multiple of $m_1$. The host placement can now be performed in complete leaf switches of $m_1$ hosts. For instance, if each leaf switch can hold 10 hosts, and a tenant requests $N = 267$ hosts, then we effectively allocate it $N' = m_1 \lceil N/m_1 \rceil = 270$ hosts.

Moreover, since the approximation in 3-level fat-tree allocates complete $lvl_1$ switches, it is equivalent to the 2-level LaaS problem: $lvl_1$ switches are equivalent to hosts, $lvl_2$ switches are like leaf switches and $lvl_3$ switches are like spines. Thus the approximated 3-level fat-tree LaaS problem has to comply to the same conditions as for the 2-level tree. We denote the allocation of full $lvl_1$ switches using a similar notation to the 2-level: $Q'$ is the number of allocated 2-level subtrees, each with $D' = Q$ leaves. Optionally there may be one additional 2-level subtree with $R'$ allocated leaves. $N' = \lceil N/m_1 \rceil = Q' \cdot D' + R'$.

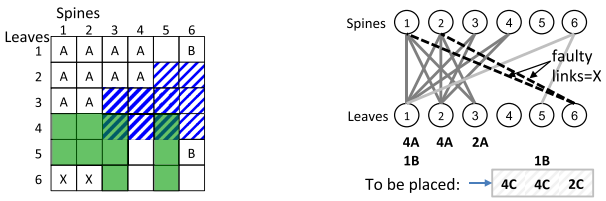
An example of such allocation for a tenant of 32 hosts on a 3-level fat-tree, with 4 hosts per leaf, is provided in Fig. 6. On the left $Q' = 2$ subtrees, the tenant uses $D' = 3$ leaves and thus $U_1 = U_2 = 3$ for all $FBG_2$. In addition a single unique subtree $r$ with $R' = 2$ leaves is also allocated and $U_r = 2$ for all $FBG_2$. So all the $FBG_2$ are thus identical. Each one of them has to support $Q'$ $lvl_2$ switches of $D' = 3$ flows and one $lvl_2$ switch with $R' = 2$ flows. These requirements meet the condition of Theorem 2 and thus may be feasible.

### D. LaaS Algorithm

We now want to implement our final *LaaS* algorithm for concurrent host placement and link allocation in fat-trees. To do so, we rely on our *Approximated* approach, and track the allocated up-links in a matrix similar to Fig. 7(a). The required set of leaves and links is of the form $N = Q \cdot D + R$. Following the analysis in Section IV-A, in a general fat-tree, this translates to $R$ spines that connect to all the $Q+1$ allocated leaves and $D - R$ spines connected just to the $Q$ repeated leaves. These requirements are equivalent to finding a set of $Q$ leaves that have $D$ free up-ports to a common set of spines, and a single leaf that has only $R$ free up-ports that form a subset of the spines used by the previous $Q$ leaves.

**Algorithm 1** LAAS($N$)

```
// Try 1 level allocation
if N ≤ m1 then
    for l = 0 to m2 · m3 − 1 do
        if FLAP(N, 1, 0, l, l, 0, {}, {}) then
            return true
// Try 2 level allocation
if N ≤ m1 · m2 then
    for D = max(N, m1) to 1 do
        Q = ⌊N/D⌋; R = N − Q · D
        for l = 0 to m3 − 1 do
            if FLAP(D, Q, R, l·m2, (l+1)·m2−1, 0, {}, {})
                then
                return true
// Try 3 level allocation
U = ⌈N/m1⌉
for D = max(U, m2) to 1 do
    Q = ⌊U/D⌋; R = U − Q · D
    if Q ≤ m3 then
        if FLAP2(D, Q, R, 0, m3 − 1, 0, {}, {}) then
            return true
return false
```




(a) Link Allocation Table (b) Corresponding Topology

Fig. 7. Algorithm illustration: Given are two allocated tenants A and B, with the existence of faulty links X. A third tenant C of 10 hosts needs to be mapped: (a) shows the allocation of each link connecting leaves and spines, and (b) displays the corresponding topology. Two possible allocations for tenant C appear in (a), the first in green solid filling, and the second with slanted blue lines. In both, the $10 = 2 \cdot 4 + 2$ requested hosts are allocated into three leaves with 4, 4, and 2 hosts in each.

The search for $Q$ leaves with enough common spines is performed recursively. It may require examining at most all $\binom{m_2}{Q}$ combinations. Our *LaaS* algorithm returns the first successful allocation, so trying the most-used leaves first packs the allocations and achieves the best overall utilization results.

Fig. 7 demonstrates the process of evaluating a specific $D, Q, R$ division. Consider a new tenant C of 10 hosts, arranged as 2 leaves of 4 hosts plus 1 leaf of 2 hosts. We show 2 possible placements: The first would use 4 hosts on leaves 4 and 5, and 2 hosts on another leaf 6. The second would use 4 hosts on leaves 3 and 4, and 2 hosts on another leaf 2. We also illustrate how we could take into account two faulty links in our link allocation if needed.

In the following section we describe the algorithm for mapping free leaves. We refer to a method that calculates the above examples as FLAP. The recursive function is assuming the availability of matrix $M[l]$ of free ports on each leaf switch. It is given the following constants: $D, R, Q$ and the start and end leaf switch indexes $l_s, l_e$. The recursive function provides its current state on the recursion using the following variables: $l$ represents the current leaf index to examine, $r$ the number of $Q$ size leaves that were already found, $\{ports\}$ the set of ports that are possible for this allocation, $\{rl\}$ the collected set of, so far, $Q$ size leaves. Eventually the recursion provides the following results: $\{D_L\}$ set of leaves with $Q$ hosts, $\{D_{PORTS}\}$ the set of ports to be used by the $Q$ size leaves, $U_L$ the unique, sized $R$, leaf and $\{U_{PORTS}\}$ the ports on that leaf. The higher level algorithm considering the possible valid combinations of $Q, D$ and $R$, for 2-level and 3-level fat-trees is provided in Algorithm 1.

## VI. EVALUATION

Our evaluation is reported in three sub-sections. The first deals with the resulting *cloud utilization* when applying LaaS conditions. It shows that our *LaaS* algorithm reaches a reasonable cloud utilization, within about $10\%$ of bare-metal allocation. The second part describes the system implementation on top of OpenStack, and the third part shows how the LaaS architecture improves the performance of a tenant in the presence of other tenants through a complete tenant isolation.

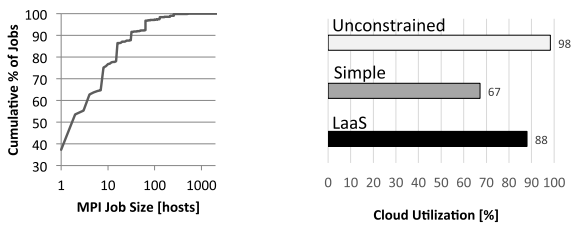
### A. Evaluation of Cloud Utilization

*1) Cloud Utilization:* We want to study whether our LaaS network isolation constraints significantly reduce the number of hosts that can be allocated to tenants. We define the *cloud utilization* as the average percentage of allocated hosts in steady state. Assuming that tenants pay a fee proportional to the number of used hosts and the time used, the cloud utilization is a direct measure of the cloud provider revenue.

*2) Scheduling Simulator:* To evaluate the different heuristics on large-scale clouds, we developed a scheduling simulator that runs many tenant requests over a user-defined topology. The simulator is configured to run any of the above algorithms for host and link allocation. This algorithm may succeed and place the tenant, or fail. We use a strict FIFO scheduling, i.e. when a tenant fails, it blocks the entire queue of upcoming tenants. Note that this blocking assumption forms an extremely conservative approach in terms of cloud utilization. In practice, clouds would typically not allow a single tenant to block the entire queue and use resource reservation with back-filling techniques to overcome such cases. Since smaller tenants are easier to place, for any tenant size distribution, not letting smaller tenants bypass those waiting means that we fill fewer tenants into the cloud. Thus, the result should be regarded as an intuitive lower-bound for a real-life cloud utilization.

*3) Settings:* We simulate the scheduler with LaaS algorithm on the largest full-bisectional-bandwidth 3-level fat-tree network that can be built with 36-port switches, i.e. a cloud of 11,664 hosts. The evaluation uses a randomized sequence of 10,000 tenant requests. A random run-time in the range of 20 to 3,000 time units is assigned to each tenant. The variation of run-time makes scheduling harder as it increases fragmentation.

We evaluate 2 distribution types for the number of hosts requested by the tenants. First, we randomly generate sizes




(a) (b)

Fig. 8. Cloud utilization evaluation with Julich JUROPA job scheduler traces. (a) Measured job-size cumulative distribution function (CDF), with peaks in sizes that are powers of two. (b) Resulting cloud utilization. *LaaS* achieves 88%. The *unconstrained* host allocation does not each 100% due to possible failure upon receiving a request larger than the number of currently available hosts. *Simple* trades off cluster utilization for simplicity, through rounding up the number of allocated nodes.

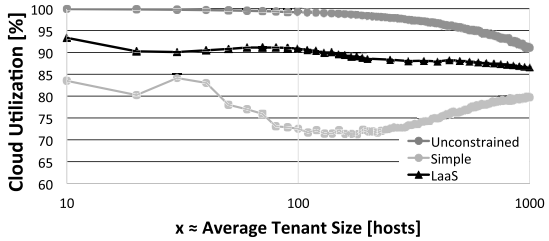



Fig. 9. Impact of request size on cloud utilization. The requested tenant size distribution is truncated according to various average tenant size following exponential distribution in a cloud of 11,664 hosts. In both *LaaS* and *unconstrained*, the utilization degrades with the tenant size but remains better than that of *simple*.

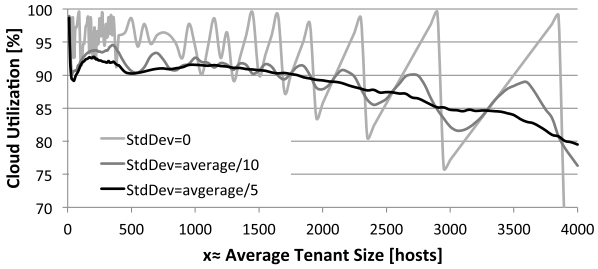



Fig. 10. Impact of request average size and its deviation on cloud utilization: A truncated Gaussian distribution $\mathcal{N}(x, \alpha)$ for tenant host sizes in a cloud of 11,664 hosts for $\alpha \in \{0, x/10, x/5\}$, where the average tenant size is $x \leq 4000$. Without deviation, we can see a sawtooth shape. When size deviations are positive, the larger deviations allow smmothness in curves.

according to a job size distribution extracted from the Julich JUROPA job scheduler traces. These previously-unpublished traces represent 1.5 years of activity (Jan. 2010 – June 2011) of a large high-performance scientific-computing cloud. Second, we use a truncated exponential distribution of variable average $x$. It is truncated between 1 and the cluster size.

In order to measure the utilization loss we fill the cluster with tenants by assuming all tenant requests are available at simulation start. Tenants' run-time is randomized with uniform distribution from 10 to 3000 time units.

As a baseline algorithm, we implement an *Unconstrained* placement approach that simply allocates unused hosts to the request, as in bare-metal allocation. Note that some requests may still fail if the tenant requests more hosts than the number of currently-free cloud hosts. We compare this baseline to the *Simple* and *LaaS* algorithms, as described in Section IV.

*4) Simulation Results:* Fig. 8(a) illustrates the Cumulative Distribution Function (CDF) of the tenant sizes (in number of hosts) collected from the Julich JUROPA cluster. The CDF shows peaks for numbers of hosts that are powers of 2 (1, 2, 4, 8, 16, and 32). We further generated 10,000 tenants with this job-size distribution, and the same random run-time distribution as above (instead of the original run-times, since they resulted in a low load, and therefore in an easy allocation). Fig. 8(b) shows the tenant allocation results: again, the cost of our *LaaS* allocation versus the *Unconstrained* bare-metal provisioning is about 10% of cloud utilization (88% vs. 98%).

To further test the sensitivity of our algorithm to the tenant sizes, we use a truncated exponential distribution for tenant host sizes and modify the exponential parameter $x$. The distribution of the JUROPA tenant sizes is similar to such a truncated exponential distribution. Fig. 9 illustrates the cloud utilization for *Unconstrained*, *Simple*, and *LaaS*, is plotted as a function of the exponential parameter $x$, which is close to the average tenant host size due to the truncation. The *Unconstrained* line shows how the utilization degrades with the job size, even without any network isolation. This is an expected behavior of bin packing. As the job size grows, so does the probability for more nodes to be left unassigned when the cloud is almost full. The utilization of our *LaaS* algorithm stays steadily at about 10% less than the *Unconstrained* algorithm. Finally, *Simple* has the lowest cloud utilization for the entire tenant size range. Note that it is less steady, since its utilization is more closely tied to the sizes of the leaves and subtrees. Once the tenant size crosses the leaf size (18 in our case), it is rounded up to a multiple of that number. Likewise, once it crosses the size of a complete subtree (324 hosts), it is rounded up to the nearest multiple of that number. These results show that our LaaS algorithm provides an efficient solution for avoiding tenant variability, as its cost is only about 10% for a wide range of tenant sizes.

*Simple* suffers from a particularly large fluctuation in utilization. *LaaS* is more stable over the entire range, with about 90% utilization. There are a few points where the *Simple* heuristic provides a better utilization than *LaaS*. But, note that utilization stability is key to cloud vendors, since changing the allocation algorithm dynamically would require predicting the future size distribution, and thus may produce worse results when the distribution does not behave as expected.

Fig. 10 plots the *LaaS Approximation* utilization for different spreads of tenant sizes around the average. We refer to the same cloud size of a total of 11,664 hosts. A standard deviation of $x/5, x/10$ and 0 are shown where an average tenant size of $x$ is considered up to roughly 4000. The zero deviation curve exhibits the expected saw-tooth shape that is caused by the fact that it is possible to get 100% utilization when the tenant size is a divisor of the number of nodes. As the deviation of the tenant sizes grows, so does the smoothness of the curve. This is common to all scheduling algorithms behavior providing the peaks and valleys around the job sizes crossing the single

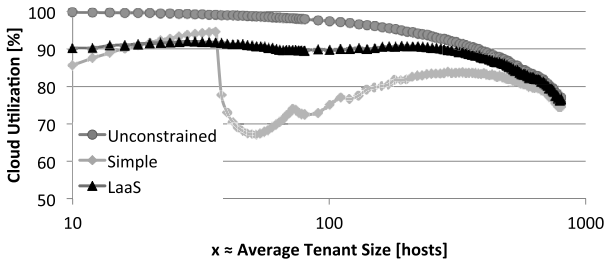



Fig. 11. Impact on cloud utilization of request average size selected with a uniform distribution. Cloud utilization vs. average tenant size for 10,000 requests with uniform (0.2×,1.8×) size distribution.

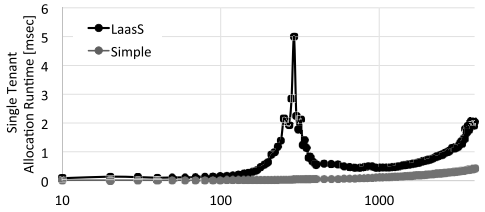



Fig. 12. Run-time of single tenant allocation vs. average tenant size. The average allocation time for a tenant is often smaller than 1 msec. It rises up to 5 ms for tenant sizes closer to 324 hosts. For slightly smaller tenant requests, a large number of allocations has to be considered in a single subtree.

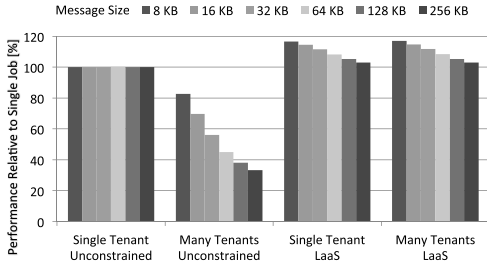



Fig. 13. Simulated relative performance for tenants running stencil scientific-computing applications on a cloud of 1,728 hosts, either alone or as 32 concurrent tenants. While tenant performance degrades when placed unconstrained (without link isolation), the performance of single and multiple tenants with LaaS appears identical, fulfilling the promise of LaaS.

leaf or subtree size. Most significantly, up to an average tenant size of several hundreds of hosts, the utilization typically stays above 90%, which is very encouraging for our algorithm.

Fig. 11 presents the utilization obtained for a uniform distribution of the tenant sizes with a large variance. As can be seen there is a clear advantage to the LaaS Placement heuristic that maintains a utilization of about 90% from the unconstrained host assignment. Moreover, for larger average tenant sizes, the performance of the LaaS Placement is even closer to that of the unconstrained.

### *B. System Implementation*

We implemented the LaaS architecture by extending the *OpenStack Nova scheduler* with a new service that first runs the LaaS host and link allocation algorithm, and then translates the resulting allocation to an SDN controller that enforces the link isolation via routing assignments.

*1) Host and Link Allocation:* The integration of the *LaaS* algorithm was done on top of OpenStack (Icecube release), utilizing filter type: *AggregateMultiTenancyIsolation*. This filter allows limiting tenant placement to a group of hosts declared as an "aggregate", which is allocated to the specific tenant id. Our automation, provided as a standalone service on top of OpenStack's *nova* controller, obtains new tenant requests, and then calls the *LaaS* allocation algorithm. If the allocation succeeds, we invoke the command to create a new aggregate that is further marked by the tenant id. The allocated hosts are then added to the aggregate. The filter guarantees that a new host request, conducted by a user that belongs to a specific tenant, is mapped to a host that belongs to the tenant aggregate.

*2) Network Controller:* We further implement a method to provide the link allocation to the InfiniBand SDN controller [3], which allows it to enforce the isolation by changing routing. The controller supports defining sub-topologies, by providing a file with a list of the switch ports and hosts that form each sub-topology. Then each sub-topology may have its own policy file that determines how it is routed.

*3) Run-Time:* The LaaS Approximation scans through all possible placements for valid link allocation. This involves evaluating all possible valid combinations of $R$ and $Q$ values. Fig. 12 presents the average run-time per tenant request for placing tenants on 11,664 nodes cluster providing a truncated exponential tenant size distribution. Run time was measured on an Intel® Xeon® CPU X5670 @ 2.93GHz. The peak in run-time of about 5 msec appears just below the average tenant size of 324, which is the exact point where our algorithm first scans all possible placements under a single subtree and continues with multiple subtree placement.

### *C. Evaluation of Tenant Performance*

Since LaaS guarantees tenant isolation, tenant performance should be independent of the number of other tenants that run on the same network. To demonstrate LaaS tenant isolation, we simulate a large cluster using a well known InfiniBand flit level simulator used for instance by [20].

Fig. 13 presents the relative performance of single and multiple tenants running Stencil scientific-computing applications on a cloud of 1,728 hosts, under either *Unconstrained* or *LaaS*, normalized by the performance of a single tenant placed without constraints. The figure illustrates many effects. First, the performance of a single tenant with *Unconstrained* significantly degrades when other tenants are active, *e.g.,* to 45% with 32-KB message sizes. This is because the bare-metal allocation of *Unconstrained* does not provide link isolation. Second, under our *LaaS* algorithm, *the single-tenant performance is not impacted when the other tenants become active* (the third and fourth sets of columns look identical). This was the key goal of this work. *LaaS* prevents any inter-tenant traffic contention. Finally, we can observe an additional surprising effect (first vs. third sets of columns): the tenant performance is slightly improved for small messages under *LaaS* versus the *Unconstrained* allocation. The reason is that *LaaS* does not accept tenants unless it can place them with no contention, and therefore the resulting placement tends to be tighter, thus improving the run-time performance with small

message sizes when the synchronization time of the tasks is not negligible. The lower network diameter of *LaaS* improves the synchronization time, which is latency-dominated.

## VII. Conclusions

In this paper, we demonstrated that the interference with other tenants causes a performance degradation in cloud applications that may exceed 65%, and argued that it is a hurdle to network softwarization in shared clouds. We introduced LaaS (Links as a Service), a novel cloud allocation and routing technology that provides each tenant with the same bandwidth as in its own private data center. We showed that LaaS completely eliminates the application performance degradation. We further explained how LaaS can be used in clouds today without any hardware change, and showed how it can rely on open-source software code we contributed. Finally, we also used tenant-size statistics of a large scientific-computing cloud, obtained over a long time period, to construct a random workload that illustrates how isolation is possible at the cost of some 10% cloud utilization loss.

We have analyzed how to find a LaaS assignment without any utilization reduction for fat-tree sizes of a bounded size. A natural open question is whether such a LaaS assignment without reduction exists for any fat-tree size, and if so, whether finding it can be performed by a polynomial-time algorithm. Another possible extension would be to generalize the isolation definition, letting any link to be shared by a bounded number of tenants. Understanding the tradeoff between potentially higher utilization and degraded performance for that generalization is also left to future work.

## VIII. Acknowledgments

The authors would like to appreciate our gratitude to our colleagues in the Technion and Mellanox Technologies for their support and enthusiasm about our approach.

## References

[1] *LaaS Source Code, Experiments and Simulation Conditions*. Accessed: Apr. 7, 2019. [Online]. Available: https://www.dropbox.com/sh/uzla7rcmdiqrxig/AADpw5ALG-q8VFzMSVcmYvmJa/

[2] *Mellanox OFED User Manual V2.3-1.0.1*. Accessed: Apr. 7, 2019. [Online]. Available: http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v2.3-1.0.1.pdf

[3] *OpenSM InfiniBand Open SDN Controller*, document, 2018. [Online]. Available: http://www.mellanox.com/related-docs/whitepapers/WP_InfiniBand_Production_SDN.pdf

[4] *OpenStack Ironic*. Accessed: Apr. 7, 2019. [Online]. Available: https://wiki.openstack.org/wiki/Ironic

[5] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6D mesh/torus interconnect for exascale computers," *IEEE Comput.*, vol. 42, no. 11, pp. 36–40, Nov. 2009.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM (SIGCOMM)*, Aug. 2008, pp. 63–74.

[7] A. Altin, H. Yaman, and M. C. Pinar, "The robust network loading problem under hose demand uncertainty: Formulation, polyhedral analysis, and computations," *J. Comput.*, vol. 23, no. 1, pp. 75–89, 2011.

[8] A. Andreyev, "Introducing data center fabric, the next-generation Facebook datacenter network," Facebook, Tech. Rep., Nov. 2014.

[9] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska, "End-to-end performance isolation through virtual datacenters," in *Proc. 11th Symp. Oper. Syst. Des. Implement.*, 2014, pp. 233–248.

[10] D. Artz, "The secret weapons of the AOL optimization team," Velocity Conf., San Jose, CA, USA, Tech. Rep., 2009.

[11] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proc. Comput. Commun. Rev. (SIGCOMM)*, Aug. 2011, pp. 242–253.

[12] C. H. Benet, A. J. Kassler, T. Benson, and G. Pongracz, "MP-HULA: Multipath transport aware load balancing using programmable data planes," in *Proc. Morning Workshop Netw. Comput.*, Aug. 2018, pp. 7–13.

[13] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *Proc. 13th Symp. Netw. Syst. Des. Implement.*, 2016, pp. 407–424.

[14] M. Chowdhury, M. R. Rahman, and R. Boutaba, "ViNEYard: Virtual network embedding algorithms with coordinated node and link mapping," *IEEE/ACM Trans. Netw.*, vol. 20, no. 1, pp. 206–219, Feb. 2012.

[15] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. ACM (HotNets)*, Oct. 2012, pp. 31–36.

[16] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proc. ACM (SIGCOMM)*, Oct. 2011, pp. 98–109.

[17] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *Proc. IEEE (INFOCOM)*, Apr. 2011, pp. 1629–1637.

[18] V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, "A survey of network isolation solutions for multi-tenant data centers," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 4, pp. 2787–2821, 4th Quart. 2016.

[19] A. A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *Proc. IEEE (INFOCOM)*, Apr. 2013, pp. 2130–2138.

[20] J. Domke, T. Hoefler, and W. E. Nagel, "Deadlock-free oblivious routing for arbitrary topologies," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2011, pp. 616–627.

[21] R. D. Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, "VeRTIGO: Network virtualization and beyond," in *Proc. Eur. Workshop Softw. Defined Netw.*, Oct. 2012, pp. 24–29.

[22] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive, "A flexible model for resource management in virtual private networks," in *Proc. ACM (SIGCOMM)*, Aug. 1999, pp. 95–108.

[23] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. (NSDI)*, Mar. 2011, p. 24.

[24] Y. Gong, B. He, and J. Zhong, "Network performance aware MPI collective communication operations in the cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 3079–3089, Nov. 2015.

[25] C. Guo *et al.*, "Secondnet: A data center network virtualization architecture with bandwidth guarantees," in *Proc. 6th Int. Conf.*, Nov. 2010, P. 15.

[26] C. E. Hopps, "Analysis of an equal-cost multi-path algorithm," in *Proc. IETF RFC*, 2015, p. 125.

[27] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *Proc. 11th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2011, pp. 104–113.

[28] A. Jajszczyk, "Nonblocking, repackable, and rearrangeable clos networks: Fifty years of the theory evolution," *IEEE Commun. Mag.*, vol. 41, no. 10, pp. 28–33, Oct. 2003.

[29] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *Proc. Comput. Commun. Rev. (SIGCOMM)*, Oct. 2015, pp. 435–448.

[30] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: Practical network performance isolation at the edge," in *Proc. 10th Symp. Netw. Syst. Des. Implement. (NSDI)*, 2013, pp. 297–311.

[31] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang, "Joint VM placement and routing for data center traffic engineering," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 2876–2880.

[32] A. Jokanovic, G. Rodriguez, J. C. Sancho, and J. Labarta, "Impact of inter-application contention in current and future HPC systems," in *Proc. 18th IEEE Symp. Perform. Interconnects*, Aug. 2010, pp. 15–24.

[33] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta, "Quiet neighborhoods: Key to protect job performance predictability," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2015, pp. 449–459.

[34] N. Katta *et al.*, "Clove: Congestion-aware load balancing at the virtual edge," in *Proc. 13th Int. Conf. Emerg. Netw. Exp. Technol.*, Nov. 2017, pp. 323–335.

[35] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner, "Cicada: Introducing predictive guarantees for cloud networks," in *Proc. 6th Workshop Topics Cloud Comput.*, Jun. 2014, pp. 1–6.

[36] S. Radhakrishnan, R. Pan, A. Vahdat, and G. Varghese, "Netshare and stochastic netshare: Predictable bandwidth allocation for data centers," *Comput. Commun. Rev.*, vol. 42, no. 3, pp. 5–11, 2012.

[37] G. Linden, *Make Data Useful*, Stanford CS345 Talk, 2006. Accessed: Apr. 7, 2019. [Online]. Available: https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt

[38] M. Mayer, *In Search of A Better, Faster, Stronger Web*, Velocity Conference, San Jose, CA, USA, 2009. Accessed: Apr. 7, 2019. [Online]. Available: https://assets.en.oreilly.com/1/event/29/Keynote%20Presentation%202.pdf

[39] J. C. Mogul and L. Popa, "What we talk about when we talk about cloud network performance," *Comput. Commun. Rev.*, vol. 42, no. 5, pp. 44–48, 2012.

[40] S. R. Ohring, M. Ibel, S. K. Das, and M. J. Kumar, "On generalized fat trees," in *Proc. 9th Int. Parallel Process. Symp.*, Apr. 1995, pp. 37–44.

[41] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of EC2 cloud computing services for scientific computing," in *Proc. Int. Conf. Cloud Comput.*, Oct. 2009, pp. 115–131.

[42] J. A. Pascual, J. Navaridas, and J. Miguel-Alonso, "Effects of topology-aware allocation policies on scheduling performance," in *Proc. Workshop Scheduling Strategies Parallel Process.*, May 2009, pp. 138–156.

[43] R. Perlman and J. D. Touch, "Transparent interconnection of lots of links (Trill)," in *Proc. IETF RF*, 2009, p. 556.

[44] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," *Comput. Commun. Rev.*, vol. 44, no. 4, pp. 307–318, 2014.

[45] F. Petrini and M. Vanneschi, "k-ary n-trees: High performance networks for massively parallel architectures," in *Proc. 11th Int. Parallel Process. Symp.*, Apr. 1997, pp. 87–93.

[46] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proc. 16th Conf. Comput. Commun. Secur.*, Nov. 2009, pp. 199–212.

[47] A. Samuel, E. Zahavi, and I. Keslassy, "Routing keys," in *Proc. 25th Annu. Symp. High-Perform. Interconnects (HOTI)*, Aug. 2017, pp. 9–16.

[48] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[49] R. Sherwood *et al.*, "Flowvisor: A network virtualization layer," OpenFlow Switch Consortium, Tech. Rep. 1, 2009.

[50] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[51] A. Shieh, S. Kandula, A. G. Greenberg, and C. Kim, "Seawall: Performance isolation for cloud datacenter networks," in *Proc. 2nd Conf. Topics Cloud Comput.*, Jun. 2010, p. 1.

[52] L. Wang, W. Wang, and B. Li, "Utopia: Near-optimal coflow scheduling with isolation guarantee," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2018, pp. 891–899.

[53] P. Wang, G. Trimponias, H. Xu, H. Liu, and Y. Geng, "Luopan: Sampling-based load balancing in data center networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 1, pp. 133–145, Jan. 2019.

[54] X. Wen, K. Chen, Y. Chen, Y. Liu, Y. Xia, and C. Hu, "Virtualknotter: Online virtual machine shuffling for congestion resolving in virtualized datacenter," in *Proc. 32nd Int. Conf. Distrib. Comput. Syst.*, Jun. 2012, pp. 12–21.

[55] E. Zahavi, A. Shpiner, O. Rottenstreich, A. Kolodny, and I. Keslassy, "Links as a service (LaaS): Guaranteed tenant isolation in the shared cloud (extended version)," in *Proc. IEEE Symp. Architectures Netw. Commun. Syst.*, Mar. 2019, pp. 87–98.

[56] F. Zahid, "Network optimization for high performance cloud computing," M.S. thesis, Simula Res. Lab., Univ. Oslo, Oslo, Norway, 2017.

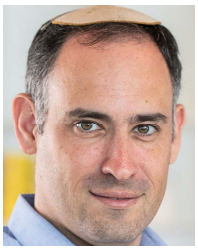
[57] F. Zahid, E. G. Gran, B. Bogdanski, B. D. Johnsen, and T. Skeie, "Partition-aware routing to improve network isolation in infiniband based multi-tenant clusters," in *Proc. 15th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2015, pp. 189–198.

[58] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. H. Katz, "DeTail: Reducing the flow completion time tail in datacenter networks," in *Proc. Conf. Appl., Technol., Architectures, Protocols Comput. Commun.*, Aug. 2012, pp. 139–150.

[59] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *Proc. ACM (SIGCOMM)*, Aug. 2016, pp. 160–173.


**Eitan Zahavi** received the B.Sc. degree and the Ph.D. degree in forwarding in compute clusters from the Department of Electrical Engineering, Technion-Israel Institute of Technology, in 1987 and 2015, respectively. He is currently a Distinguished Architect and the Co-Founder of Mellanox Technologies. He also teaches logic design automation for VLSI systems with the Department of Electrical Engineering Department, Technion. He leads the Mellanox Network Architecture Group, where is currently focusing on cluster level performance. He also acts as the Co-Chair of the IBTA Technical Working Group.

**Alexander Shpiner** received the B.Sc. degree in computer engineering from the Technion-Israel Institue of Technology, the M.B.A. degree from The Faculty of Industrial Engineering and Management, Technion-Israel Institue of Technology, and the Ph.D. degree in electrical engineering from the Technion-Israel Institue of Technology, where he researched topics in the fields of computer networks: congestion control, switch scheduling, network processors, and networks on chip. From 2013 to 2018, he held a position at the System Architecture Group, Mellanox Technologies, where he was focusing on network-wise QoS-related features in switches and NICs. In addition, he has six years of experience in IDF signal corps, where is currently involved in various telecommunication projects for the defense industry. He is a System Architect with Lightbits Labs., Inc., a startup developing cloud infrastructure including networked storage. He held patents. He has also coauthored research papers, two of which received the best paper award.



**Ori Rottenstreich** received the B.Sc. degree *(summa cum laude)* in computer engineering and the Ph.D. degree from Technion in 2008 and 2014, respectively. From 2015 to 2017, he was a Postdoctoral Research Fellow with the Department of Computer Science, Princeton University. He is a Faculty Member with the Department of Computer Science and the Department of Electrical Engineering, Technion, Haifa, Israel. His main research interests are computer networks.

**Avinoam Kolodny** received the Ph.D. degree in microelectronics from the Technion-Israel Institute of Technology in 1980. He joined Intel Corporation, where he was involved in research and development in the areas of device physics, VLSI circuits, electronic design automation, and organizational development. He joined the Faculty of Electrical Engineering, Technion, in 2000. His research interests are focused primarily on interconnects in VLSI systems, at both physical and architectural levels.

**Isaac Keslassy** received the M.S. and Ph.D. degrees in electrical engineering from Stanford University, in 2000 and 2004, respectively. He is currently a Full Professor with the Viterbi Faculty of Electrical Engineering, Technion, Israel. His recent research interests include the design and analysis of datacenter networks and high-performance routers. He was a recipient of the Test of Time Award from the ACM SIGCOMM 2015, the Allon, Mani, Yanai, and Taub Awards, and the ERC Starting Grant.