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

Eitan Zahavi
Mellanox Technologies
eitan@mellanox.com

Alexander Shpiner
Mellanox Technologies
alexshp@mellanox.com

Ori Rottenstreich
Princeton University
orir@cs.princeton.edu

Avinoam Kolodny
Technion
kolodny@ee.technion.ac.il

Isaac Keslassy
VMWare & Technion
isaac@ee.technion.ac.il

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, we derive theoretical conditions for enabling 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. LaaS is implementable with common network gear, tested to scale to large networks and provides full tenant isolation at the worst cost of a 10% reduction in the cloud utilization.

Keywords

Data centers; High performance computing; Tenant isolation

1. INTRODUCTION

Many owners of private data centers would like to move to a shared multi-tenant cloud, which can offer a reduced cost of ownership and better fault-tolerance. For some of these tenants it is vital that their applications will not be affected by other tenants, and will keep exhibiting the same performance[1] [11, 36, 37]. 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 [12, 27]. This unpredictable performance is mainly caused by two factors: *server sharing* and *network sharing* [7, 14, 17, 20, 24, 26, 32, 34, 35, 38, 41, 47, 49, 52–54, 56]. 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 [3]. However, the second factor, *network sharing*, is much more difficult to address. When 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 [31]. This can of course prove even worse when other tenants purposely generate adversarial traffic for DoS or side-channel attacks [48].

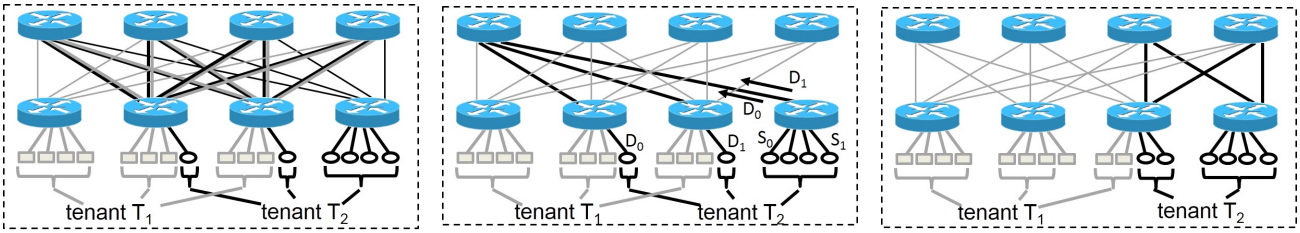
As detailed in Section 2, current solutions either (a) require tenants to provide and adhere to a specific traffic matrix declared in advance, which often proves impractical [14, 56]; (b) follow the hose model by providing enough throughput for any set of admissible traffic matrices [12, 21, 54], but 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, but cannot guarantee constant performance [24, 35, 47, 49, 53]. In addition, while it is known that tailoring the packet forwarding method to the specific tenant application can increase its performance, none of the current cloud solutions allow multiple forwarding algorithms to co-exist on the same network without impacting performance.

In this paper, we introduce a simple and effective approach that eliminates any interference in the cloud network. This approach allows each tenant to use a network forwarding algorithm that is optimized for its own application. Keeping with the notion that good fences make good neighbors, we argue that the most demanding tenants should be provided with exclusive access to a subset of the data center links, such that each tenant receives its own dedicated fat-tree network. We refer to this cloud architecture model as *Links as a Service (LaaS)*. The LaaS model guarantees that these tenants can obtain the exact same bandwidth and delay as

[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.

ANCS '16, March 17-18, 2016, Santa Clara, CA, USA

DOI: http://dx.doi.org/10.1145/2881025.2881028



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

Figure 1: Two tenants hosted on a cloud. (a) Their traffic 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, the network can service any admissible tenant traffic demands.

if they were alone in the shared cloud, independently of the number of additional tenants. We show that allocation of links to tenants is cost-effective and implementable by using common hardware. Note that LaaS can similarly support a relaxed model that splits physical links into time-domain-multiplexed channels. This relaxed model allows multiple tenants per server, but requires accurate packet pacing [29] not provided by common hardware today.

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. To generalize the above examples, we further analyze the fundamental requirements for providing LaaS guarantees to tenants in 2- and 3-level homogeneous fat-trees. Under minor assumptions, our analysis provides the necessary and sufficient conditions to guarantee the same bandwidth and delay performance over the shared fat-tree networks as when being alone in the shared cloud. These conditions are novel and greatly reduce the complexity for the online allocation algorithm presented in Section 3.

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. Thus, the performance improvement exceeds the utilization cost for such applications, uncovering an economic potential (Section 4).

While we focus, for brevity, on full-bisectional-bandwidth fat-trees, we show how LaaS can be extended to support over-provisioned (slimmed) fat-trees. We also describe how LaaS can fit more general cloud cases, e.g. when mixing highly-demanding tenants with regular tenants (Section 5).

Our evaluations show that LaaS is practical and efficient, and completely avoids inter-tenant performance dependence.

2. RELATED WORK

Application variability. Several studies about the variability of cloud services and HPC application performance were presented by [12, 13, 27, 31, 40, 51]. They show significant variability for such applications, which strengthens the motivation for using LaaS.

Network isolation. Specific high-dimensional tori supercomputers like IBM BlueGene, Cray XE6, and the Fujitsu K-computer provide scheduling techniques to isolate tenants [5, 13, 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 same fat-tree by applying hard placement constraints [33]. This work reduces but does not guarantee jobs isolation from each other.

Packet forwarding. Many architectures rely on Equal Cost Multiple Path (ECMP) [25] to spread the allocated tenant traffic and avoid the need to allocate exact bandwidth on each of the used physical links [12,30,46]. 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] aims to provide guaranteed latency, bandwidth and burst size to multiple tenants for a worst-case traffic pattern, assuming that tenants do not optimize their forwarding scheme. Silo achieves its guarantees by applying accurate rate- and burst-size moderation to enforce centrally-calculated values obtained from network calculus. Unfortunately, Silo does not take forwarding into account. For instance, consider a tenant of 200 VMs placed across more than one 2-level sub-tree (which normally can contain thousands of VMs). If 100 VMs need to send traffic to the other VMs through the same uplink because of the forwarding rules, then each would be restricted to use at most 1/100th of the link bandwidth and 1/100th of the switch buffer size, which is unacceptable for current large tenants. LaaS allows the tenants to adapt their forwarding to the traffic pattern without introducing inter-tenant interference, thus allowing them to fully consume the full network bandwidth.

Time separation. Some systems like Cicade [34] accept the need for handling the varying nature of tenant traffic instead of relying only on the average demand. They assume that traffic demands change at a pace that is slow enough to enable them to react. Alternatively, scheduling the MapReduce shuffle stages was proposed by Orchestra [16]. A generalization of this approach that allows a tenant to describe its changing communication needs is suggested by Coflow [15]. On the same line of thought, scheduling at a finer grain was proposed by Hedera [7]. 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. Even though fairness does improve, the tenant performance variability grows.

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 [38].

Virtual Network Embedding maps tenants' requested topologies and traffic matrix over arbitrary clusters [14, 56]. However, tenants must know and declare their exact traffic demands which is mostly impractical. Moreover, valid embedding is calculated by variants of linear programming, which are known not to scale as the size of the data centers and number of tenants grow. In addition, as most of these solutions rely on the tenant traffic matrix, they consider only the average demands, falling short of representing the dynamic nature of the application traffic. For example, they prove problematic when an application alternates between several traffic permutations, each utilizing the full link bandwidth.

Other proposals, such as Topology Switching and Oktopus [12,54], 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 context of WAN [8]. In addition, [10] attempts to provide a feedback-based fair-share bandwidth using edge-based rate-limiting. However, to guarantee tenant latency predictability and isolation, such solutions would need strict time-pacing of packets, small limits on allowed VM bandwidth and burst-size allocation, as shown in [29]. As mentioned above, these are impractical in current networks.

Another approach for isolation may rely on distributed rate limiting like [47], NetShare [35], ScondNet [24], Seawall [53], Gatekeeper [49] and Oktopus [12]. But distributed rate limiting 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.

Fairness. FairCloud provides a generalization of the required fairness properties of the shared cloud network [45]. LaaS tenant isolation satisfies these requirements, and avoids the allocation complexity of the general case.

Application-based routing. The above schemes for network resource allocation ignore the fact that each tenant application may perform best with a different routing scheme. Routing algorithm types span a wide range. Some are completely static and optimized for MPI applications [22, 57]. Others rely on traffic-spreading techniques like ECMP [25], rely on traffic spray as in RPS or DeTail [18, 58], use adaptive routing as proposed by DARD [55], or even rely on per-packet synchronized schemes like FastPass [43]. 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 the different routing engines must continuously coordinate the actual bandwidth each one of them utilize from each link. It is clear that the involved complexity of such scheme renders it slow and impractical.

3. LAAS ALGORITHM

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.

3.1 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].

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 $N = \sum_{i=1}^{r} N_i$.

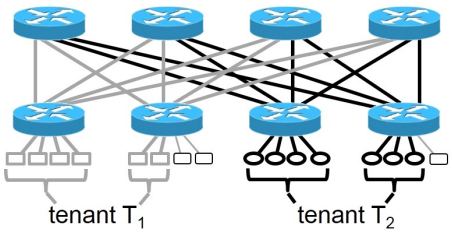


Figure 2: Two tenants of sizes 6 and 7 hosts placed by the *Simple* heuristic, where each tenant fills a number of complete sub-trees.

(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.

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 sub-tree, then traffic from other tenants will not be routed through that sub-tree. Note that for 2-level fat-trees a sub-tree 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.

Fig. 2 illustrates the *Simple* algorithm, showing how tenant T_1 obtains a placement for $N = 6$ hosts. First, $s = \lceil 6/4 \rceil = 2$. Assuming T_1 arrives first, the two left leaves are available when it arrives, and they are used to host T_1. Also, all the up-links of these 2 leaf switches are allocated to T_1. When it arrives, tenant T_2 is similarly allocated the two right leaves and their up-links.

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].

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:*

$$\forall i < j \in [1,t] : N_i = \min(N_i, N_j) \leq |S_i \cap S_j| \tag{2}$$

PROOF. Consider a traffic permutation among the tenant hosts. There are up to N_i full-link-capacity host-to-host flows going from L_i to L_j (or back). Since each flow has to use a different link and each link goes to a different spine switch, we will need at least N_i common spine switches in $|S_i \cap S_j|$. □

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.* $\forall i,j,k \in [1,t] : \min(N_i + N_j, N_k) \leq |S_i \cup S_j|$.

PROOF. Let $c = \min(N_i + N_j, N_k)$. There are at most c flows going from L_k to either L_i or L_j (or back). Since each flow has to use a different link and each link goes to a different spine switch, we will need at least c spines in the union $S_i \cup S_j$ of the spines connected to the two leaves. □

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

PROOF. Assume the contrary: $N_i + N_j < N_k$. There are only two cases: $N_i \leq N_j < N_k$ or $N_j \leq N_i < N_k$. W.l.o.g., we assume the first. If so, $\min(N_i + N_j, N_k) = N_i + N_j$. By Lemma 1, to enable connectivity between N_i and N_j, they must have at least N_i spines in common: $|S_i \cap S_j| \geq N_i$. Substituting the above into Lemma 2 we obtain: $\forall i,j,k \in [1,t] : \min(N_i + N_j, N_k) = N_i + N_j \leq |S_i \cup S_j| = |S_i| + |S_j| - |S_i \cap S_j|$. But since $N_i = |S_i|$ and $N_j = |S_j|$ in LaaS by Equation (1), we get $0 \leq -|S_i \cap S_j|$. But $S_i \cap S_j$ is non-empty because otherwise traffic from hosts in leaf i to hosts in j wouldn't be able to pass. So we get a contradiction, thus $N_i + N_j \geq N_k$. □

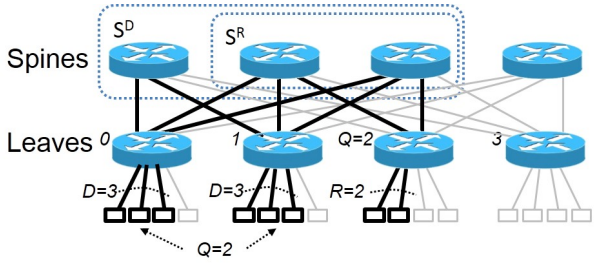


Figure 3: A tenant of $N = 8 = Q \cdot D + R$ hosts. To implement LaaS, there must be Q leaves of D hosts and optionally one leaf of $R < D$ hosts.

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 $N_1 \leq N_2 \leq \cdots \leq N_t$.

THEOREM 1. *A necessary condition for LaaS is*

$$N_1 \leq N_2 = N_3 = \cdots = N_t, \tag{3}$$

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

PROOF. We show that $N_2 = N_t$. By Lemma 1, L_1 and L_2 must have at least $N_1 = |S_1|$ spines in common, i.e. $S_1 \subseteq (S_1 \cap S_2)$. Therefore, S_1 is a subset of S_2, so $|S_1 \cup S_2| = |S_2| = N_2$. By Lemma 3, when $i = 1$, $j = 2$ and $k = t$, $N_1 + N_2 \geq N_t$ thus $\min(N_1 + N_2, N_t) = N_t$. So, when N_t flows are sent from L_t to L_1 and L_2, we must have at least N_t common spines: $|S_1 \cup S_2| = N_2 \geq N_t$. But since $N_2 \leq N_t$, it follows that $N_2 = N_t$. □

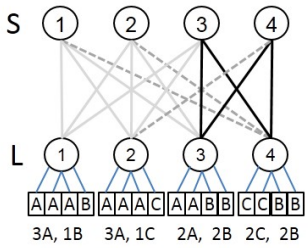
Given Theorem 1, the tenant placement should follow the form: $N = Q \cdot D + R$, where Q is the number of repeated leaves with D hosts each, and we optionally add one unique leaf with a smaller number of hosts R. This notation follows the Divisor, Quotient and Remainder of N. This result is useful 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.

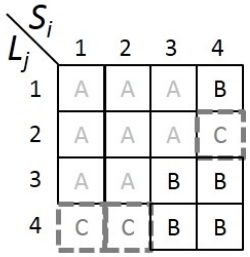
Sufficient link allocation. We can now prove sufficient conditions on the link allocation to satisfy LaaS.

THEOREM 2. *A 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).*

PROOF. For the case $R = 0$, the link allocation above means there is a group of D spine switches that connect to all leaf switches. Thus the tenant sub-topology reduces to



(a) Placement



(b) Link Allocation

Figure 4: Illustration that a simple host placement is not sufficient, and a joint host placement and link allocation is necessary for LaaS. (a) All tenants satisfy the host placement necessary conditions, e.g. the placement of C is $3 = Q \cdot D + R = 2 \cdot 1 + 1$. 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.

an *Full Bipartite Graph* (FBG) with $m' = D$ spine switches and $n' = D$ hosts per leaf. Since $m' = n'$ such topology is rearrangeable non-blocking folded-Clos which is known to support any admissible traffic matrix as mentioned above.

For the case of one additional leaf L_{j_R} of R 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 $m' = n' = D$ and $r' = Q + 1$. Again, $m' = n'$ so it is guaranteed by the rearrangeable non-blocking theorem that every full permutation of $n' \cdot r'$ flows is route-able. Routing is symmetric with respect to the spine switches. Moreover, to avoid congestion, each spine needs to carry exactly 1 flow from each leaf and 1 flow to each leaf. So any full permutation of our original topology where L_{j_R} has only R flows will be $D - R$ flows short. We extend these flows with $D - R$ flows going from L_{j_R} to L_{j_R}. Since these flows share the same leaf switch they must be routed through $D - R$ different spines. After completing the full permutation routing, and since all spines connect to all leaves, we swap between each spine that carries one of the added $D - R$ flows with a spine that is not included in S^R. As the links allocated to the extra flows are not needed, any permutation is fully routed by the original topology. □

A necessary host allocation is not sufficient. The above theorems provide us with guidelines for implementing LaaS. We now show that due to previous tenant allocations, a host placement as in Theorem 1 is not always sufficient to provide a needed link allocation as in Theorem 2. This is why Theorem 2 proves essential. If the link allocation cannot be found for a specific placement our algorithm will need to search for another host allocation.

LEMMA 4. *A host placement that meets Theorem 1 does not guarantee the existence of a link allocation that meets Theorem 2, and therefore does not guarantee LaaS.*

PROOF. We prove Lemma 4 by the example provided in Fig. 4. Three tenants are shown placed according to the provided heuristic of the previous section: A has $8 = 2 \cdot 3 + 2$ hosts, B has $5 = 2 \cdot 2 + 1$, and C has $3 = 1 \cdot 2 + 1$. We track

allocated up-links of the leaf switches in a matrix where rows represent the leaf switches and columns represent the spines each port connects to. As can be observed, there is no possible link allocation for tenant C, since the leaves it is placed on do not have free links connected to any common spine. There is no link allocation possible for C even though it was placed according to the conditions of Theorem 1. The online link allocation algorithm for C (after A and B are placed) cannot allocate the links. In fact, even an offline version of link allocation - reassigning the links of A and B - cannot solve the problem once the placement of A and B does not change. □

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.

3.2 Isolation for 3-level Fat Trees

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, 9]. We use the notation of Extended Generalized Fat Trees (XGFT) [39], which defines fat-trees of h levels and the number of sub-trees at each level: $m_1, m_2, \ldots, m_h$. and the number of parent switches at each level: $w_1, w_2, \ldots, w_h$.

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

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 sub-tree 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:

$$l_{min} = \min\{l | (R_{l-1} < N) \wedge (R_l \geq N)\} \quad (4)$$

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

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 [44].

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. We show that for a LaaS link allocation to be feasible, the condition of Theorem 1 needs to hold not only for each 2-level sub-tree but also for each lvl_2 - lvl_3 Full Bipartite Graph (FBG_2) at the top of the tree. One of these FBGs is highlighted in Fig. 5.

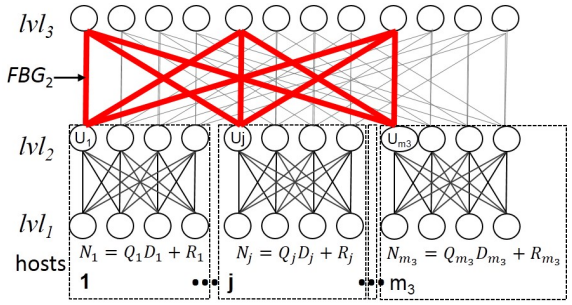


Figure 5: A 3-level fat-tree showing the host allocation on each 2-level sub-tree matching Theorem 1. One of the lvl_2 - lvl_3 Full Bipartite Graphs (FBG_2) is highlighted. We denote as U_j the maximal number of flows injected into this FBG_2 from the j^{th} 2-level sub-tree.

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 1. So the number of tenant hosts within the 2-level sub-tree 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 $Q_j = 1$.

Fig. 5 depicts a Theorem 1-compliant host allocation within each of the 2-level sub-trees. It follows the form: $N_j = Q_j \cdot D_j + R_j | j \in \{1...m_3\}$. Note that the link assignment within the 2-level sub-trees must also adhere to Theorem 2 such that $S_j^R \subset S_j^D$. Consequently, the maximum number U_j of flows leaving the 2-level sub-tree 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 $s \in S_j^R$.

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 $U_1 \leq U_2 = U_3 = \cdots = U_{m_3}$. 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 sub-trees) could select a different set of S_j^D and S_j^R. This means that a solution could allow each 2-level sub-tree to select a different set of FBG_2 to carry its flows, as long as the above rule is maintained for each FBG_2.

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 in practical time even on high-end processors. If we were to provide an optimal allocation we would conclude here that our problem is too hard. But our task is not to find the *optimal* solution, or even *any* solution at a specific iteration. 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.

Approximated algorithm. We provide a simpler algorithm that compromises cluster utilization in favor of reduction of the solution search space. Our approximation

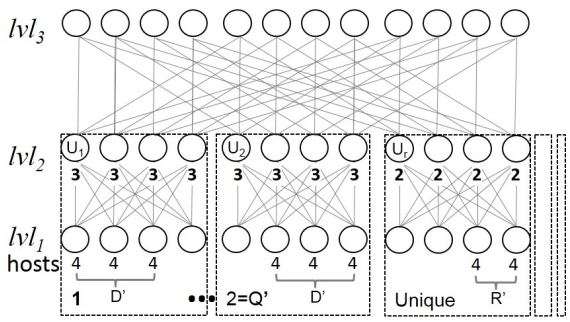


Figure 6: An example of host placement with $N = 32$ hosts on a 3-level fat-tree using the *Approximated* method. Using a notation similar to the 2-level fat-tree, this allocation is of the form: $Q' = 2$, $D' = 3$ and $R' = 2$.

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 sub-tree. 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 sub-trees, 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 sub-tree, 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 sub-trees, each with $D' = Q$ leaves. Optionally there may be one additional 2-level sub-tree 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 $m_1 = 4$ hosts per leaf, is provided in Fig. 6. On the left $Q' = 2$ sub-trees, the tenant uses $D' = 3$ leaves and thus $U_1 = U_2 = 3$ for all FBG_2. In addition a single unique sub-tree r with $R' = 2$ leaves is also allocated and thus $U_r = 2$ 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 1 and thus may be feasible.

LaaS algorithm.

We now want to implement our final *LaaS* algorithm for concurrent host placement and link allocation in fat-trees.

Algorithm 1 FLAP$(D, Q, R, l, l_e, r, \{ports\}, \{rl\})$

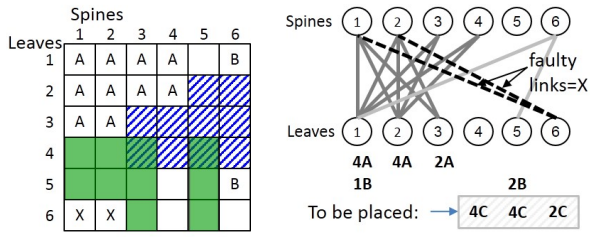
```
1: // find next Q size leaf
2: for i = l to l_e do
3:   if |M[i]| >= Q then
4:     {nPorts} = {ports} ∩ M[i]
5:     if |nPorts| ≥ Q then
6:       {newRL} = {rl} ∪ i
7:       if r = D then
8:         // found all repeated leaves
9:         if findUniqueLeaf(R, l_s, l_e; {nPorts} {rl}) then
10:          {D_PORTS} = {nPorts}
11:          {D_L} = {newRL}
12:          return true
13:        end if
14:      else
15:        j = i + 1; s = r + 1
16:        if FLAP(D,Q,R,j,l_e,s,{nPorts},{newRL}) then
17:          return true
18:        end if
19:      end if
20:    end if
21:  end if
22: end for
23: return false
```

Algorithm 2 LAAS(N)

```
1: // Try 1 level allocation
2: if N ≤ m_1 then
3:   for l = 0 TO m_2 · m_3 − 1 do
4:     if FLAP(N, 1, 0, l, l, 0, {}, {}) then
5:       return true
6:     end if
7:   end for
8: end if
9: // Try 2 level allocation
10: if N ≤ m_1 · m_2 then
11:   for D = max(N, m_1) to 1 do
12:     Q = ⌊N/D⌋
13:     R = N − Q · D
14:     for l = 0 TO m_3 − 1 do
15:       if FLAP(D, Q, R, l · m_2, (l+1) · m_2 − 1, 0, {}, {}) then
16:         return true
17:       end if
18:     end for
19:   end for
20: end if
21: // Try 3 level allocation
22: U = ⌈N/m_1⌉
23: for D = max(U, m_2) to 1 do
24:   Q = ⌊U/D⌋
25:   R = U − Q · D
26:   if Q ≤ m_3 then
27:     if FLAP2(D, Q, R, 0, m_3 − 1, 0, {}, {}) then
28:       return true
29:     end if
30:   end if
31: end for
32: return false
```

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 $N = Q \cdot D + R$. As described in the sub-section 'LaaS placement analysis', 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.

The search for Q leaves with enough common spines is performed recursively. In the worst case, it may require examining all $\binom{m_2}{Q}$ combinations. Our *LaaS* algorithm returns



(a) Link Allocation Table (b) Corresponding Topology

Figure 7: Example of allocation with 2 potential placements. (a) Table of leaf up-links holding the link assignments of tenants A and B, as well as 2 faulty links X. (b) Corresponding topology. The new tenant C of 10 hosts, arranged as $Q \cdot D + R = 2 \cdot 4 + 2$, can be assigned one of two allocations. In (a), the first link allocation is shown in solid, and the second with slanted lines.

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. The algorithm to perform the above example is provided in Algorithm 1. 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 2.

Extension for over-subscribed fat trees. In order to reduce the network equipment cost, some cloud vendors use over-subscribed fat-trees, also known as *slimmed fat-trees* [50]. In an over-subscribed fat-tree, the number of uplinks is smaller than the number of downlinks in the switches, contrarily to the full bisectional bandwidth fat-tree, where they are equal. (We assume equal-bandwidth links). In such trees, we denote O_i the ratio between the two total number of links: those connecting switches at level i to the previous level $i-1$, and those connecting to the next level $i+1$. By this definition for XGFT:

$$O_i = \frac{m_i}{w_{i+1}} \tag{5}$$

We describe here how to provide LaaS for over-subscribed fat-trees, without requiring hardware-assisted accurate TDMA link sharing. For simplicity we do not support tenant selection of their requested bandwidth. Since we allow no link-sharing between tenants, and we have no preference between tenants, a tenant placed across a level i of the tree has at least O_i permutation flows shared on each link. So for crossing level i we only require S common switches at level $i+1$:

$$S = \left|S^D\right| = \left\lceil \frac{D}{\lceil O_i \rceil} \right\rceil \tag{6}$$

Clearly, a selection of D such that it is not divisible by $\lceil O_i \rceil$ reduces the cluster utilization, so the order by which we search for sub-trees should reflect that priority. The changes to Algorithm 1 are a new function argument S which defines the number of spines required, and its usage in line 7: ***if*** $r = S$ ***then***. The changes to Algorithm 2 involve adding an S of Equation (6) to the calls of $FLAP$ and also adding an external loop around the ***for*** statements in lines $11-19$ and $23-31$ to try D values divisible by $\lceil O_i \rceil$ first.

4. EVALUATION

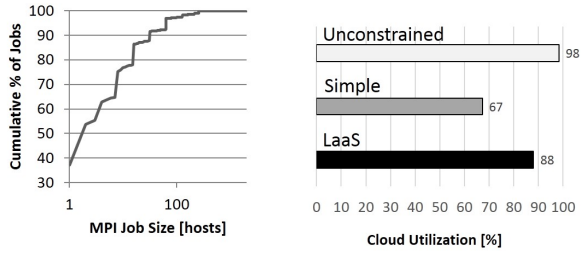
Our evaluation is reported in three sub-sections. The first deals with the resulting *cloud utilization* when applying LaaS conditions. It shows that our *LaaS* algorithm reaches a reasonable cloud utilization, within about 10% of bare-metal allocation. The second part describes the system implementation 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 by completely isolating the tenants from each other.

4.1 Evaluation of Cloud Utilization

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 revenue of the cloud provider.

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.

Simulation 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 11,662 hosts. The evaluation uses a randomized



(a) (b)

Figure 8: (a) Measured job-size Cumulative Distribution Function (CDF) for the Julich JUROPA scientific-computing cloud. (b) Resulting cloud utilization. *LaaS* achieves 88%.

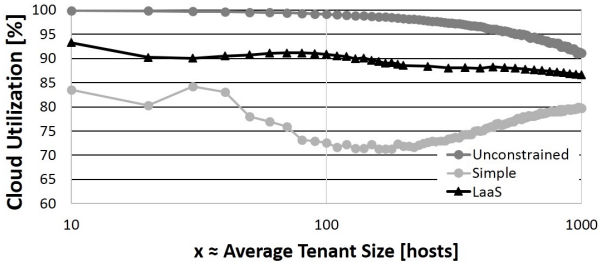


Figure 9: Cloud utilization for a truncated exponential distribution of tenant host sizes in a cloud of 11,662 hosts.

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 incoming tenants. First, we randomly generate sizes 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 3.

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 probability 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 sub-trees. 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 sub-tree (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.

4.2 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.

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.

Network controller. We further implement a method to provide the link allocation to the InfiniBand SDN controller [2], which allows it to enforce the isolation by chang-

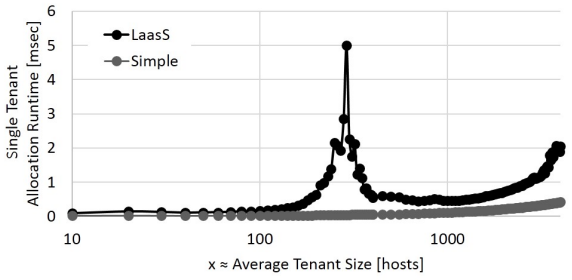


Figure 10: Average run-time of single tenant allocation versus average tenant size.

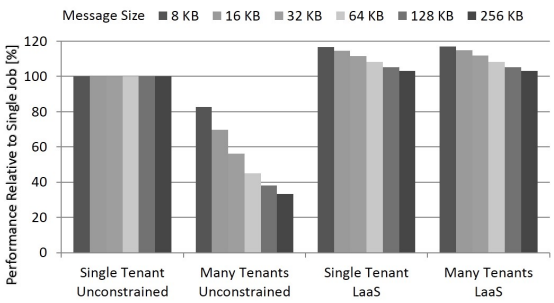


Figure 11: Simulated relative performance for tenants running Stencil scientific-computing applications on a cloud of 1,728 hosts, either alone or as 32 concurrent tenants. While tenant performance degrades when placed unconstrained (without link isolation), the performance of single and multiple tenants with LaaS appears identical, fulfilling the promise of LaaS.

ing 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.

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. 10 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 sub-tree and continues with multiple sub-tree placement.

4.3 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 by [19, 23, 57].

Fig. 11 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.

5. DISCUSSION

Recursive LaaS. When talking to industry vendors, they pointed out simple extensions that would easily generalize the use of LaaS. First, LaaS could be applied recursively, by having each tenant application or each sub-tenant reserve its own chunk of the cloud within the tenant's chunk of the cloud. Second, LaaS could also be applied in private clouds, with cloud chunks being reserved by applications instead of tenants. Third, shared-cloud vendors could easily restrict LaaS to a subset of their cloud, while keeping the remainder of their cloud as it is today. This can be done by reserving large portions of the topology to a *virtual* tenant that is shared between many real tenants. Pre-allocation and modification of that sub-topology is already supported by our code. As a result, LaaS offers a smooth and gradual transition to better service guarantees, enabling cloud vendors to start only with the tenant owners who are most ready to pay for it.

Off-the-shelf LaaS. LaaS is implementable today with no extra hardware cost in existing switches and no host changes. The algorithm requires only a moderate software change in the allocation scheme, which we provide as open source. It also relies on an isolated-routing feature of the SDN controller, which is already available in InfiniBand and could be implemented in Ethernet SDN controllers like OpenDaylight.

Proportional network power. LaaS eases the use of an elastic network link power that would be made proportional to cloud utilization [4]. This is because it explicitly mentions which links and switches are to be used, and therefore can turn off other links and switches. In other approaches the control has to happen as a result of traffic load change and thus is not realistic for common switch hardware for which the turn-ON time is much larger than a microsecond.

Heterogeneous LaaS. Host allocation in heterogeneous clouds involves allowing tenants to express their required host features in terms of CPU, memory, disk and available accelerators. On such systems, the host allocation algorithm should allow the provider to trade off the acceptance of a new tenant versus the cost of the available hosts, which may be higher as their capabilities may exceed the user needs. Our

LaaS algorithm could support these requirements. *Although this requirement complicates the allocation algorithm, it is feasible to support it in LaaS.* First, it should use the host costs to order the search. Second, it should try all the possible divisors and select the one with best accumulated cost. A trade-off between the resulting fragmentation and the cost difference could extend it.

LaaS with VMs. LaaS could easily support multiple tenants running as virtual machines (VMs) on the same host, assuming accurate packet pacing and burst control is provided by hosts and switches. LaaS could then treat each link as a set of isolated links and assign them to different tenants. This includes the links leaving the host.

Non-FIFO tenant scheduling. We conservatively evaluated our *LaaS* allocation algorithm assuming FIFO scheduling of incoming tenants. To improve the cloud utilization, we could equally rely on a non-FIFO policy, e.g. by using back-filling, reservations, or a jointly-optimal allocation of multiple tenants [42].

Fault Tolerance. When a link is down before being allocated it is easy to avoid allocating it to new tenants. However, if a link was already allocated to a tenant, it is not always possible to provide an alternative link without breaking the current operation of the tenant. Similarly to losing a link on the private cloud, the tenant will see some degradation until the link is fixed or the forwarding plane is adapted.

6. CONCLUSIONS

In this paper, we demonstrated that the interference with other tenants causes a performance degradation in cloud applications that may exceed 65%. 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 change of hardware, and showed how it can rely on open-source software code that we contributed. Finally, we also used previously-unpublished tenant-size statistics of a large scientific-computing cloud, obtained over a long period of time, to construct a random workload that illustrates how isolation is possible at the cost of some 10% cloud utilization loss.

7. ACKNOWLEDGMENTS

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

This work was partly supported by the Technion Funds for Security Research, the Intel ICRI-CI Center, the Hasso Plattner Institute Research School, the Gordon Fund for Systems Engineering, the Israel Ministry of Science and Technology, and the Shillman, Erteschik and Greenberg Research Funds.

8. REFERENCES

[1] LaaS source code, experiments and simulation conditions. https://www.dropbox.com/sh/uzla7rcmdiqrxig/AADpw5ALG-q8VFzMSVcmYvmJa/.

[2] OpenSM - InfiniBand Open SDN Controller.

[3] OpenStack ironic. https://wiki.openstack.org/wiki/Ironic.

[4] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy proportional datacenter networks. *SIGARCH Comput. Archit. News*, 2010.

[5] Y. Ajima, S. Sumimoto, and T. Shimizu. Tofu: A 6d mesh/torus interconnect for exascale computers. *Computer*, 2009.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 2008.

[7] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. *NSDI*, 2010.

[8] A. Altin, H. Yaman, and M. C. Pinar. The robust network loading problem under hose demand uncertainty. *INFORMS Journal on Computing*, 2010.

[9] A. Andreyev. Introducing data center fabric, the next-generation Facebook the next generation datacenter network, 2014.

[10] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end performance isolation through virtual datacenters. *USENIX OSDI*, Berkeley, CA, USA, 2014.

[11] D. Artz. The secret weapons of the AOL optimization team. *Velocity Conference*, 2009.

[12] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. *SIGCOMM*, 2011.

[13] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. There goes the neighborhood: Performance degradation due to nearby jobs. *ACM SC*, 2013.

[14] M. Chowdhury, M. R. Rahman, and R. Boutaba. ViNEYard: Virtual network embedding algorithms. *IEEE/ACM ToN*, 2012.

[15] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. *ACM HotNets*, 2012.

[16] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM*, 2011.

[17] A. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. *IEEE Infocom*, 2011.

[18] A. Dixit, P. Prakash, Y. Hu, and R. Kompella. On the impact of packet spraying in data center networks. *IEEE Infocom*, 2013.

[19] J. Domke, T. Hoefler, and W. E. Nagel. Deadlock-free oblivious routing for arbitrary topologies. *IEEE IPDPS*, 2011.

[20] R. Doriguzzi Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori. VeRTIGO: Network virtualization and beyond. *EWSDN*, 2012.

[21] 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. *ACM SIGCOMM '99*, New York, NY, USA, 1999.

[22] Y. Gong, B. He, and J. Zhong. Network performance aware MPI collective communication operations in the cloud. *IEEE TPDS*, 2013.

[23] E. Gran, S.-A. Reinemo, O. Lysne, T. Skeie,

E. Zahavi, and G. Shainer. Exploring the scope of the InfiniBand congestion control Mechanism. *IEEE IPDPS*, 2012.

[24] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. *ACM CoNext*, 2010.

[25] C. E. Hopps. Analysis of an equal-cost multi-path algorithm. http://tools.ietf.org/html/rfc2992, 2015.

[26] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE TPDS*, 2011.

[27] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. *CCGrid*, 2011.

[28] A. Jajszczyk. Nonblocking, repackable, and rearrangeable clos networks. *IEEE Communications Magazine*, 2003.

[29] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: predictable message latency in the cloud. *ACM SIGCOMM*, 2015.

[30] V. Jeyakumar, M. Alizadeh, D. MaziÃÍres, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: practical network performance isolation at the edge. *NSDI USENIX*, 2013.

[31] A. Jokanovic, G. Rodriguez, J. Sancho, and J. Labarta. Impact of inter-application contention in current and future HPC systems. *IEEE HotI*, 2010.

[32] A. Jokanovic, J. Sancho, J. Labarta, G. Rodriguez, and C. Minkenberg. Effective quality-of-service policy for capacity high-performance computing systems. *IEEE HPCC*, 2012.

[33] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta. Quiet neighborhoods: key to protect job performance predictability. *IEEE IPDPS 2015*, Hyderabad, India, 2015.

[34] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. *USENIX HotCloud*, 2014.

[35] V. T. Lam, S. Radhakrishnan, R. Pan, A. Vahdat, and G. Varghese. Netshare predictable bandwidth allocation for data centers. *SIGCOMM Comput. Commun. Rev.*, 2012.

[36] G. Linden. Make data useful, 2006.

[37] M. Mayer. In search of a better, faster, stronger web. *Velocity Conference*, 2009.

[38] J. C. Mogul and L. Popa. What we talk about when we talk about cloud network performance. *SIGCOMM Comput. Commun. Rev.*, 2012.

[39] S. Ohring, M. Ibel, S. Das, and M. Kumar. On generalized fat trees. *IPPS*, 1995.

[40] J. Orduna, F. Silla, and J. Duato. A new task mapping technique for communication-aware scheduling strategies. *ICPP Workshops*, 2001.

[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. LNICST. Springer, Jan. 2010.

[42] J. A. Pascual, J. Navaridas, and J. Miguel-Alonso. Effects of topology-aware allocation policies on scheduling performance. *Job Scheduling Strategies for Parallel Processing*. 2009.

[43] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. *ACM SIGCOMM*, 2014.

[44] F. Petrini and M. Vanneschi. k-ary n-trees: high performance networks for massively parallel architectures. *IPPS*, 1997.

[45] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. *ACM SIGCOMM*, 2012.

[46] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. ElasticSwitch: Practical Work-conserving Bandwidth Guarantees for Cloud Computing. *ACM SIGCOMM*, New York, NY, USA, 2013.

[47] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. *ACM SIGCOMM*, 2007.

[48] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud. *ACM CCS*, 2009.

[49] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *WIOV*, 2011.

[50] G. Rodriguez, C. Minkenberg, R. Beivide, R. Luijten, J. Labarta, and M. Valero. Oblivious routing schemes in extended generalized Fat Tree networks. *IEEE CLUSTER*, 2009.

[51] J. Schad, J. Dittrich, and J.-A. QuianÃl'-Ruiz. Runtime measurements in the cloud. *VLDB Endowment*, 2010.

[52] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[53] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: performance isolation for cloud datacenter networks. *USENIX HotCloud*, 2010.

[54] K. C. Webb, A. C. Snoeren, and K. Yocum. Topology switching for data center networks. *Hot-ICE Workshop*, 2011.

[55] X. Wu and X. Yang. DARD: Distributed adaptive routing for datacenter networks. *IEEE ICDCS*, 2012.

[56] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking virtual network embedding. *SIGCOMM Comput. Commun. Rev.*, 2008.

[57] E. Zahavi. Fat-tree routing and node ordering providing contention free traffic for MPI global collectives. *JPDC*, 2012.

[58] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: reducing the flow completion time tail in datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 2012.