IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

Links as a Service (LaaS): Feeling Alone in the Shared Cloud

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

CCIT Report #888
September 2015

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT Report #888 September 2015

Links as a Service (LaaS): Feeling Alone in the Shared Cloud

Eitan Zahavi*§, Alex Shpiner§, Ori Rottenstreich¶, Avinoam Kolodny* and Isaac Keslassy*
*Technion §Mellanox ¶Princeton University

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 physical 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. Under simple assumptions, we derive theoretical conditions for enabling LaaS without capacity overprovisioning in fat-trees. New tenants are only admitted in the network when they can be allocated hosts and links that maintain these conditions. Using experiments on real clusters as well as simulations with real-life tenant sizes, we show that LaaS completely avoids the performance degradation caused by traffic from concurrent tenants on shared links. Compared to mere host isolation, LaaS can improve the application performance by up to 200%, at the cost of a 10% reduction in the cloud utilization.

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. But it is vital for these tenants that their applications will will not be affected by other tenants, and will keep exhibiting the same performance [1–3]. (By *performance*, we refer to the inverse of the total application run-time, including both the computation and communication times.)

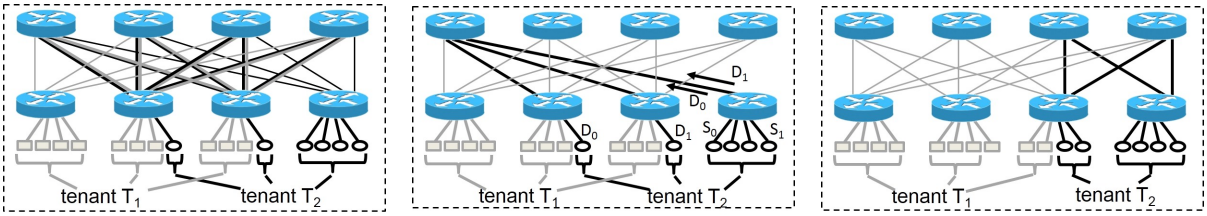
Unfortunately, distributed applications often suffer from unpredictable performance when run on a shared cloud [4, 5]. This unpredictable performance is mainly caused by two factors: *server sharing* and *network sharing* [6–22]. The first factor, *server sharing*, is easily addressed by using bare-metal provisioning of servers, such that each server is allocated to a single tenant [23]. 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 [24]. This can of course prove even worse when other tenants purposely generate adversarial traffic for DoS or side-channel attacks [25].

As detailed in Section 2, current solutions either (a) require tenants to provide the traffic matrix in advance, which often proves impractical [11, 21]; (b) provide enough throughput for any set of admissible traffic matrices using a hose model, but then fail to provide a predictable latency that does not depend on other tenants [4, 16]; or (c) attempt to track the current traffic matrix, but then cannot guarantee the same performance [14, 15, 17, 19, 22].

In Section 3, we establish experimentation and simulation environments to better understand the impact of network contention as a function of the number of tenants and of the cluster size. We show that concurrent tenants with similar traffic can degrade MapReduce performance by 25%, and reduce the performance of scientific computing jobs by up to 65%.

In this paper, we introduce a simple and effective approach that eliminates any interference in the cloud network. 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 such a cloud architecture model as *Links as a Service (LaaS)*. Under the LaaS model, we guarantee that tenants can obtain the exact same bandwidth and delay as if they were alone in the shared cloud, independently of the number of additional tenants.

While the LaaS abstraction sounds 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



(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, the network can service any admissible tenant traffic demands.

the tenants share common links. Likewise, the allocation of hosts and links in Fig. 1(b) also does not satisfy LaaS, even though there are no common links. This is because 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 abstraction. All tenants obtain dedicated hosts and links, and can service any admissible traffic demands between their nodes, independently of the traffic of other tenants.

In this paper, we focus on the practical online incremental LaaS allocation problem: We consider a single incoming tenant at a time, and assign hosts and links out of the unassigned ones, without any migration of previously-allocated tenants. We further analyze the fundamental requirements for providing LaaS guarantees to incoming 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 dedicated fat-tree networks as when being alone in the shared cloud. These conditions are novel and key for providing an online scalable packing-type allocation algorithm. We further present and implement a practical algorithm for providing LaaS guarantees (Section 5).

Our evaluations show that LaaS is practical and efficient, and completely avoids inter-tenant performance dependence. We contribute our implementation of a standalone LaaS scheduler that automates tenant placement on top of OpenStack, as well as configures an InfiniBand SDN controller to provide isolated routing without interference. Our open-source code is made available online [27]. 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 machine provisioning time. 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, and most importantly, we show performance improvements of 50%-200% for highly-correlated tenant traffic. Thus, the performance improvement typically exceeds the utilization cost for such applications, uncovering an economic potential (Section 6).

Finally, while we focus on full-bisectional-bandwidth fat-trees in the provided implementation, our approach can be easily extended to support oversubscribed trees. We also describe how LaaS can fit more general cloud cases, e.g. when mixing highly-demanding jobs with regular jobs (Section 7).

2. RELATED WORK

Application variability. Several studies about the variability of cloud services and HPC application performance were presented by [4, 5, 24, 28, 29]. They show significant variability for such applications, which strengthens the reasons for using LaaS. We extend these claims using experiments and simulations (Section 3). We show that a larger number of tenants causes higher performance loss, and also show that lossy Ethernet-based networks suffer from similar issues.

Network isolation. Specific high-dimensional tori super-computers like IBM BlueGene, Cray XE6, and the Fujitsu K-computer provide scheduling techniques to isolate tenants [30–32]. 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.

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 [6].

Virtual Network Embedding maps tenants' requested topologies and traffic matrix over arbitrary clusters [11, 21]. However, tenants must know and declare their exact traffic demands. Also, 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 [4, 16], 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 [33]. Unfortunately, these proposals still maintain some level of link sharing, and therefore do not address our demands. For instance, when a tenant sends temporary traffic bursts, these burst may conflict at the network switches with temporary traffic bursts from another tenant. As a result, the network latency may be significantly higher when there are many tenants in the shared cloud.

In addition, many of the above systems rely on ECMP-based load-balanced forwarding to spread the allocated tenant bandwidth and avoid the need to allocate exact bandwidth on each of the used physical links [4, 34]. However, while ECMP load-balancing is able to balance the average bandwidth, it suffers from a heavy tail of the load distribution. Again, other tenants will affect the application performance, even in the presence of a slow network feedback.

Proposals that allocate parts of a link bandwidth to different tenants enforce that allocation using *rate limiters* that aggregate tenant traffic bandwidth. But such aggregated bandwidth rate limiters are scarce hardware resource on the switches. Also, they are known to fail to relieve the contention caused by incast. Moreover, they are bound to fail when traffic patterns change rapidly. For example, if tenant T_1 synchronously iterates over all its hosts as destinations, the temporal incast will likely fill up some network buffers. Such traffic pattern of T_1 is impossible to rate-limit when the average bandwidth of each flow is lower than the link bandwidth, or worst, if the aggregate application bandwidth is lower than the total tenant requested bandwidth. If tenant T_2 shares some of its links with tenant T_1, its performance will be impacted, contradicting the goal of this paper. To overcome the limited availability of switch rate limiters, some proposals use Distributed Rate Limiting like [22], NetShare [14], ScondNet [17], Seawall [19], Gatekeeper [15] and Oktopus [4]. The distributed rate limiting at the network edge requires tenant-wide coordination to avoid bottlenecks due to load-imbalance, which results in response times in the order of milliseconds [34], while the co-flow time characteristics for high-demanding applications are 2 to 3 orders of magnitude shorter.

Time separation. Some systems like Cicade [10] do not rely on average rate limiters, but suggest handling the varying nature of tenant traffic by assuming that it is slow enough to react to. Alternatively, scheduling the MapReduce shuffle stages was proposed by Orchestra [35]. A generalization of this approach that allows a tenant to describe its changing communication needs is suggested by Coflow [36]. On the same line of thought, scheduling at a finer grain was proposed by Hedera [18]. However, since these schemes propose 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.

Fairness. FairCloud provides a generalization of the required fairness properties of the shared cloud network [37]. 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 through a wide range. They can be completely static and optimized for MPI applications [38, 39], or rely on traffic-spreading techniques like ECMP [40], rely on traffic spray as in RPS or DeTail [41, 42], use adaptive routing as proposed by DAR [43], or even rely on per-packet synchronized schemes like FastPass [44]. LaaS isolates the sub-topology of each tenant, and therefore allows each tenant to use the routing that maximizes its application performance.

3. IMPACT OF TENANT INTERFERENCE

This section presents the impact of concurrent tenant traffic on tenant performance. The presented results are obtained from measurements on real hardware, as well as simulations of InfiniBand and Ethernet networks. We also provide online a full description of the settings and of our code for the experiments [27].

Tenant interference in cluster experiments. The experimental topology is a non-blocking two-level fat-tree with 8 hosts in each of the 4 leaf switches. The leaf switches are fully connected to 4 spine switches, with two parallel links per connection. We assume 4 tenants, and randomly assign 8 dedicated hosts to each of the 4 tenants. The reason for using a random placement is that even a scheduler that follows a bin-packing algorithm is known to show a large degree of fragmentation in steady state [30]. The tenants independently

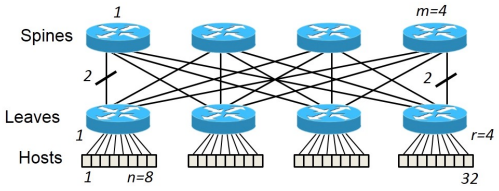


Figure 2: Experimental fat-tree cluster.

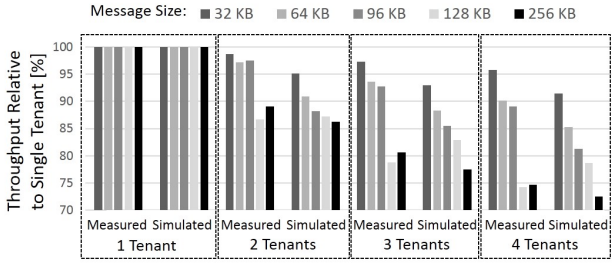


Figure 3: Relative performance, obtained by experiment and simulation, of an application based on all-to-all traffic, for 1–4 concurrent tenants of 8 hosts each. The maximal degradation is about 25%, even for this small cluster of 32 nodes. The full bars on the single tenant runs demonstrate we normalize each run condition separately.

alternate between computation and all-to-all communication, i.e. each node computes new results and sends different data to the rest of the nodes that belong to the same tenant, as a sequence of un-synchronized shift permutations. This traffic pattern is representative of the Shuffle stage of MapReduce, and of scientific-computing applications such as those based on Fast Fourier Transform. We keep the total computation time constant, while the communication time changes with the increasing message size. For a single tenant with 32KB messages, the communication time represents roughly 2/3 of the total time.

Fig. 3 presents the relative application performance in our cluster, measured for various reasonable message sizes [45] and for 1–4 parallel tenants. The results show that even in such a small cluster, the performance of a tenant may degrade (i.e., its run-time may increase) by 25% for large messages when other tenants run concurrently. Larger message sizes degrade the performance due to the larger buffering needs and larger communication time.

Since we also want to analyze the performance of the applications in larger clusters, we further rely on a simulator based on an InfiniBand model [46]. For sanity check, we compare our small cluster measurements with simulated results. The figure illustrates that the simulation results for 4 tenants are about 3% worse, and show the same trend as the experiment. The difference probably results from a lack of accuracy in modeling the MPI computation time, and therefore it would be expected to decrease in larger networks with a more significant network contention.

Tenant interference in scaled-up simulations. We now evaluate the impact of cloud size. As the number of tenants and their sizes grow, we would expect an increased inter-tenant friction, and therefore a degraded application performance in the presence of concurrent tenants. We simulate the effect of the concurrent tenant traffic on a cloud of 1,728 hosts for 8 and 32 randomly-placed tenants, each of 216 and 54 hosts respectively. We measure the average relative performance of a tenant, defined as the ratio of its performance when running concurrently with all other tenants by its performance when running alone. We show the impact of inter-tenant friction on scientific-computing applications as well as on MapReduce. For the scientific-computing benchmark, we select stencil codes, which are parallel programs that break the problem space (mainly 3-dimensional) into sub-spaces, apply the same procedure to each sub-space and exchange data mostly with neighboring sub-spaces. This scheme is common to many scientific programs, and especially those solving partial differential equations, such as weather prediction and flow dynamics. The computation time is again kept constant while the communication time changes with the increasing message size. For a single tenant with 32KB messages, the communication time represents roughly 4/9 of the total time.

Fig. 4 shows how the relative performance of each tenant decreases as the number of tenants and the message size increase. For instance, for 32 concurrent tenants exchanging 32KB messages, the performance degrades by 45% compared to a tenant running alone (equivalently, providing isolation from concurrent tenants would more than double the performance). This significant loss of performance happens despite a modest message size of 32KB, and presents a large source of potential run-time variability. Note that the degradation of performance is clearly a result of network contention, since each job runs on dedicated hosts. MapReduce (simulated at similar conditions) experiences a smaller impact than stencil applications. Interestingly, the smaller interference from other tenants is a result of higher self-contention: due to the Shuffle all-to-all traffic pattern, there is network contention even when MapReduce runs alone. Stencil applications suffer less from self-contention because their traffic matrix is less dense. Our second set of simulations illustrates tenant interference on a partition-aggregate traffic pattern, which is characteristic of distributed database queries run by many Web2.0 services like Facebook [**?**, 42, 47]. We simulate such a traffic pattern on the same cluster, assuming each of the 32 tenants splits its hosts equally

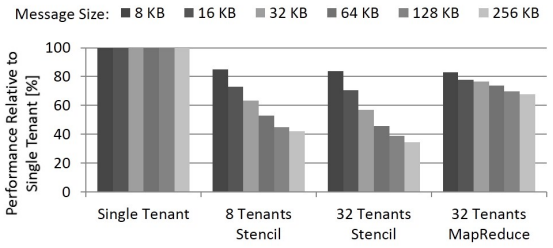


Figure 4: Simulated relative performance for 8 and 32 tenants on a cloud of 1,728 hosts, with Stencil scientific-computing applications or MapReduce-based applications. The relative performance to a single tenant degrades as the traffic volume and the number of concurrent tenants increase.

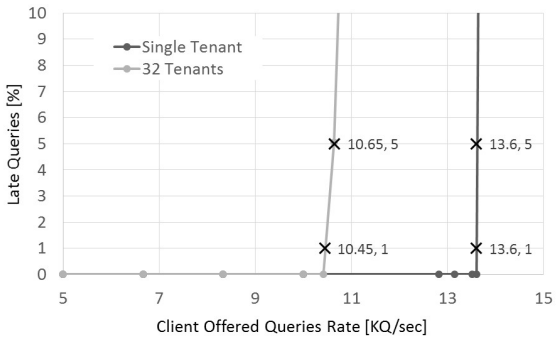


Figure 5: Simulated distributed database tenants placed randomly on 1,728 nodes cluster. The percentage of queries not meeting a 10msec deadline vs. offered query-rate show steep saturation.

between servers and clients. The query arrivals follow a Poisson process with a controllable rate. Each query is sent to all servers in parallel.

Fig. 5 shows the percentage of late queries not meeting a 10-msec deadline. The steep increase of late queries happens at about 10,450 queries per second for the 32 concurrent tenants, versus 13,600 queries per second for a single tenant. The network link sharing resulted in a degradation of about 30% in the effective query rate.

We further want to confirm that similar results are obtained for a lossy Ethernet network. We simulate a 32-node Ethernet cluster employing ECMP routing and DCTCP [47], using an INET [48] simulator enhanced with a specially-implemented DCTCP plugin. We simulate 32 nodes and not 1,728 nodes because this simulator is less scalable. There are only two tenants: The first is a regular 8-node tenant implementing MapReduce, of random Map and Reduce times and variable Shuffle data size (producing a similar ratio of communication

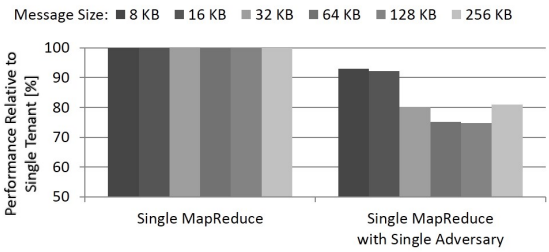


Figure 6: Simulated relative performance of an 8-node MapReduce tenant on a 32-node Ethernet cluster running DCTCP. An adversarial 8-node tenant degrades the performance by 25%.

Figure 7: Simulated distributed database queries 99.9 percentile of query latency for single and 32 tenants. In the presence of other tenants the query latency at 10Kquery/sec is 10 times larger if than running alone.

time to total time). The second is an 8-node adversarial aggressor tenant. Each adversarial node continuously generates 1MB messages, sent in parallel to all its other nodes. We intentionally keep half the nodes unused to illustrate the detrimental impact of other tenants even in an over-provisioned cluster. Fig. 6 presents the relative performance of MapReduce in the presence of the adversarial tenant as compared to its performance when running alone. The worst relative performance is obtained for messages of 128KB, with a degradation of 25% even in such a small and over-provisioned cluster. We suspect that the increase in the last value with 256KB message results from an artifact of DCTCP.

The first job time to complete 100 compute/communicate iterations is measured. The jobs start one after the other with some delay, such that the resulting measurement show a gradual increase of the first job iteration time due to jobs interference. The results are plotted in Fig. 8 which shows a degradation of 20% = 0.18/0.15. Note that on larger systems where the jb sizes are larger and many more jobs exist the expected impact on job run-time is larger.

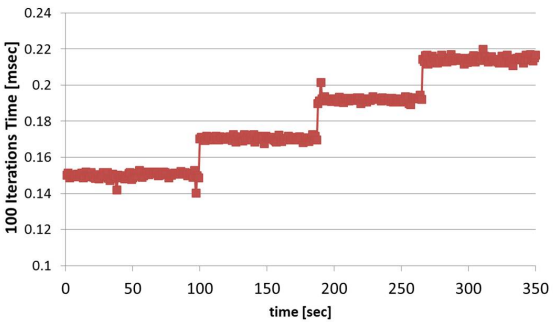


Figure 8: MPI app run-time, on a 32 nodes 10GB/s InfiniBand cluster, degrades by 20% with gradual start of other similar apps.

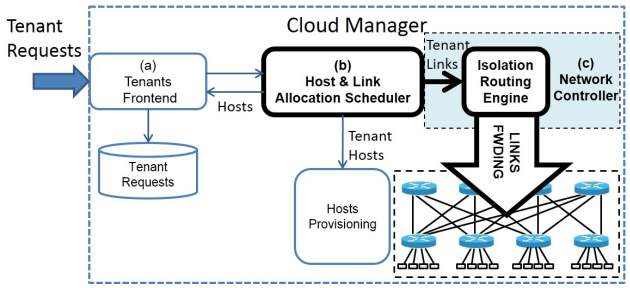


Figure 9: Cloud management system architecture, with LaaS extensions in bold.

4. LAAS ARCHITECTURE

A typical cloud architecture 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 backend). 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. 9 emphasizes these two extensions by bold lines on an abstract cloud management software architecture.

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 [49–51]. But then, tenant run-times would be impacted by new tenants, which is precisely what we want to avoid.

Network controller. As depicted in Fig. 9, 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 [52]. 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 [20]. Then each tenant may incorporate its own SDN controller, which can then only discover its own isolated sub-topology. The other extreme 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 [53]. This isolated-routing feature could also be implemented by Ethernet SDN controllers like OpenDaylight.

5. LAAS ALGORITHM

In this section we describe online algorithms for *tenant placement* and *link allocation* in the LaaS scheduler. By online, versus offline, placement-algorithms, we require existing tenant placement to be maintained when a new job is placed. Similarly we provide online link-allocation algorithms to avoid any traffic interruption when a new tenant is introduced. The algorithm we describe provably guarantee 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 study 2-level fat trees, and then generalize the results to 3 levels. We also first present a *Simple* heuristic

algorithm, and then extend it with a refined *LaaS* algorithm that achieves a better cloud utilization.

5.1 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. 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.

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.

Simple heuristic algorithm. We first introduce a *Simple* heuristic algorithm, against which we later compare our suggested algorithms. 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.

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. 10 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 [54].

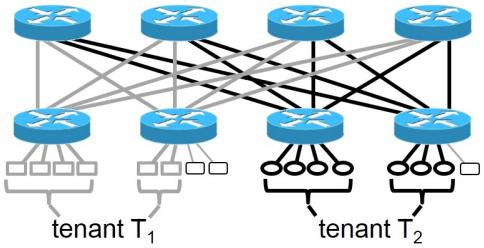


Figure 10: Two tenants of sizes 6 and 7 hosts placed by the *Simple* heuristic, where each tenant fills a number of complete sub-trees.

5.2 Extended Simple Heuristics Single Tenant Leaving the Sub-Tree

An extension of the simple solution is to allow a single tenant with hosts within a sub-tree to span across multiple sub-trees. The same argument used for the simple case, only the traffic of the single tenant leaving the sub-tree is crossing the top level of the sub-tree thus isolation is maintained. Since the entire set of links at this layer must match the number of hosts within that sub-tree the obtained topology supports any admissible traffic matrix.

For example Fig. **??** show how tenant t4 occupies a part of 2-level sub-tree which is shared by tenant t3 extending out of that sub-tree. No traffic other that of t3 would need to leave the same sub-tree and thus all the top links in that tree are allocated to tenant t3.

5.3 The General Case

The simple heuristic and its extension are extreme in their constraints, wasting hosts by rounding job size up. What can we say on the general case of placing the tenant hosts without restriction? We provide here some analysis of the resulting capacity loss caused by the non-blocking requirement.

What are the conditions Link allocation should meet?

PROPERTY 1. $\forall i \in [1, r] : \min(N_i, N - N_i) \leq |S_i|$.

PROOF. There is no need to carry more flows out of the leaf switch than there are other hosts of the tenant. So the number of flows is the minimum between local and total remote hosts. The last are by definition $N - N_i$. Since every link on the bipartite connects to a different spine the number of links is the size of S_i. □

COROLLARY 2. *Corollary of Property 1 is that for most cases where $N \gg N_i$ there need to be at least N_i links allocated on leaf L_i. This also means that when $|S_i| > N_i$ some hosts left on that leaf switch cannot be used for future tenants. We denote this number of wasted hosts:*

$$W_i = |S_i| - N_i \tag{1}$$

PROPERTY 3. $\forall i,j \in [1,r] : \min(N_i, N_j) \leq |S_i \cap S_j|$.

PROOF. Let $c = \min(N_i, N_j)$. There are at most c 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 c common spine switches in $|S_i \cap S_j|$. □

Without loss of generality we sort the N_i such that $N_1 \leq N_2 \leq \cdots \leq N_r$. So Property 1 becomes:

$$\forall i,j \in [1,r] : N_i = \min(N_i, N_j) \leq |S_i \cap S_j| \quad (2)$$

PROPERTY 4. $\forall i,j,k \in [1,r] : \min(N_i + N_j, N_k) \leq |S_i \cup S_j|$.

PROOF. Let $c = \min(N_i + N_j, N_k)$. There are at most c flows going from L_k to L_i or 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 c spines in the union of the two leaves connected spines $|S_i \cup S_j|$. □

THEOREM 5. *The total wasted hosts capacity* $W = \sum_{i=1}^{k} W_i$ *by a host placement* N_i *is lower bounded by:*

$$W = max(\sum_{k=1}^{\lfloor \frac{r-1}{2} \rfloor} min(N_{2k-1}, N_r - N_{2k}), \sum_{k=1}^{\lfloor \frac{r-1}{2} \rfloor} min(N_k, N_r - N_{r-k})) \quad (3)$$

PROOF. Apply triangle in-equation to Property 2:

$$\min(N_i + N_j, N_r) \leq |S_i \cup S_j| = |S_i| + |S_j| - |S_i \cap S_j| \quad (4)$$

Or:

$$|S_i \cap S_j| \leq |S_i| + |S_j| - \min(N_i + N_j, N_r) \quad (5)$$

Further apply Property 1 and obtain:

$$N_i \leq |S_i| + |S_j| - \min(N_i + N_j, N_r) \quad (6)$$

We have two cases:

$$\begin{aligned} N_i + N_j < N_r &: N_i \leq |S_i| + |S_j| - N_i - N_j \\ N_i + N_j \geq N_r &: N_i \leq |S_i| + |S_j| - N_r \end{aligned} \quad (7)$$

And further:

$$\begin{aligned} N_i < N_r - N_j &: N_i \leq W_i + W_j \\ N_i \geq N_r - N_j &: N_r - N_j \leq W_i + W_j \end{aligned} \quad (8)$$

Finally we can merge the two equations

$$\min(N_i, N_r - N_j) \leq W_i + W_j \quad (9)$$

We can now sum the Equation (9) either for all pairs $i,j : j = i+1$ or for furthest pairs: $i,j : j = r - i$. □

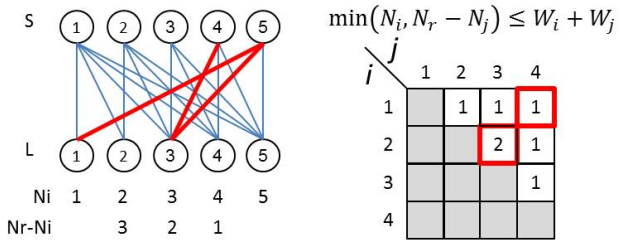


Figure 11: The number of extra links required for providing RNB network for tenant with $N_i = i$. The calculation of lower bound on number of additional cables, and the communication pattern they are required for is depicted on the right table showing $W_i + W_j$

The obtained lower bound is demonstrated by Fig. 11 where 5 leaves are allocated with $N_i = i$. In order to calculate a lower bound of number of additional links. we fill a table of $W_i + W_j$ by evaluation of Equation (9) for each $i,j : j > i$ pairs. Then select the pairs producing the maximal total W. An optional allocation of the extra cables is denoted on the network diagram with bold links. **LaaS placement analysis.** We now want to present some steps towards a tenant placement algorithm that could obtain a better performance than *Simple* by allowing the placement of more than a single tenant on the same leaf, and therefore by reaching a tighter packing. To do so, we first analyze the fundamental conditions for providing LaaS.

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:

$$|S_i| = N_i. \quad (10)$$

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 $N_1 \leq N_2 \leq \cdots \leq N_r$.

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. Due to space constraints, we present the proofs of all results of this paper an online technical report [27].

LEMMA 1. *The number of common spines that connect two leaves must at least equal their minimal number of allocated hosts, i.e.*

$$\forall i < j \in [1, r] : N_i = \min(N_i, N_j) \leq |S_i \cap S_j| \quad (11)$$

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

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.* $\forall i \neq j \neq k \in [1, r] : N_i, N_j, N_k > 0 \rightarrow N_i + N_j \geq N_k$

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: $\forall i, j, k \in [1, r] : \min(N_i + N_j, N_k) = N_i + N_j \leq |S_i \cup S_j| = |S_i| + |S_j| - |S_i \cap S_j|$. But since $N_i = |S_i|$ and $N_j = |S_j|$ in LaaS by Equation (10), 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 $N_i + N_j \geq N_k$. □

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 $N_1 \leq N_2 \leq \cdots \leq N_r$.

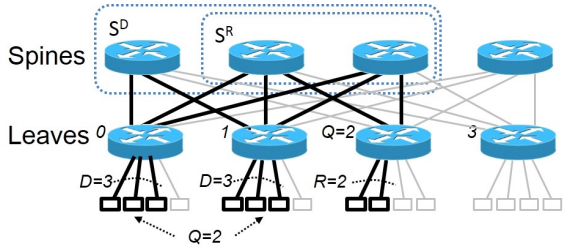


Figure 12: A tenant of $N = 8 = Q \cdot D + R$ hosts. To implement LaaS, there must be Q leaves of D hosts and optionally one leaf of $R < D$ hosts.

THEOREM 6. *A necessary condition for LaaS is*

$$N_1 \leq N_2 = N_3 = \cdots = N_r, \quad (12)$$

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 $N_2 = N_r$. 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 $k = r$, $N_1 + N_2 \geq N_r$ thus $\min(N_1 + N_2, N_r) = N_r$. So, when N_r flows are sent from L_r to L_1 and L_2, we must have at least N_r common spines: $|S_1 \cup S_2| = N_2 \geq N_r$. But since $N_2 \leq N_r$, it follows that $N_2 = N_r$. □

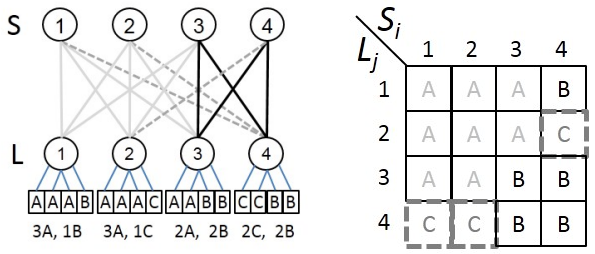
Given Theorem 6, the tenant placement should follow the form: $N = D \cdot Q + R$, where Q is the number of repeated leaves with D hosts each, and we optionally add one unique leaf with a smaller number of hosts R. This notation follows the Divisor, Quotient and Remainder of N. This result is useful because it greatly simplifies the solution of the host placement problem defined above.

Fig. 12 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 7. *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 full bipartite graph with $m' = D$ spine switches and $n' = D$ hosts per leaf, which supports any admissible traffic matrix as mentioned above.



(a) Placement (b) Link Allocation

Figure 13: (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.

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 full-bipartite 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 $r' = Q + 1$. 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 replace 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 6 is not always sufficient to provide a needed link allocation as in Theorem 7. This is why Theorem 7 proves essential.

PROPERTY 8. *A host placement that meets Theorem 6 does not guarantee the existence of a link allocation that meets Theorem 7, and therefore does not guarantee LaaS.*

PROOF. We prove Property 8 by the example provided in Fig. 13. 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 6. 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 Property 8, 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. In the sections ahead, we develop such algorithms for 2- and 3-level fat trees.

5.4 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 [55, 56]. We use the notation of Extended Generalized Fat Trees (XGFT) [57], 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 Section 5.1 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 $R_l = \sum_{i=1}^{l} m_i$. 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:

$$l_{min} = \min \{l | (R_{l-1} < N) \wedge (R_l \geq N)\}, \qquad (13)$$

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 is potentially unacceptable to

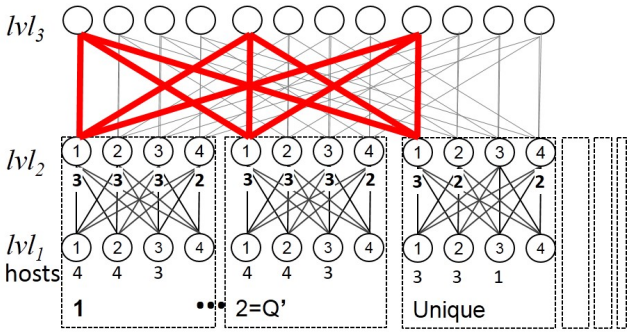


Figure 14: An example 3-level fat tree matching LaaS requirements. The number of flows injected from each sub-tree to the top bipartite graphs is marked in bold below the lvl_2 switches.

cloud vendors, so we keep looking for a better one.

***Hierarchical* decomposition.** A hierarchical decomposition describes 3-level fat trees as collections of 2-level trees. For a LaaS link allocation to be feasible, the condition of Theorem 6 needs to hold also for the top level of the tree. We denote the switches on the tree by their levels (from bottom up) lvl_1, lvl_2 and lvl_3. To enable LaaS, within each of the lower levels of the tree, tenants may only be allocated hosts with $N_{lvl_2} = Q \cdot D + R$. Note that an allocation that fits in a single leaf switch also follows this scheme with $Q = 1$.

Fig. 14 depicts a host allocation that follows the form: $N_{lvl_2} = Q \cdot D + R$ of Theorem 6 and Fig. 12 in each level-2 sub-tree. The two left sub-trees use an allocation of $11 = 2 \cdot 4 + 3$ hosts each, while the right sub-tree holds $7 = 2 \cdot 3 + 1$ hosts. Now consider one of the bipartite graphs at the top of the tree (emphasized in the drawing). We show that allocating isolated links while maintaining support for any admissible traffic matrix on this bipartite graph is similar to the allocation done on a single sub-tree.

First, we obtain the number of flows entering/leaving the top bipartite graph. For instance, in the left sub-tree of Fig. 14, the tenant uses all the 4 lvl_2 spines to avoid network contention. The number of flows entering the lvl_2 switches of the left sub-tree, i.e. the number of lvl_1 switches with allocated links to these lvl_2 switches, is 3, 3, 3 and 2, respectively. This is because there are only 2 lvl_1 switches that have 4 assigned hosts and thus need to use all the lvl_2 switches; while the 3^{rd} lvl_1 switch only has 3 assigned hosts, and thus does not use the 4th (rightmost) lvl_2 switch. Generally, each one of the R lvl_2 switches needs to support $Q+1$ flows from/to the $Q+1$ lvl_1 switches, and $D-R$ lvl_2 switches need to support just Q flows. Thus, each of the top full bipartite graphs needs to support either $Q_i + 1$ or Q_i flows entering from each of the various sub-trees.

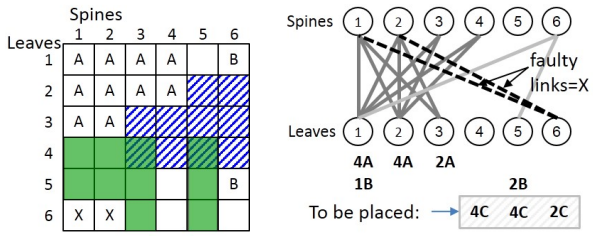
We apply Theorem 6 to each of the full bipartite graphs made of lvl_3 and lvl_2 switches (like the bold line in Fig. 14). Accordingly, in order to fulfill LaaS requirements, it is required that the number of flows entering the bipartite graph from the lvl_2 switches should be equal, except for maybe one lvl_2 switch that must carry fewer flows. Therefore, we should have Q' repeated sub-trees of $N_{lvl_2} = Q \cdot D + R$ hosts, and possibly one additional sub-tree of $\bar{N}_{lvl_2} = \bar{Q} \cdot \bar{D} + \bar{R}$ hosts. There are exactly Q bipartite graphs with Q' sub-trees injecting $Q + 1$ flows, and $D - R$ with Q' sub-trees injecting Q flows. Similarly, the unique remaining sub-tree injects $\bar{Q} + 1$ flows into $\bar{R}$ bipartite graphs, and $\bar{Q}$ flows into $\bar{D} - \bar{R}$ bipartite graphs.

Finally, since $\bar{D}$ may be bigger, smaller or equal to D, we conclude that the condition of Theorem 6 translates to: (a) $D > R$, (b) $\bar{D} > \bar{R}$, and (c) if $R > \bar{R} : Q \geq \bar{Q}$, else $Q \geq \bar{Q} + 1$. Each tenant job of size N hosts may then be decomposed into: $N = Q'(Q \cdot D + R) + \bar{Q} \cdot \bar{D} + \bar{R}$.

***Approximated* algorithm.** The general decomposition approach described in the previous section leads to a huge space of possible placements and link allocations for any incoming tenant. Instead of treating this entire space, we propose to simplify the 3-level tree problem with an *Approximated* approach. If a tenant cannot fit within a single sub-tree, we round up its size to the nearest complete number of hosts contained under a lvl_1 switch, i.e. m_1 in the XGFT definition. 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 maximal number m_1 of flows leaving each leaf is identical to the number of lvl_2 switches in the sub-tree, all top-level bipartite graphs have the exact same maximal number of entering flows, and their solution is symmetrical. This means that we can represent the 3-level fat tree as a 2-level tree (a single top bipartite graph), and solve in the same way we did in the 2-level case. We have effectively limited the placement to $D = \bar{D} = m_1$ and $R = \bar{R} = 0$. The tenant job of size N is first rounded to $N' = m_1\lceil N/m_1 \rceil$ hosts, and then placed as: $N' = m_1(Q' \cdot Q + \bar{Q})$.

Final *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 15(a). The required set of leaves and links is of the form $N = Q \cdot D + R$. As described in Section 5.3, 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



(a) Link Allocation Table (b) Corresponding Topology

Figure 15: (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 $Q \cdot R + D = 2 \cdot 4 + 2$, can be assigned one of two allocations. In (a), the first link allocation is shown in solid, and the second with slanted lines.

the 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 the first successful allocation, so trying the most-used leaves first packs the allocations and achieve the best overall utilization results.

Fig. 15 demonstrates the process of evaluating 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 leafs. The extension to L2 allocation is trivial. 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 leafs that were already found, $\{ports\}$ the set of ports that are possible for this allocation, $\{rl\}$ the collected set of, so far, Q size leafs. Eventually the recursion provides the following results: $\{D_L\}$ set of leafs with Q hosts, $\{D_{PORTS}\}$ the set of ports to be used by the Q size leafs, U_L the unique, sized R, leaf and $\{U_{PORTS}\}$ the ports on that leaf. The overall algorithm for 3 level fat trees is provided in Algorithm 2.

6. EVALUATION

Our evaluation is reported in three sub-sections. The

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

find next Q size leaf
for $i = l$ **to** l_e **do**
  if $|M[i]| >= Q$ **then**
    $\{nPorts\} = \{ports\} \cap M[i]$
    if $|nPorts| \geq Q$ **then**
      $\{newRL\} = \{rl\} \cup i$
      if $r = D$ **then**
        found all repeated leafs
        if findUniqueLeaf$(R, l_s, l_e; \{nPorts\}\{rl\})$ **then**
          $\{D_{PORTS}\} = \{nPorts\}$
          $\{D_L\} = \{newRL\}$
          return true
        end if
      else
        if FLAP$(i{+}1, l_e, r{+}1, \{nPorts\}, \{newRL\})$ **then**
          return true
        end if
      end if
    end if
  end if
end for
return false

Algorithm 2 LAAS(N)

Try 2 level allocation first
if $N < m_1 \cdot m_2$ **then**
  for $D = max(N, m_1)$ **to** 1 **do**
    $Q = \lfloor \frac{N}{D} \rfloor$
    $R = N - D \cdot Q$
    if $FLAP(0, m_1 \cot m_2 - 1, 0, 1, \{\}, \{\})$ **then**
      return true
    end if
  end for
end if
$U = \lceil \frac{N}{m_1} \rceil$
for $D = max(U, m_2)$ **to** 1 **do**
  $Q = \lfloor \frac{U}{D} \rfloor$
  $R = U - D \cdot Q$
  if $m_3 \geq Q$ **then**
    if $FLAP2(0, m_3 - 1, 0, 1, \{\}, \{\})$ **then**
      return true
    end if
  end if
end for
return false

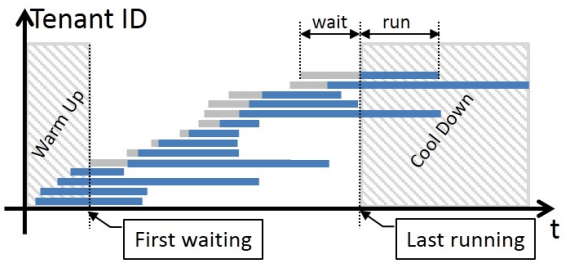


Figure 16: Utilization is measured after the first tenant cannot enter the cloud and before the cloud starts draining out of tenants.

first deals with the different placement algorithms and compares their resulting *cloud utilization*. 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.

6.1 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. Since smaller tenants are easier to place, for any tenant size distribution, not letting smaller tenants bypass those waiting means that we favor fairness over cloud utilization. Thus, the result should be regarded as an intuitive lower-bound for a real-life cloud utilization.

All host placement heuristics need to eventually select a possible placement from many valid ones. We follow a bin-packing approach that considers the most-used leaf switches first (i.e., those with the least number of available hosts). This is known to minimize fragmentation [58]. We also tested other placement-order policies, like preferring a larger spread of the tenants, but they produced inferior results.

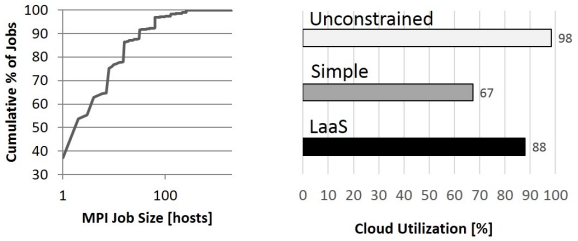
Simulation settings. We simulate the largest full-bisectional-bandwidth 3-level fat-tree network that can be built with 36-port switches, i.e. a cloud of 11,662 hosts. The evaluation uses a randomized sequence of 10,000 tenant requests. A random run-time in the range of 20 to 3,000 time units is assigned to each tenant. The variation of run-time makes scheduling harder as it increases fragmentation.

We evaluate 3 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, an exponential distribution of variable parameter x. It is truncated between 1 and the cloud size. Last, a Gaussian distribution of average parameter x and standard deviation parameter $\frac{x}{5}$, truncated again within the cloud size. The use of x allows a variation that is proportional to the average tenant size.

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 5.

Simulation results. We analyzed the scheduling logs of the Julich JUROPA cluster, a large scientific-computing cloud of 3,288 hosts, over the period of 2010 and half 2011. Fig. 17(a) illustrates the CDF of the tenant MPI (message passing interface)-based job sizes that were run on the cloud over this period. 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. 17(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. 18 illustrates the cloud utilization for *Unconstrained*, *Simple*, and *LaaS*, is plotted as a function of the exponential parameter x, which is close to the aver-



(a) (b)

Figure 17: (a) Measured job-size CDF for the Julich JUROPA scientific-computing cloud. (b) Resulting cloud utilization. *LaaS* achieves 88%.

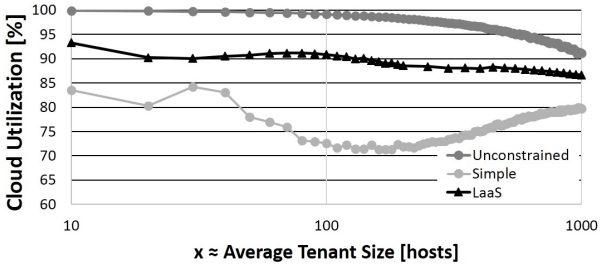


Figure 18: Cloud utilization for a truncated exponential distribution of tenant host sizes in a cloud of 11,662 hosts.

age 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 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. Fig. 19 illustrates the cloud utilization for the truncated Gaussian distribution. This distribution provides a harder test for the allocation algorithm, since tenant sizes are made similar, and they may be just beyond the above-mentioned thresholds of a leaf size (18 hosts) or a sub-tree size (324 hosts). These thresholds are where *LaaS* and the *Simple* are less efficient when compared to *Unconstrained*. *Simple* suffers from a particularly large fluctuation in utilization. *LaaS* is more stable over the entire range, with about 90% uti-

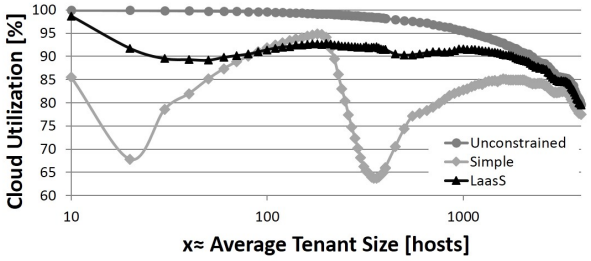


Figure 19: Cloud utilization for a truncated Gaussian distribution $\mathcal{N}(x, x/5)$ of tenant host sizes in a cloud of 11,662 hosts.

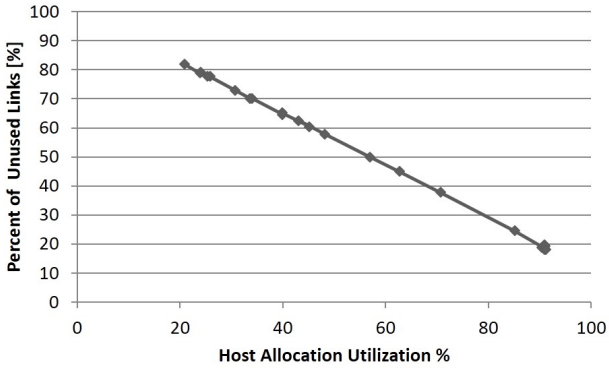


Figure 20: Percentage of links that can be turned off in the 3-level fat tree as a function of the cloud utilization.

lization. There are also a few points where the *Simple* heuristic provides a better utilization than *LaaS*. 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.

Another aspect of LaaS is the knowledge of exact links being used in the network. The calculated percentage of links not utilized, that could be turned off, is provided in Fig. 20 for the LaaS Placement Approximation run with the Julich distribution of tenant sizes on the large network. As can be observed, the average percentage of links that could be turned off is linear with the cloud utilization. As the utilization decreases the number of unused links grows accordingly and the network power can be linearly reduced.

6.2 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.

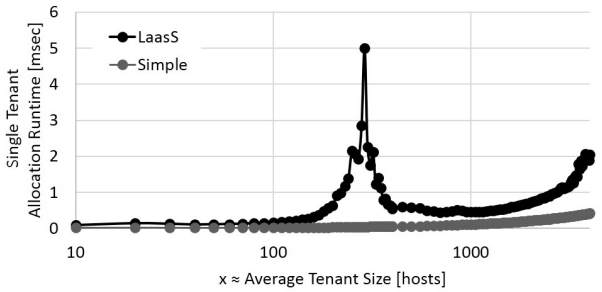


Figure 21: Average run-time per single tenant allocation for Gaussian(x,x/5) tenant sizes.

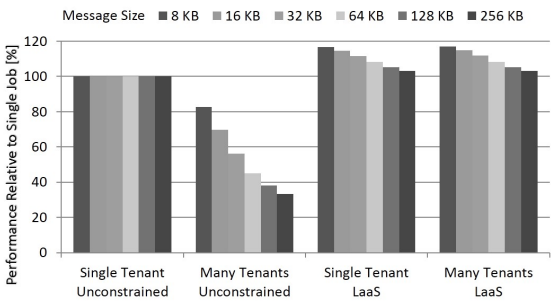


Figure 22: Simulated relative performance for tenants running Stencil scientific-computing applications on a cloud of 1,728 hosts, either alone or as 32 concurrent tenants. While tenant performance degrades when placed unconstrained (without link isolation), the performance of single and multiple tenants with LaaS appears identical, fulfilling the promise of LaaS.

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 (OpenSM), 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. We ran the SDN controller over the simulated network of 1,728 hosts, as well as over our 32-host experimental cluster.

Run-time. The LaaS Approximation scans through all possible placements for valid link allocation. This involves evaluating all possible combinations. Fig. 21 presents the average run-time per tenant request as 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.

6.3 Evaluation of Tenant Performance

Given a full LaaS architecture implementation, we have rerun the same traffic scenarios presented in Section 3, and observed that the tenant run-time looks indeed independent of other tenants, thus achieving our goal of isolation.

Fig. 22 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. It is an extension of Fig. 4. The figure illustrates many effects. First, as already seen in Fig. 4, 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.

7. 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 [26]. This is because it explicitly mentions which links and switches are to be used, and therefore can turn off other links and switches.

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 rate limits can be enforced by the host (a feature supported by most modern network cards), LaaS could treat each host as if it were two or more hosts by a simple change of topology description. Then each partition of the host can be allocated independently.

LaaS with oversubscription. LaaS can also handle partial-bisectional-bandwidth fat-trees (with a reduced bisectional bandwidth closer to the roots). Assuming the bandwidth is reduced by a factor p, each link allocated should carry p flows. As long as the number D of hosts allocated to the tenant on each leaf is a multiple of p, we can simply divide D by p and apply the *LaaS* algorithm. Note that on most fat trees, p is a divisor of the number of ports anyway, and the extra constraint on D only has a small effect on cloud utilization.

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 [30].

Energy Proportionality is another benefit of LaaS. Since links are explicitly allocated, those links left unallocated can be proactively turned off and save power, as opposed to only partially throttled down [26] when traffic is load balanced over the network. We show that with LaaS the number of used links is proportional to the cloud utilization.

8. 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.

9. REFERENCES

[1] G. Linden, "Make data useful," 2006.

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

[3] D. Artz, "The secret weapons of the aol optimization team," in *Velocity Conference*, 2009.

[4] H. Ballani *et al.*, "Towards predictable datacenter networks," in *SIGCOMM*, 2011.

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

[6] J. C. Mogul and L. Popa, "What we talk about when we talk about cloud network performance," *SIGCOMM*, 2012.

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

[8] A. Iosup *et al.*, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE TPDS*, 2011.
[9] S. Ostermann *et al.*, "A performance analysis of EC2 cloud computing services for scientific computing," ser. LNICST. Springer, Jan. 2010.
[10] K. LaCurts *et al.*, "Cicada: Introducing predictive guarantees for cloud networks," in *USENIX HotCloud*, 2014.
[11] M. Chowdhury, M. R. Rahman, and R. Boutaba, "ViNEYard: Virtual network embedding algorithms," *IEEE/ACM ToN*, 2012.
[12] A. Jokanovic *et al.*, "Effective quality-of-service policy for capacity high-performance computing systems," in *IEEE HPCC*, 2012.
[13] R. Doriguzzi Corin *et al.*, "VeRTIGO: Network virtualization and beyond," in *EWSDN*, 2012.
[14] V. T. Lam *et al.*, "Netshare predictable bandwidth allocation for data centers," *SIGCOMM*, 2012.
[15] H. Rodrigues *et al.*, "Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks." in *WIOV*, 2011.
[16] K. C. Webb, A. C. Snoeren, and K. Yocum, "Topology switching for data center networks," in *Hot-ICE Workshop*, 2011.
[17] C. Guo *et al.*, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *ACM CoNext*, 2010.
[18] M. Al-Fares *et al.*, "Hedera: Dynamic flow scheduling for data center networks." *NSDI*, 2010.
[19] A. Shieh *et al.*, "Seawall: performance isolation for cloud datacenter networks," in *USENIX HotCloud*, 2010.
[20] R. Sherwood *et al.*, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.
[21] M. Yu *et al.*, "Rethinking virtual network embedding," *SIGCOMM*, 2008.
[22] B. Raghavan *et al.*, "Cloud control with distributed rate limiting," *SIGCOMM*, 2007.
[23] "OpenStack ironic," https://wiki.openstack.org/wiki/Ironic.
[24] A. Jokanovic *et al.*, "Impact of inter-application contention in current and future HPC systems," in *IEEE HOTI*, 2010.
[25] T. Ristenpart *et al.*, "Hey, you, get off of my cloud," *ACM CCS*, 2009.
[26] D. Abts *et al.*, "Energy proportional datacenter networks," *SIGARCH Comput. Archit. News*, 2010.
[27] "LaaS technical report, code, experiments and simulation conditions," https://www.dropbox.com/sh/ovy3ag255twqpln/AABM4OhNXFmVzAirrFsa3Doma/.
[28] J. Schad, J. Dittrich, and J.-A. Quian-Ruiz, "Runtime measurements in the cloud," *VLDB Endowment*, 2010.
[29] J. Orduna, F. Silla, and J. Duato, "A new task mapping technique for communication-aware scheduling strategies," in *ICPP Workshops*, 2001.
[30] J. A. Pascual, J. Navaridas, and J. Miguel-Alonso, "Effects of topology-aware allocation policies on scheduling performance," in *Job Scheduling Strategies for Parallel Processing*, 2009.
[31] A. Bhatele *et al.*, "There goes the neighborhood: Performance degradation due to nearby jobs," *ACM SC*, 2013.
[32] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6d mesh/torus interconnect for exascale computers," *Computer*, 2009.
[33] A. Altn, H. Yaman, and M. . Pnar, "The Robust Network Loading Problem Under Hose Demand Uncertainty," *INFORMS Journal on Computing*, 2010.
[34] V. Jeyakumar *et al.*, "EyeQ: Practical Network Performance Isolation at the Edge," in *USENIX NSDI*, 2013.
[35] M. Chowdhury *et al.*, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM*, 2011.
[36] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *HotNets*. ACM, 2012.
[37] L. Popa *et al.*, "FairCloud: Sharing the network in cloud computing," in *SIGCOMM*, 2012.
[38] E. Zahavi, "Fat-tree routing and node ordering providing contention free traffic for MPI global collectives," *JPDC*, 2012.
[39] Y. Gong, B. He, and J. Zhong, "Network performance aware MPI collective communication operations in the cloud," *IEEE TPDS*, 2013.
[40] C. E. Hopps, "Analysis of an equal-cost multi-path algorithm," http://tools.ietf.org/html/rfc2992, 2015.
[41] A. Dixit *et al.*, "On the impact of packet spraying in data center networks," in *IEEE Infocom*, 2013.
[42] D. Zats *et al.*, "DeTail: reducing the flow completion time tail in datacenter networks," *SIGCOMM*, 2012.
[43] X. Wu and X. Yang, "DARD: Distributed adaptive routing for datacenter networks," in *IEEE ICDCS*, 2012.
[44] J. Perry *et al.*, "Fastpass: A centralized zero-queue datacenter network," *SIGCOMM*, 2014.
[45] R. Rabenseifner *et al.*, "The parallel effective i/o bandwidth benchmark: b eff io," *Parallel I/O for Cluster Computing*, 2001.
[46] J. Domke, T. Hoefler, and W. E. Nagel,

"Deadlock-free oblivious routing for arbitrary topologies," in *IEEE IPDPS*, 2011.
[47] M. Alizadeh *et al.*, "Data center TCP (DCTCP)," *ACM SIGCOMM*, 2010.
[48] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," *Simutools*, 2008.
[49] J. Jiang *et al.*, "Joint VM placement and routing for data center traffic engineering," in *IEEE Infocom*, 2012.
[50] X. Wen *et al.*, "VirtualKnotter: Online virtual machine shuffling," in *IEEE ICDCS*, 2012.
[51] "Virtual machine migration comparison VMWARE VSPHARE vs. hyper-v," 2011. [Online]. Available: \url{http://www.vmware.com/files/pdf/vmw-vmotion-verus-live-migration.pdf}
[52] R. Perlman, "Transparent interconnection of lots of links (trill)," in *IETF RFC 5556*, 2009.
[53] "Mellanox OFED user manual v2.3-1.0.1," http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v2.3-1.0.1.pdf, section 3.2.2.5.7 Routing Chains.
[54] A. Jajszczyk, "Nonblocking, repackable, and rearrangeable clos networks," *Communications Magazine, IEEE*, 2003.
[55] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *SIGCOMM*, 2008.
[56] A. Andreyev, "Introducing data center fabric, the next-generation facebook the next generation datacenter network," 2014.
[57] S. Ohring *et al.*, "On generalized fat trees," in *IPPS*, 1995.
[58] R. Grandl *et al.*, "Multi-resource packing for cluster schedulers," in *SIGCOMM*. ACM, 2014.

APPENDIX

A. APPENDIX

A. LAAS SOFTWARE RELEASE 1.0

The software is provided in [27] under the directory laas_1.0/ as well as in a single archive file: laas_1.0.tgz. In this section, we provide all the information required to get the LaaS service installed, and instructions to run a demonstration of the service. We also provide the simulation setup used for obtaining the cluster utilization, run-time and correctness.

The simulator and a service of LaaS are coded in Python and are built on top of the core algorithm coded in C++. At the heart of the package is the LaaS algorithm coded in *isol.cc*. It is using facilities specific to 3-level fat-trees provided in *ft3.cc* and port-mask utility class in *portmask.cc* used for tracking availability of links. The *laas.cc* implements the service API provided in *laas.h* and exposed in Python using SWIG which uses the declarations in *laas.i*. We provide a scheduling simulator, to obtain cluster utility, in *sim.py* and a tenant allocation service in *laas_service.py*.

The LaaS service provides a RESTful interface and serves tenant requests [**?**]. It outputs OpenStack command files required to control tenant host placement and also provides SDN configuration files to enforce isolation via packet routing/forwarding.

The scheduling simulator takes a CSV file with tenant requests (id, size and arrival time) and process them in a FIFO manner.

A.1 License

Due to the double-blind review this release of the software is intended for SIGCOMM reviewers only. Please do not distribute. Once we can add the copyright, this software will be provided with a choice of GPLv2 or BSD license and published on our website.

A.2 Content

The following sub-directories are included in this release:

- *src* - The core algorithm c++ and python service/simulator
- *bin* - Random tenants generator and isol.log checker
- *examples* - A set of files used by the demo below

Out of the entire set of source files, the one most interesting for integration is lass.h which provides the API exposed in Python.

A.3 Software dependency

- Any Linux environment, for example Ubuntu 12.04
- SWIG Version 2.0.11
- Python 2.7
- Python 2.7 Flask 0.7
- Python 2.7 Flask Restful 0.3.1

The Perl code (for utilities only) depends on:

- Perl v5.18.2
- Perl Math::Random 0.71
- Perl Math::Round 0.07

A.4 Installation

```
tar xvfz laas_1.0.tgz
cd laas_1.0/src
make
```

A.5 Running LaaS Service

1. *Choose your cluster topology:*
   For ease of review we choose a small 2 level fat tree. The example topology is XGFT(2; 4,8; 1,4). Due to limitation of the current implementation we represent it as if it were a 3 level fat tree with one top switch: XGFT(3; 4,8,1; 1,4,1) The data needed to run a larger topology is also included in the examples directory.
2. *Prepare name mapping file:*
   The LaaS engine eventually needs to configure OpenStack and an SDN controller that rely on physical naming and port numbering and not on general fat-tree indexing. A file that provides mapping of the tree level, index within the tree and port indexing to the actual cluster hardware is thus required. For this example topology we provide the mapping file: examples/pgft_m4_8_w1_4.csv.
   The first line hints at the content of each column:

   ```
   # lvl,swIdx,name,UP,upPorts,DN,dnPorts
   ```

   The example line below describes a host, providing its level is 0 and index is 10, its name is comp-11 and it has a single UP port, number 1, connecting to L1 switch (on level 1).

   ```
   0,10,comp-11,UP,1,,,,,,,,,,,,,,,,,,,,,,
   ```

   An example L1 switch line is provided below. See this is the 4th switch in L1, its name as recognized by the SDN controller is SW_L1_3 and its ports 5-8 are connecting to hosts:

   ```
   1,3,SW_L1_3,UP,1,2,3,4,DN,5,6,7,8
   ```

   Note: The file does not include any mapping for the non existant level 3 switches.
3. *Start the service:*
   Once started the LaaS service reports its address and port. The Restful API is up and any change in tenant status will result in updates in the OSCfg/ and SDNCfg/ directories.

   ```
   |$ python ./src/laas_service.py  -m 4,8,1 -w 1,4,1 \
       -n examples/pgft_m4_8_w1_4.csv
   |-I- Defined 64 up ports and 64 down port mappings
   |
   |* Running on http://127.0.0.1:12345/
   |* Restarting with reloader
   |-I- Defined 64 up ports and 64 down port mappings
   ```

4. *Run a demo:*
   We provide here an example sequence of calls to the service. After each step we discuss the results and the created files if any.

A.5.1 List tenants:

```
| $ curl http://localhost:12345/tenants
| {}
```

As expeted it returns an empty list

A.5.2 Create a tenant of 10 nodes:

(Expecting it will span 2.5 leafs.)

```
 $ curl http://localhost:12345/tenants -d "id=4" -d "n=10" -X POST
| {
| "N": 10,
|  "hosts": 10,
|   "l1Ports": 10,
| "l2Ports": 0
| }
```

See how the tenant-id may be any number for which there is no pre-existing tenant in the system. Let's inspect the created files. First see the new file in the OSCfg:

```
cmd-1.log:
| #!/bin/bash
| #
| # Adding tenant 4 to OpenStack
| #
| echo Adding tenant 4 to OpenStack > OSCfg/cmd-1.log
| keystone tenant-create --name laas-tenant-4 \
|   --description "LaaS Tenant 4" >> OSCfg/cmd-1.log
| tenantId=`keystone tenant-get laas-tenant-4 | \
|   awk '/ id /{print $4}'` >> OSCfg/cmd-1.log
| nova aggregate-create laas-aggr-4 >> OSCfg/cmd-1.log
| nova aggregate-set-metadata laas-aggr-4 \
|   filter_tenant_id=$tenantId >> OSCfg/cmd-1.log
| nova aggregate-add-host laas-aggr-4 comp-1 >> cmd-1.log
| nova aggregate-add-host laas-aggr-4 comp-2 >> cmd-1.log
| nova aggregate-add-host laas-aggr-4 comp-3 >> cmd-1.log
| nova aggregate-add-host laas-aggr-4 comp-4 >> cmd-1.log
| nova aggregate-add-host laas-aggr-4 comp-5 >> cmd-1.log
| nova aggregate-add-host laas-aggr-4 comp-6 >> cmd-1.log
| nova aggregate-add-host laas-aggr-4 comp-7 >> cmd-1.log
| nova aggregate-add-host laas-aggr-4 comp-8 >> cmd-1.log
| nova aggregate-add-host laas-aggr-4 comp-9 >> cmd-1.log
| nova aggregate-add-host laas-aggr-4 comp-10 >> cmd-1.log
```

Similarly, the SDNCfg/ directory now holds a full set of configuration files required for OpenSM to configure the network. We will not go through the full description of these files but focus on the groups.conf. This file now holds the definition of the hosts and switch ports used by the first tenant:

```
| port-group
| name: T4-hcas
| obj_list:
|    name=comp-1/U1:P1
|    name=comp-2/U1:P1
|    name=comp-3/U1:P1
|    name=comp-4/U1:P1
|    name=comp-5/U1:P1
|    name=comp-6/U1:P1
|    name=comp-7/U1:P1
|    name=comp-8/U1:P1
|    name=comp-9/U1:P1
|    name=comp-10/U1:P1;
| end-port-group
|
| port-group
| name: T4-switches
| obj_list:
|    name=SW_L1_2/U1 pmask=0x6
|    name=SW_L1_0/U1 pmask=0x1e
|    name=SW_L1_1/U1 pmask=0x1e;
| end-port-group
```

A.5.3 To fill in the network we create another 10-node tenant:

```
| $ curl http://localhost:12345/tenants -d "id=1" -d "n=10" -X POST
| {
| "N": 10,
|  "hosts": 10,
|   "l1Ports": 10,
| "l2Ports": 0
| }
```

A.5.4 List again the tenants:

```
| $ curl http://localhost:12345/tenants
| {
|     "1": {
|         "N": 10,
|         "hosts": 10,
|         "l1Ports": 10,
|         "l2Ports": 0
|     },
|     "4": {
|         "N": 10,
|         "hosts": 10,
|         "l1Ports": 10,
|         "l2Ports": 0
|     }
| }
```

A.5.5 Get the allocated hosts and links for a specific tenant:

```
| $ curl http://localhost:12345/tenants/1/hosts
| [
|     "comp-11",
|     "comp-12",
|     "comp-13",
|     "comp-14",
|     "comp-15",
|     "comp-16",
|     "comp-17",
|     "comp-18",
|     "comp-19",
|     "comp-20"
| ]
```

As expected the four spines are going to be used (all up ports of 2 leafs) and only 2 ports of the leaf SW_L1_2 holding just 2 nodes.

```
| $ curl http://localhost:12345/tenants/1/l1Ports
| [
|     { "pNum": 3, "sName": "SW_L1_2" },
|     { "pNum": 4, "sName": "SW_L1_2" },
|     { "pNum": 1, "sName": "SW_L1_3" },
|     { "pNum": 2, "sName": "SW_L1_3" },
|     { "pNum": 3, "sName": "SW_L1_3" },
|     { "pNum": 4, "sName": "SW_L1_3" },
|     { "pNum": 1, "sName": "SW_L1_4" },
|     { "pNum": 2, "sName": "SW_L1_4" },
|     { "pNum": 3, "sName": "SW_L1_4" },
|     { "pNum": 4, "sName": "SW_L1_4" }
| ]
```

A.5.6 A bad request example:

Now let's see what happens if we try to over-provision the cluster by requesting a tenant of $13 = 32 - 20 + 1$ hosts:

```
| $ curl http://localhost:12345/tenants -d "id=2" -d "n=13" -X POST
| {
|     "message": "Fail to allocate tenant 2"
| }
```

A.5.7 Delete tenant 1:

```
| $ curl http://localhost:12345/tenants/1 -X DELETE
```

We now have a command file under OSCfg/ that deletes the OpenStack tenant and aggregate

```
| #!/bin/bash
| #
| # Removing tenant 1 from OpenStack
| #
| nova aggregate-remove-host laas-aggr-1 comp-11 >> cmd-3.log
| nova aggregate-remove-host laas-aggr-1 comp-12 >> cmd-3.log
| nova aggregate-remove-host laas-aggr-1 comp-13 >> cmd-3.log
| nova aggregate-remove-host laas-aggr-1 comp-14 >> cmd-3.log
| nova aggregate-remove-host laas-aggr-1 comp-15 >> cmd-3.log
| nova aggregate-remove-host laas-aggr-1 comp-16 >> cmd-3.log
| nova aggregate-remove-host laas-aggr-1 comp-17 >> cmd-3.log
| nova aggregate-remove-host laas-aggr-1 comp-18 >> cmd-3.log
| nova aggregate-remove-host laas-aggr-1 comp-19 >> cmd-3.log
| nova aggregate-remove-host laas-aggr-1 comp-20 >> cmd-3.log
| nova aggregate-delete laas-aggr-1 >> OSCfg/cmd-3.log
| keystone tenant-delete laas-tenant-1 >> OSCfg/cmd-3.log
```

A.5.8 Retry allocating the 13 nodes tenant:

```
| $ curl http://localhost:12345/tenants \
|    -d "id=2" -d "n=13" -X POST
| {
|     "N": 13,
|     "hosts": 13,
|     "l1Ports": 13,
|     "l2Ports": 0
| }
```

A.6 Running Simulation of LaaS algorithm

In this section we provide instruction for the simulation of a LaaS engine handling a large number of tenant requests. The procedure provided here is similar to the one used to obtain the results in the paper. In the paper, we also used a scheduler that implements the *Simple* and the *Unconstrained* algorithms.

1. *Choose your cluster topology:*
   For example the maximal full bisection 3 level XGFT with 36 port switches is: XGFT(3; 18,18,36; 1,18,18) It has 11,628 hosts, 648 L1, 648 L2 and 324 L3 switches.
2. *Generate a set of tenant requests:*
   We do that by running the utility bin/genJobsFlow: For this example we use an exponential distribution with an average of 8 hosts. The tenant run time is uniformly distributed in the range [20,3000]. Please try –help to see other possible options.

   ```
   | ./bin/genJobsFlow -n 10000 -s 8 -r 20:3000 -a 0 > \
       examples/exp=8_tenants=1000_arrival=0.csv
   ```

3. *Run the simulator*
   After we have prepared the tenant requests file and decided about the topology we can run:

   ```
   | $ python ./src/sim.py -m 18,18,36 -w 1,18,18 \
   |          -c examples/exp=8_tenants=1000_arrival=0.csv
   | -I- Obtained 10000 jobs
   | -I- first waiting job at: 20 lastJobPlacementTime 10623
   | -I- Total potential hosts * time = 1.23673e+08
   | -I- Total considered jobs: 9976 skip first: 0 last: 24
   | -I- Total actual hosts * time = 1.17281e+08
   | -I- Host Utilization = 94.83 %
   | -I- L1 Up Links Utilization  = 38.36 %
   | -I- L2 Up Links Utilization  = 10.70 %
   | -I- Total Links Utilization  = 48.40 %
   | -I- Run Time = 14.2 sec
   ```

   The details of each allocation/deallocation are provided in the log file: isol.log. Each line describes one transaction and contains the total hosts/links as well as their detailed indices within the topology.

4. *Check that the results are legal:*
   The checker needs to know the topology size. So it requires this info on the command line:

   ```
   checkAllocations -n/--hosts-per-leaf n
                    -k/--num-l1-per-l2 m2
                    -1/--total-l1s t1
                    -2/--total-l2s t2
                    -3/--total-l3s t3
                    -l/--log log-file
   ```

   ```
   | $ ./bin/checkAllocations -n 18 -k 18 -1 648 -2 648 -3 324 \
   |        -l isol.log
   | -I- Checked 10000 ADD and 8760 REM jobs
   | -I- Added/Rem 35573/30117 L1PORTS and 567/482 L2PORTS
   ```

B. EXPERIMENTAL SETUP

B.1 Hardware

The experiment was run on the 32-node cluster presented in Fig. 2. The hosts are of two types:

- 30 hosts are HP ProLiant DL320e G8 E3-1220v2 B120i 2x1Gb 1x8GB 1x500GB HOT PLUG DVD-RW 350W 3Y. Each containing 4-core Intel Xeon CPU E3-1220 V2 at 3.10GHz.
- 2 hosts are IBM System x3450 servers featuring Intel Xeon processors 2.80 GHz and 3.0 GHz/1600 MHz, with 12 MB L2, and 3.4 GHz/1600 MHz, with 6 MB L2.

The InfiniBand NICs are: MHQH19-XTC Single 4X IB QDR Port, PCIe Gen2 x8, Tall Bracket, RoHS-R5 HCA Card, QSFP Connector. The InfiniBand switches are: MIS5024Q-1BFR 36-port non-blocking 40Gb/s unmanaged Switch System.

B.2 Software

The machines run Scientific Linux release 6.5 (Carbon). The MPI used is mvapich2-2.0rc2. Our experiment uses a simple MPI program that executes an MPI_AllToAll collectives or 2 dimensional stencil communications using ISend/IRecv followed by MPI_Barrier. The programs are provided under the sub-directory **mpi_experiment**. This directory also holds the RUN script that was used to invoke each of the 4 tenants MPI applications with a delay after invoking the previous. The host files used are also included.

C. ETHERNET SIMULATIONS

For simulation of an Ethernet-based topology we used an enhanced iNET framework. We base our code on iNET 2.2 and extend it with DCTCP modules. The switch forwarding is also enhanced with ECMP-like forwarding with a hash function that works modulo $(SrcHostIndex + DstHostIndex, NumberOfUpPorts)$. The parameters used by the simulator are described in the following Table 1.

The application used to generate the MapReduce Shuffle stage is an application that runs Scatter and

Parameter	Value	Description
MACRelayUnitPP.bufferSize	65,536	Per port buffer size, meaning total buffer size = bufferSize*numRealPorts
MACRelayUnitPP.processingTime	3.00E-07	Switch processing delay
TCP.advertisedWindow	65,535	Receiver window of TCP
TCP.delayedAcksEnabled	false	No delayed ACKS
TCP.minRexmitTimeout	0.3	Minimal retransmission timeout
TCP.mss	1452	TCP MSS
TCP.nagleEnabled	true	TCP parameter
TCP.tcpAlgorithmClass	DCTCPNewReno	DCTCP based on NewReno is used
TCPScatterGatterClientApp.idleInterval	exponential(200us)	Time between successive Shuffles (computation time)
TCPScatterGatterClientApp.reconnectInterval	1.00E-06	Time to setup the new connection
TCPScatterGatterClientApp.replyLength	2	Resolver reply just ACK
TCPScatterGatterClientApp.requestLength	65,536	Example data size of 64KiB from Mapper to Resolver

Table 1: Ethernet model (iNET) parameters and their values

then Gather from a list of nodes. To mimic the Shuffle, the Scatter provides parallel send of the Mapper data size and the Gather is of size 2 bytes only.

The tenants are placed on hosts numbered:
Tenant 1: 7, 3, 25, 18, 0, 13, 24, 12
Tenant 2: 8, 9, 20, 29, 6, 1, 28, 5
Tenant 3: 11, 4, 16, 21, 2, 17, 22, 14
Tenant 4: 19, 15, 10, 27, 31, 26, 23, 30

D. INFINIBAND SIMULATIONS

The InfiniBand simulation utilizes Mellanox published model [**?**]. We have enhanced this model with an application that relies on MPI semantics and is able to replay MPI traces. The parameters used for our simulation are provided in Table 2.

The tenants that are placed on the 1,728-node cluster are of the sizes:

- Two tenants: the two are 810 and 834.
- Eight tenants: all are 216 nodes.
- Thirty two tenants: all are 54 nodes.

The tenants execute cycles of computation and communication. The computation time is of uniform distribution in the range $[5, 15]\mu sec$. So the traffic to computation ratio for Stencil application exchanging 32KB of data on each dimension is: Calculation = $10\mu sec$. Communication = $\frac{32KB}{4GB/sec} = 8\mu sec$. So the ratio of ideal computation to communication is 10/8 for 32KB exchanges. For all-to-all shuffles we increase the computation time to be uniform in the range $[20, 80]\mu sec$, but the data is sent to each other node in the tenant. So for a 32KB exchange on an 8-node tenant, the ratio of computation to ideal communication time is $\frac{50}{7*8} = 50/56$.

Module.Parameter	Value	Description
IBGenerator.flit2FlitGap	0.001	A gap inserted between flits [nsec]
IBGenerator.flitSize	64	The flit size (IB credit is 64 bytes)
IBGenerator.genDlyPerByte	2.5e-10	Speed of generating bytes [sec/B]
IBGenerator.maxContPkts	10	Maximum number of continuous packets of same application
IBGenerator.pkt2PktGap	0.001	Gap inserted between packets [nsec]
IBGenerator.popDlyPerByte	2e-10	speed of popping up data to next layer [sec/B]
IBInBuf.maxBeingSent	3	Switch speedup - number of parallel packets being drained from input buffer
IBInBuf.maxStatic0	800	Buffer size [credits]
IBInBuf.maxVL	0	Maximal VL simulated
IBInBuf.width	4	Link withs is 4 lanes
IBOutBuf.credMinTime	0.256	Maximal time between credit updates [usec]
IBOutBuf.maxVL	0	Maximal VL simulated
IBOutBuf.size	66	Host output buffer size [B]
IBOutBuf.size	78	Switch port output buffer size [B]
IBSink.flitSize	64	The flit size (IB credit is 64 bytes)
IBSink.hiccupDelay	1e+06	The receiver may hiccup for 1usec
IBSink.hiccupDuration	0.0001	Length of a hiccup
IBSink.maxVL	0	Maximal VL simulated
IBSink.popDlyPerByte	2.5e-10	Speed of removing Bytes to the PCIe
IBVLArb.busWidth	24	Input bus width of the switch arbiter
IBVLArb.coreFreq	250,000,000	Switch core frequency
cModule.ISWDelay	50	Intrinsic latency of the switch input buffer [nsec]
cModule.VSWDelay	50	Intrinsic latency of the switch arbiter [nsec]

Table 2: InfiniBand model parameters and their values