NetHint: White-Box Networking for Multi-Tenant Data Centers

Jingrong Chen   Hong Zhang[†]   Wei Zhang   Liang Luo[#]   Jeffrey Chase   Ion Stoica[†]   Danyang Zhuo

Duke University   [†]*UC Berkeley*   [#]*University of Washington*

Abstract

A cloud provider today provides its network resources to its tenants as a black box, such that cloud tenants have little knowledge of the underlying network characteristics. Meanwhile, data-intensive applications have increasingly migrated to the cloud, and these applications have both the ability and the incentive to adapt their data transfer schedules based on the cloud network characteristics. We find that the black-box networking abstraction and the adaptiveness of data-intensive applications together create a mismatch, leading to sub-optimal application performance.

This paper explores a white-box approach to resolving this mismatch. We propose NetHint, an interactive mechanism between a cloud tenant and a cloud provider to jointly enhance application performance. With NetHint, the provider provides a *hint* — an *indirect indication* of the underlying network characteristics (e.g., link-layer network topologies for a tenant's virtual machines, number of co-locating tenants, network bandwidth utilization), and the tenant's applications then adapt their transfer schedules accordingly. The NetHint design provides abundant network information for cloud tenants to compute their optimal transfer schedules, while introducing little overhead for the cloud provider to collect and expose this information. Evaluation results show that NetHint improves the average performance of allreduce completion time, broadcast completion time, and MapReduce shuffle completion time by 2.7×, 1.5×, and 1.2×, respectively.

1 Introduction

Data-intensive applications (e.g., network functions, data analytics, deep learning) have increasingly moved to the cloud for resource elasticity, performance, security, and ease of management. The performance of the cloud network is critical for these applications' performance. Cloud providers have thus spent significant effort to optimize various aspects of cloud networks, including network topology [34, 73, 76], congestion control and network stack [3, 33, 42, 44, 69, 77, 92], load balancing [2, 46, 63, 88], bandwidth guarantee [6, 9, 43, 48, 51, 67], debugging [7, 31], fault recovery [53], hardware [8, 27, 52, 58], and virtualization [66].

Today, a cloud provider exposes the network to its tenants as a black box: the cloud tenants have little visibility into their expected network performance (e.g., a constant worst-case bandwidth assurance) or the underlying network characteristics including the link-layer network topology, number of co-locating tenants, and instantaneous available bandwidth.

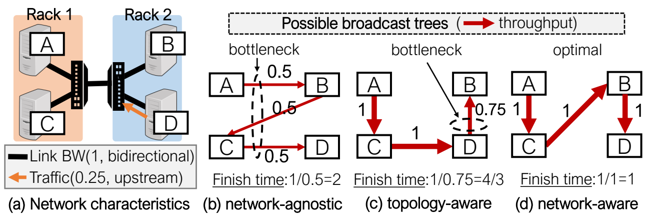


Figure 1: **Applications have the ability and the incentive to adapt their transfer schedules based on network characteristics:** Consider broadcasting a unit-size data object from VM A to VM B, C, and D. (a) shows the network characteristics, all links have bidirectional bandwidth of 1. VM D has upstream background traffic of 0.25. (b) to (d) show possible broadcast trees and their corresponding broadcast finish time. The arrows represent traffic flows and the numbers represent the throughput.

The black-box model has worked well for decades due to its simplicity. However, with the emergence of popular data-intensive applications (e.g., data analytics, distributed deep learning, and distributed reinforcement learning) in the cloud, we observe that such a black-box model is no longer efficient (§2). The crux is that many of these emerging applications have both the *ability* and the *incentive* to adapt their transfer schedules based on the underlying network characteristics, but it is difficult to do so with a black-box network.

Consider broadcast, an important communication primitive in reinforcement learning and ensemble model serving. Figure 1 shows an example that VM A broadcasts to VM B to VM D. Figure 1b shows a possible broadcast tree constructed under the black-box model. Without the underlying network characteristics, the broadcast tree is *network-agnostic*, which introduces link stress on the cross-rack link. Figure 1c shows a broadcast tree based on the topology information (i.e., topology-aware), which improves the broadcast finish time from 2 to $\frac{4}{3}$ time units by minimizing the cross-rack traffic. Figure 1d shows a broadcast tree based on both the topology and bandwidth information (i.e., network-aware). It builds an optimal broadcast tree that avoids the congested upstream link on VM D, further improving the finish time to 1 time unit. The performance gains increase for data center networks that have larger oversubscription ratios or more skewed traffic.

The above example illustrates a fundamental *mismatch* between the black-box nature of existing network abstractions and the ability of a data-intensive application to adapt its traffic. With the black-box model, the cloud tenant is unaware of the network characteristics, and the cloud provider is unaware of the application communication semantics and the transfer schedule. This misses an opportunity for the cloud tenants and the cloud provider to adapt the data flows to the underlying network topology and conditions to enhance performance and efficiency for these applications. The potential gains are substantial: our benchmark experiment on AWS shows that the allreduce latency for a deep learning experiment varies by up to 2.8× across different allreduce transfer schedules. One candidate approach is for applications to probe and profile the network and then plan their data flows accordingly [5, 57]. A second option is to report their possible transfer schedules to the provider for the provider to choose. We observe that these alternatives introduce substantial communication latency and system overhead (§2.2).

In this paper, we explore a *white-box* approach to resolve this mismatch. One possibility would be for the cloud provider to expose the physical network topology, the VM locations, along with bandwidth assurances to the application. However, this approach has two major drawbacks. First, exposing VM placement and data center network topology may compromise security for cloud tenants and can raise concerns for the cloud provider (§2). Second, the bandwidth available to a tenant depends on the communication patterns of other tenants, which may be highly dynamic. Predictions that are not timely or not accurate may do more harm than good.

This paper explores an alternative approach. We design and implement NetHint, a mechanism for a cloud tenant and cloud provider to interact to enhance the application performance jointly. The key idea is that the provider provides a *hint* — an *indirect indication* of the bandwidth allocation to a cloud tenant (e.g., a virtual link-layer network topology, number of co-locating tenants, network bandwidth utilization). The tenant applications then adapt their transfer schedules based on the hints, which may change over time. NetHint balances confidentiality and expressiveness: on one hand, the hint avoids exposing the physical network topology or traffic characteristics of other tenants (§9). On the other hand, we show that the hint provides sufficient network information to enable tenants to plan efficient transfer schedules. (§5).

The effectiveness of NetHint relies on addressing three important challenges. First, *what information should the hint contain?* We provide each cloud tenant with a virtual link-layer network topology along with available bandwidth on each link in the virtual topology. This allows applications to adapt their transfer schedules to avoid network congestion.

The second challenge is *how to provide this hint at a low cost*. We design a two-layer aggregation method to collect network statistics on the hosts. We designate a NetHint server in a rack to aggregate network characteristics in the rack. NetHint servers then use all-to-all communication to exchange network characteristics globally. A cloud tenant can thus query its rack-local NetHint server for hints.

The final challenge is *how should applications react to the hint*. We present several use cases for NetHint to optimize communication in a range of popular data-intensive applications including deep learning, MapReduce, and

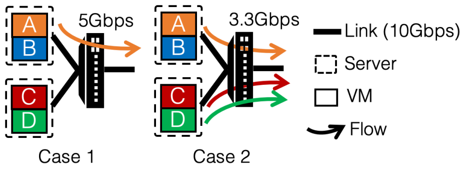


Figure 2: **Examples to illustrate the black-box networking abstraction: tenants cannot predict their network performance.** VM A to D are placed in two servers. All links have 10 Gbps bandwidth. We assume bandwidth is statically partitioned on the end host (each VM can get at most 5 Gbps).

serving ensemble models. The takeaway is that for all these applications, tenants can use the NetHint information via simple scheduling algorithms. Adaptation also has a downside: hints can be stale and adapting transfer schedules based on stale information can hurt performance. We design a policy for applications to adapts flexibly with different hints in different scenarios: applications use temporal bandwidth information when background network conditions are stable and adaptation overhead is low, and otherwise applications fall back to using only the time-invariant topology information (§6).

We evaluate the overheads and the potential performance gain of having NetHint in data centers using a small testbed and large-scale simulations. Our results show that NetHint speeds up the average performance of allreduce completion time in distributed data-parallel deep learning, broadcast completion time in ensemble model serving, and MapReduce shuffle completion time in distributed data analytics by 2.7×, 1.5×, and 1.2×, respectively. Moreover, these benefits are cheap to obtain: NetHint incurs modest CPU, memory, and network bandwidth overheads.

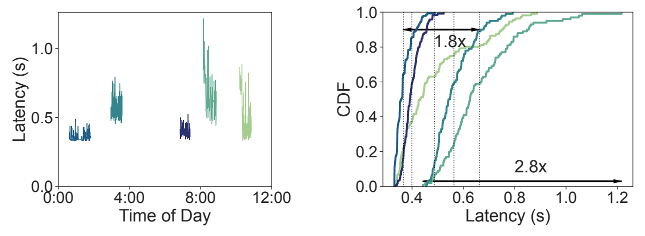
In summary, this paper makes the following contributions:

- We identify a mismatch between the current black-box network abstraction and the communication needs of data-intensive applications.
- We explore a white-box networking approach for multi-tenant data centers.
- We design and implement NetHint, a low-cost system to allow data-intensive applications to adapt their data transfer schedules to enhance performance.

2 Background

2.1 Black-Box Networking Abstraction

Today, the networking abstraction a cloud has is merely a per-VM bandwidth allocation at the end hosts. The abstraction is a *black box*: tenants are unaware of the underlying network characteristics including network topology, number of co-locating tenants, and instantaneous available bandwidth. As a result, the cloud tenants cannot predict their network performance. Figure 2 shows an example. Even with a static allocation of 5 Gbps per VM, VM A cannot predict its network performance because it depends on the traffic demand of other VMs. VM A can get only a bandwidth of 3.33 Gbps when



(a) Allreduce latency across time (b) CDF

Figure 3: Empirical allreduce (256MB) latency of 5 trials. Two trials may have different VM allocations spatially, and each trial contains 100 consecutive runs. (a) shows 5 trials over different times of a day. In (b), each line is the latency CDF of a trial. Each vertical line is the mean latency for a trial. Allreduce latencies vary both across time (up to 2.8x) and across VM allocations (up to 1.8x).

two flows of VM C and D cause congestion inside the network (case 2). Even with work-conserving bandwidth guarantees, a VM's network performance depends on other VMs.

To quantify this effect, we benchmark allreduce latency on Amazon Web Service (AWS). Allreduce is a collective communication primitive that is commonly used for distributed deep learning. It aggregates a vector (i.e., gradient updates in deep learning) across all worker processes (each running in its own VM). In our experiment, we launch 32 g4dn.2XL (with Linux kernel 5.3) instances in the EC2 US-East-1 region and test ring-allreduce latency with NVIDIA NCCL (version 2.4.8)—the most popular collective communication library for deep learning—for 100 consecutive runs. We repeat the above experiment for 5 trials, and different trials may have different VM placements on the physical topology. Figure 3 shows our findings: ring-allreduce performance on 256MB buffer varies both spatially across different trials and temporally within a trial. Comparing across different trials, the fastest trial has a $1.8\times$ better mean performance than the slowest trial; comparing the 100 runs within a trial, the fastest run is up to $2.8\times$ faster than the slowest run.

2.2 Adaptiveness in Data-Intensive Applications

Besides reinforcement learning and ensemble model serving, which can broadcast model and input data adaptively, as illustrated in Figure 1, we show that many other applications also have both the ability and incentive to adapt their transfer schedules based on the underlying network characteristics.

Many distributed data analytics workloads contain network-intensive shuffle phases between different job stages. For example, the shuffle in MapReduce applications creates an all to all data transfer between the map and reduce stages. The shuffle phase accounts for a large portion of the execution time for many data analytics workloads [16], and numerous studies [4, 15, 16, 39, 84, 87, 90] have demonstrated that optimizing shuffle performance significantly improves application performance. Given network characteristics, distributed data analytics applications can change their transfer schedules (by changing the task placement) to minimize shuffle completion time. Figure 4a shows the shuffle traffic for a MapReduce job with two mappers (m1 and m2) and two reducers (r1 and r2). We observe from Figure 4b to Figure 4d that allocating mappers and reducers based on the topology and bandwidth information effectively improves this shuffle completion time from 4 to 2 units. Moreover, emerging task-based distributed systems (e.g., Ray, Dask, Hydro) support applications with dynamic task graphs. Similar to the MapReduce example, we can change the transfer schedule of these applications by choosing different VMs to place a task.

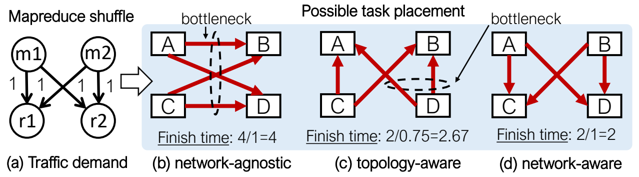


Figure 4: **MapReduce jobs can adapt transfer schedules via task placement.** Assume the same network characteristics as in Figure 1a. (a) shows the traffic demand for a MapReduce shuffle. Each arrow represents a unit traffic. (b) to (d) show possible task placement and the corresponding shuffle finish time.

Moreover, many deep learning jobs are network-intensive. This claim is validated by numerous recent studies [14, 35, 40, 71, 86] and observations from production clusters (e.g., Microsoft [30, 41, 82] and ByteDance [65]). In particular, as mentioned in §2.1, deep learning jobs contain an allreduce phase to synchronize gradient updates among workers in each training iteration. As shown in Figure 5, an allreduce phase has multiple candidate topologies. For example, the allreduce traffic can be sent via a ring connecting all the workers with a flexible ordering (Figure 5a and Figure 5b). Or, we can build an allreduce tree to (1) aggregate gradient updates to one of the workers, and (2) send the aggregated gradient updates back in the reverse direction (Figure 5c and Figure 5d). Different allreduce topologies introduce different transfer schedules. Thus, given network characteristics, deep learning jobs can change their transfer schedules by selecting the algorithm and configuration of allreduce.

2.3 Addressing the Mismatch

The black-box nature of the existing networking abstraction and the adaptiveness of data-intensive applications create a mismatch. Data-intensive applications would benefit from more network information from the cloud provider to configure their transfer schedules, but black-box networking hides this information.

Solutions based on the black-box abstraction. There are two approaches to address this mismatch without modifying the existing black-box networking abstraction. One possible approach is to let the cloud provider optimize the communication for tenants as a cloud service. To this end, we first have to develop a general networking API for cloud tenants to express their communication semantics, traffic loads and optimization objectives to the cloud provider. The API design should be similar to the coflow abstraction [16] or the virtual cluster ab-

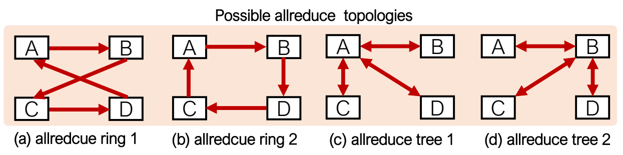


Figure 5: **Allreduce can be performed with different topologies**. (a) to (d) show 4 possible allreduce topologies to perform allreduce among the 4 VMs (workers).

straction [9], but more general to support a large variety of possible traffic patterns and user-defined objectives. Moreover, a recent measurement study [78] shows that major public clouds exhibit high bandwidth variability at a time granularity of seconds. Thus it is hard, if not impossible, for the cloud provider to perform timely network scheduling for thousands of tenants in a centralized manner, while ensuring network SLAs (e.g., defined via the networking API) for each tenant respectively.

Another potential approach is for cloud tenants to run extensive performance profiling in their allocated VMs [29, 49, 57]. For example, PLink [57] probes the VM pair-wise bandwidth and latency with DPDK and uses K-means clustering to reverse engineer the underlying network topology. This allows it to achieve high allreduce performance by choosing a good allreduce algorithm. Choreo [49] uses 3-step measurements to pinpoint congested links in the data center network to schedule data analytics workloads. Similar approaches were explored decades ago on Internet traffic routing on wide-area overlay networks [5]: picking a high-performance Internet path based on user measurement. Unfortunately, this approach is both costly, as each tenant/user has to profile the network independently, and slow, because the probing phase delays the start of the application. The PLink authors told us that they use 10000 packets to determinte bandwidth between a pair of hosts. Choreo generates 3 minutes of probe traffic to infer the network characteristics for 10 VMs.

A white-box network abstraction? Given the deficiencies of the two black-box based approaches, we instead explore a white-box approach: the provider reveals essential information about the network characteristics to the tenant, and the tenants then optimize their transfer schedules accordingly.

One possible way to achieve this objective is for the cloud provider to reveal to a tenant the location of each VM in the physical link-layer network topology, and estimate available bandwidth between each of the VM-pairs. However, this method can raise security and competitive issues. First, exposing VM allocations in the physical network introduces privacy risks for cloud tenants. For example, a malicious user can locate a targeted tenant's VMs and perform attacks. Second, the exposed VM allocation information can raise competitive concerns for the cloud provider. For instance, this information might be valuable for competitors to learn a cloud provider's scheduling policies, thus, lowering its competitive advantage. Third, the bandwidth a tenant can acquire depends on the transfer schedules of *all* the tenants, and a single change in transfer schedule of one tenant may trigger a recalculation

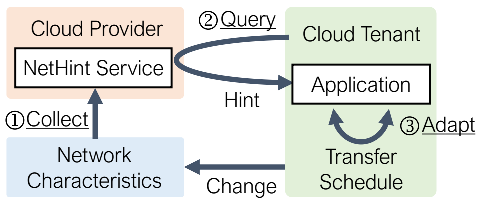


Figure 6: NetHint overview. NetHint service collects network characteristics. Cloud tenants poll hints from NetHint service and adapt their transfer schedules.

for all the tenants. As such, it is computationally expensive for the cloud provider to update the bandwidth shares in real time. Moreover, an application's bandwidth also depends on *its own* transfer schedule. For example, in Case 2 of Figure 2, if VM A sends one extra flow, the total egress bandwidth of VM A increases to 5 Gbps[1]. As a result, without knowing a tenant's transfer schedule, the cloud provider cannot provide accurate bandwidth estimates to its tenants.

3 NetHint Overview

NetHint is an interactive mechanism between a cloud tenant and a cloud provider to jointly enhance the application performance. The key idea is that the provider provides a *hint* — an *indirect indication* of the underlying network characteristics (e.g., a virtual link-layer topology for a tenant's VMs, number of co-located tenants, network bandwidth utilization) to a cloud tenant. As illustrated in Figure 6, the provider provides a NetHint service, which periodically (100 ms by default) collects the hint information to capture changes of the underlying network characteristics. A tenant application can query the NetHint service to get the hint information, and then adapt its transfer schedules based on this provided hint. Note that NetHint does not change the fairness mechanism of the underlying network. A tenant can opt in/out any time — whether or not to use NetHint will *not* affect its fair share of the network.

The hint provides a white-box network abstraction which includes additional network information to tenants. As such, users can infer their best transfer schedule without substantial probing latency or communication overhead with the provider. The hint exposes neither the physical network topology nor the location of a tenant's VMs within it (e.g., which racks). Compared with providing bandwidth information, the hint relieves the provider from the burden of calculating accurate bandwidth allocations. Moreover, compared with calculating bandwidth allocation, it is easier to acquire accurate hint messages (e.g., a virtual link-layer topology for a tenant's VMs, number of co-located tenants, network bandwidth utilization). As such, the provider is free from the potential risk of providing inaccurate information.

We require NetHint to be: (1) *readily deployable:* all the mechanisms are implementable using commodity hardware; (2) *low cost:* the cloud provider can collect network characteristics with minimal CPU, memory, and bandwidth

[1]Assume per-flow fair sharing in the network.

overheads; (3) *useful:* data-intensive workloads can leverage the hints to achieve high performance. To achieve these goals, NetHint's design and implementation must address three questions. First, what hints should be provided to the tenants? Second, how should cloud providers collect the hints with low cost? Third, how should applications use the hints to adapt their transfer schedules?

NetHint describes a virtual link-layer topology that connects a tenant's VMs. In addition, NetHint provides to the tenant recent utilization summaries and counts of co-locating tenant connections on shared network links in the virtual topology. This information allows the tenant to adapt its transfer schedules based on both the topological and temporal hot spots in the network. Further, our design ensures that the only additional information NetHint exposes is aggregated network statistics across all tenants. It is thus difficult for a tenant to acquire information about any individual other tenant. (§4.1)

For the second question, our preferred approach to collecting hints is to measure network traffic in the physical switches using network telemetry, e.g., sketching [54, 55]. However, sketches depend on specific programmable switch features, which are not widely deployed. Instead, our prototype employs a host-driven approach, in which each machine monitors local flows and transmits flow-level statistics to a NetHint measurement plane. One machine in each rack runs a *NetHint server* process to aggregate the rack-level information. These NetHint servers exchange information using periodic all-to-all communication. A cloud tenant connects to the local NetHint server to fetch hints. We show that this approach allows NetHint to provide timely hints to tenants with low CPU and bandwidth overheads (§4.2).

As for the third question, we consider two aspects of adaptation in response to the hints. First, we observe that the adaptation algorithm should take into account the application transfer schedule and semantics to maximize the performance gain. To this end, we consider several use cases for NetHint which cover a range of popular data-intensive applications, including (1) choosing allreduce algorithms in distributed deep learning, (2) constructing broadcast trees for serving ensemble models, and (3) placing tasks in MapReduce frameworks. For each case, we show how applications can adapt their transfer schedules based on the information in the hint. The takeaway is that for all of these examples, tenants can make use of the NetHint information via simple scheduling algorithms (§5).

Second, we explore the drawbacks of adaptation: it introduces extra computational overhead, and may be ineffective or even harmful or unstable if network conditions change too rapidly. We conclude that the adaptation algorithm should use different sets of hints depending on network changing frequency and adaptation overhead. For example, we find that if an application has a non-negligible latency to collect hints and compute the transfer schedules, the bandwidth information may be stale and thus may negatively affect the application performance (detailed in §6). Based on this intuition, we design a policy for applications to react to hints in a flexible manner: under stable network conditions and low adaptation overheads, applications use both bandwidth and topology information to maximize the performance gain of adaptation. Otherwise, applications use only the stable topology information (§6).

Notations & Descriptions	
$\mathcal{T}$	A virtual topology connecting all the tenant's VMs
l	A virtual link in virtual topology $\mathcal{T}$
B_e^l	A tenant's bandwidth share on link l
B_t^l	Total bandwidth on link l
B_r^l	Residual bandwidth on link l
n^l	Number of shared objects on link l

Table 1: Notations and descriptions for NetHint.

4 Providing NetHint Service

4.1 What Is in the Hint?

NetHint exposes a virtual link-layer topology $\mathcal{T}$ to a cloud tenant. The tenant's virtual topology abstracts the network as a tree data structure in which the tenant's VMs are leaf nodes. A link in the tree represents one or more physical links in the data center network, and an interior node may abstract a region of switches and links. The prototype uses a three-layer tree that captures how VMs are distributed among racks in a data center and collapses the network structure above the rack level into a single root node. VMs residing in the same rack are in the same subtree. The virtual topology abstraction does not reveal racks or servers where the tenant has no presence. Following the common observation that congestion losses often occur at the rack level [12, 43, 60, 89], these virtual topologies in the NetHint prototype ignores congestion at any structure above the rack level [23]. It is possible to represent more structure by adding layers to the tree. The tree approximation presumes that the data center network is able to balance its load, so that traffic among children of an abstract node see similar available bandwidth. There is a rich literature on efficient network load balancing for data centers [2,21,22,26,28,36,46,47,63,88], and some of them are readily deployable with commodity hardware.

NetHint allows applications to react to temporal hot spots in the network. For this purpose, NetHint exposes an estimate of utilization on each virtual link l. Recall Case 1 in Figure 2, now assume the orange flow from VM A uses only 2 out of 10 Gbps. If the tenant of VM B knows the network utilization information, it can infer that VM B can send traffic at 8 Gbps. As such, NetHint provides (1) the total bandwidth B_t^l and (2) the residual bandwidth B_r^l on each virtual link l. However, we find that this information alone is insufficient for an application to adapt its transfer schedule, especially when links are congested. For example, even if one link l has already reached 100% utilization, a tenant can still send flows through l and get a fair bandwidth share.

Shared objects and fairness models. In fact, the bandwidth share depends on the fairness model implemented by the cloud provider. Per-flow-fairness and per-VM-pair-fairness are enforced naturally for RDMA-based networks because modern RDMA NICs can be configured to choose either of them. Per-flow-fairness is ensured for containerized clouds because cloud users cannot modify the kernel TCP stack. For traditional TCP-based and VM-based clouds, many recent studies [18, 37, 62] describe how to enforce per-VM-pair-fairness. With the increasing programmability of modern switches, it now becomes possible to implement other fairness models in the network [74, 83], such as per-tenant fairness.

Consider an application placing 3 connections on a 100 Gbps network link with 7 existing connections from 3 other tenants. We assume each flow can reach 100 Gbps throughput. With per-flow fairness model, the application should get 30 Gbps bandwidth. With per-tenant fairness model, the application should get 25 Gbps bandwidth.

The example indicates that the bandwidth share also depends on the number of *shared objects* on each link l. The definition of shared object depends on the fairness model: it is a flow (VM-pair, tenant) under per-flow (per-VM-pair, per-tenant) fairness, respectively.

To provide bandwidth information, NetHint exposes the number of shared objects n^l on each link l. Taken together, NetHint provides a tuple (n^l, B^l_t, B^l_r), which includes both the current link utilization and the number of shared objects.

Bandwidth estimation. The information in the virtual topology enables a tenant to estimate its available bandwidth on each virtual link l efficiently. More formally, consider a tenant who plans to place k^l shared objects on link l in its transfer schedule. If link l is an in-network link in virtual topology $\mathcal{T}$ (i.e., not attached to any VM), the bandwidth share the tenant gets can be estimated as:

$$B^l_e = \max\left(\frac{k^l}{n^l+k^l} B^l_t, B^l_r\right) \tag{1}$$

Equation 1 indicates that when the link is under-utilized, the tenant can use up all the residual bandwidth B^l_r, and even if the link is already congested, the tenant can at least achieve its fair share based on the number of shared objects.

If link l is an edge link (i.e., attached to one VM), the bandwidth share is also affected by the underlying sharing approach. More specifically, denote the per-VM bandwidth guarantee provided by the sharing approaches as B_v, we have:

$$B^l_e = \begin{cases} \min(B_v, \max(\frac{k^l}{n^l+k^l} B^l_t, B^l_r)) & \text{static partitioning} \\ \max(\frac{k^l}{n^l+k^l} B^l_t, B^l_r, B_v) & \text{work-conserving} \end{cases} \tag{2}$$

Sources and impact of inaccuracy We acknowledge that both Equation 1 and Equation 2 are approximations and can sometimes be inaccurate. First, some shared objects (i.e., tenant, VM-pair, or connection) may have traffic demands less than their fair network share, thus calculating the exact value of B^l_e requires knowing the traffic demand for each shared object. NetHint does not provide per-object information, as doing so introduces security concerns and significant overhead given the huge number of such objects. Second, since a virtual link corresponds to the aggregation of multiple parallel paths in the physical topology, the estimation may be inaccurate under poor network load balancing across these parallel paths. We note that this is less likely to happen with recently proposed data center network load balancing designs.

Despite these inaccuracies in bandwidth estimation, our results (§8) show that even the three-level tree approximation is sufficient to adapt the transfer schedules and improve the performance of our target applications. Moreover, evaluation results also show that the benefits degrade gracefully with the quality of the approximations.

Alleviating security and competitive issues. Compared with a naive white-box solution that exposes VM allocation information and physical network topology, NetHint has alleviated the security and the competitive concerns. First, NetHint does not expose the physical location of allocated VMs, so a tenant cannot learn the provider's VM allocation policy. Second, our network statistics are aggregated over all other tenants, so it is difficult for a tenant to infer from them the network behavior of any other individual tenant. Finally, network topology among a tenant's VMs is already accessible even in today's black-box model via user probing approaches, e.g., as presented in PLink [57] and Choreo [49]. NetHint does provide easier access to this information, but we believe this does not increase the security risks. Note that NetHint does not fully eliminate these issues, and we discuss them in §9.

4.2 Timely NetHint with Low Cost

User query overhead The virtual topology is presented as a set of links (each with a Link ID). Each virtual link has its associated B_t. The temporal utilization information for each link includes a tuple of three fields (Link ID, n, B_r). Each field occupies 8 bytes. As such, the amount of data returned by a query is small. Consider a cloud tenant that has rented 100 VMs allocated across 10 racks. As upstream and downstream virtual links are considered separately, the number of virtual links equals twice the sum of the number of VMs and the number of racks the tenant occupies. The amount of query information thus has $(100+10)\times 2\times 3\times 8 = 5280$ Bytes.

There is no value or incentive for a tenant to query at a higher frequency than the information update period of NetHint (100 ms by default). Tenant VMs communicate with a NetHint server through TCP connections with rate limits that prevent queries more frequent than once per 50 ms.

Collection overhead We design a two-layer host-driven aggregation approach to collect timely hint information with low cost. Recall that we select one machine in each rack to run a NetHint server process. Each machine collects flow-level network characteristics from its operating system, and sends them to its rack-local NetHint server periodically. The information each machine has to send to the local NetHint server is a virtual link ID plus one (n, B_r) for each virtual machine to ToR link and another (n, B_r) containing only the traffic transmitting across the rack, for adding its contribution to the ToR uplink's (n, B_r). Each field is 8 bytes, so the total data size per virtual link is $(1+2\times 2)\times 8 = 40$ bytes. It is necessary to consider the upstream and downstream bandwidth independently, so each virtual machine or ToR has two associated virtual links. For example, assuming a physical machine has 10 VMs, it sends $40\times 2\times 10 = 800$ bytes of data to the NetHint server in each period. We set the information update period to 100 ms by default. Thus, the total aggregated information for one NetHint server is two (n, B_r) for every VM-to-ToR virtual link and the ToR uplink in the virtual topology. The NetHint servers then use all-to-all communication to exchange their aggregated information.

Suppose a data center has 1000 racks, and every rack has 20 machines. In each information update period, a local NetHint server gathers 16 KB information (800 bytes $\times$ 20 machines). With a 100 ms update period, the total amount of cross-rack traffic introduced by the all-to-all information exchange is $16\,\text{MB}/100\,\text{ms} = 1.3$ Gbps per rack. Let's assume each rack has outgoing bandwidth of 500 Gbps. Then the bandwidth overhead of NetHint is 0.26%.

Failure detection and recovery NetHint is a best-effort service, and applications should be prepared to function without hints, e.g., if their rack-local NetHint servers become unavailable due to failures such as link failure and server crashing. In this case, applications just revert the transfer schedule to a default one assuming no known network characteristics until a new NetHint server is available in the rack.

5 Adapting Transfer Schedules with NetHint

We find that most data-intensive applications can be categorized into two classes, based on how they can adapt to network characteristics. For each application class, we show that adapting transfer schedules corresponds to an optimization problem. Our goal here is not to present the optimal algorithm to solve the scheduling problems. Rather, our goal is to show that a broad set of distributed applications can benefit from NetHint using simple scheduling algorithms.

5.1 Optimizing Collective Communication

Many data-intensive applications run a high-level collective communication primitive (e.g., broadcast, allreduce) among a set of processes. Any such operation can be accomplished flexibly via a large set of possible *overlay topologies* among all the processes. For example, a broadcast can be performed with different broadcast trees connecting all the receivers, and an allreduce may employ different allreduce topologies (e.g., tree-allreduce or ring-allreduce). For all these communication primitives, the choice of overlay topologies affects only the efficiency (i.e., finish time) but not the correctness. Many popular ML applications belong to this category:

- **Data-parallel deep learning:** each server holds a replica of the model and calculates gradients locally. Servers use allreduce to synchronize gradients in each training iteration.
- **Reinforcement learning:** the trainer process in reinforcement learning repeatedly broadcasts the model (i.e., policy) to a dynamic set of agents.
- **Serving ensemble models:** multiple servers run DNN models simultaneously to predict the label on the same input data, and then use voting to decide the final output. For every input data batch, the front-end server broadcasts it to a set of servers holding different DNNs.

Moreover, as the object of collective communication is usually a vector of numbers, we can partition the object and apply different overlay topologies on each partition. For example, a broadcast can be accomplished via multiple broadcast trees, with each broadcast tree transferring a different (weighted) portion of the broadcast object. Similarly, an allreduce can be performed via a weighed combination of different allreduce topologies. The transfer schedule thus depends on both the choices of overlay topologies and their corresponding weights.

With NetHint, the tenant can estimate the bandwidth B_e^l available on each link l based on Equation 1 and Equation 2. For a transfer schedule s, denote the volume it transfers on each link l as d_s^l. The corresponding latency of the schedule can be estimated as $\max_l(d_s^l/B_e^l)$. Thus, we have:

Problem statement: *Given the virtual topology $\mathcal{T}$ and the estimated bandwidth on each virtual link l, find a transfer schedule that minimizes the latency* $\max_l(d_s^l/B_e^l)$.

To solve the above problem, one major challenge is that the number of candidate transfer schedules can be huge. For example, there can be $O(n^{(n-2)})$ possible broadcast trees to broadcast a message to n processes [79]. One possible solution is to use tree packing algorithms [13, 25, 79]. However, since the goal here is to show the usefulness of NetHint information rather than to find the optimal algorithm, we design simple heuristics to solve the problem. We first sample a random set of overlay topologies (broadcast and allreduce trees) which cross each rack only once. We then use linear programming to find the best weight assignment among these trees, so that the transfer schedule minimizes the latency $\max_l(d_s^l/B_e^l)$.

5.2 Optimizing Task Placement

Many distributed applications execute based on a task graph describing the tasks and their dependencies. The task graph can be static (i.e., task graph is known before the workload runs) [19, 85] or dynamic (i.e, tasks arrive as the workload runs) [61]. Since different tasks may send and receive different amounts of data, the placement of tasks onto VMs determines the transfer schedule among the VMs. Applications in data analytics frameworks and task-based distributed systems therefore can benefit from network-aware task placement:

- **Data analytics frameworks** [32, 85]: data analytics workloads contain network-intensive shuffle phases between different job stages. One shuffle phase creates an all-to-all communication between a set of sender tasks and receiver tasks, so task placement controls the shuffle performance.

Notations & Descriptions	
T_b	Average changing period of the background network condition
T_u	Duration of a transfer schedule being used
T_a	Latency to adapt (collecting information and computing a schedule)
T_s	Staleness of the hint
p	A threshold defined by the ratio between total adapting latency and JCT

Table 2: Important factors related to the impact of staleness.

- **Task-based distributed systems** [38, 61] are increasingly popular in industry. In these applications, the task graph is dynamic and generated at runtime. Tasks launch after fetching input objects from upstream tasks. As such, efficient task placement can minimize the task launch latency reducing the object fetch time.

Problem formulation For both applications, we can formulate the task placement as a classical network embedding problem. Denote the set of tasks as $\mathbb{T}$ and the set of VMs as $\mathbb{V}$. Compared with the problem statement in §5.1, which selects an efficient data transfer schedule, here we need to find an embedding $\mathcal{E}:\mathbb{T}\mapsto\mathbb{V}$ given the transfer schedule among all tasks. The algorithm inputs and optimization goals are the same as the problem statement in §5.1, except that the latency is calculated as $\max_l(d_e^l/B_e^l)$. d_e^l is the transfer volume on link l introduced by embedding $\mathcal{E}$.

We make minor modifications to the greedy heuristics proposed in Hedera [1] to solve the embedding problem. We first sort all tasks in $\mathbb{T}$ based on the amount of data they receive in decreasing order (no need if $|\mathbb{T}| = 1$). We then place tasks one by one following this order. When placing a task to $\mathbb{V}$, we optimize greedily for the objectives described in the problem statement. Before processing the next task, we update the cross rack traffic and d_e^l based on the placement.

6 Flexible Adaptation for Stale Information

Staleness of NetHint information The staleness of NetHint information during job execution is affected by the following two factors (notations listed in Table 2). First, an application controller can have a non-negligible latency to collect hints and compute the transfer schedules based on the hints, which makes the hints stale when being applied. We denote the adaptation latency as T_a.

Second, applications can adapt to hints periodically. For each adaptation period, the schedule calculated based on the previous hint will be used for the entire duration T_u. Note that for recursive jobs (e.g., model serving), recomputing the schedule for every iteration introduces too much latency. To this end, we fetch hints and recompute the schedule every k iterations, so that the latency to compute transfer schedule is within a portion p (e.g., 10% by default) of the job execution time. Moreover, for jobs that adapt the task placement based on hints (e.g., MapReduce), the adaptation period T_u equals job completion time, as the task placement usually cannot be changed during job execution.

Taken together, the staleness of NetHint information is quantified as $T_s = T_a + T_u$, which is the combination of both above factors. T_a is the total latency of four steps. The first three steps are to collect hints: sending host network characteristics to NetHint service, NetHint service exchanges rack-level network characteristics, and applications querying the NetHint service. The maximum latency for these three steps combined is 300 ms (100 ms per step due to NetHint frequency), so we use 150 ms as the estimate for the average case latency. The last step is to compute the transfer schedule, and it is application-specific (Figure 8). In our evaluation, a deep learning job of 64 workers requires 10 ms to compute its transfer schedule. We thus set $T_a = 150+10 = 160$ ms. We set $T_u = 100$ ms to keep the compute overhead to be less than 10% of the total running time.

Impact of the stale information The impact of stale information depends on the relative relationship between (1) the staleness of the information; and (2) the stability of the underlying network condition. Assume the background network condition changes every T_b time in average. A hint with staleness T_s much less than T_b can still be helpful since the network condition is likely to be similar with the condition T_s time ago. In contrast, a hint with staleness T_s much larger than T_b will be misleading, since the current network condition may be very different from the condition T_s time ago. In this case, adaptation with misleading hints can negatively affect the application performance (Figure 12d).

Flexible adaptation based on application and network condition. There are two takeaways from the above analysis. First, stale information should not be used when it is misleading. Regarding this, one approach is to simply ignore the provided hints and run applications as we run them today. However, as we show in motivating examples (e.g., Figure 1c and Figure 4c), the link-layer network topology alone can be useful for some types of applications to reduce the amount of cross-rack traffic. Compared with the bandwidth information, topology information is more stable and not affected by network dynamics.

Therefore, we propose NetHint-TO, a class of scheduling algorithms that use only the stable topology information from NetHint. For example, with NetHint-TO, we create a ring that crosses each rack only once for ring-allreduce and a chain that crosses each rack only once for tree-broadcast.

The second takeaway is that there is no one-size-fits-all solution. Each application should have two scheduling algorithms, one uses bandwidth information (in §5) and another one uses stable topology information only (NetHint-TO). We design a policy to choose between these two algorithms based on both the application and the network conditions (i.e., T_b, T_u, T_a). More specifically, when $T_s < T_b$, applications use the scheduling algorithm in §5 to calculate the optimal schedule based on both bandwidth and topology information. When $T_s \geq T_b$, applications adopt NetHint-TO to minimize the impact of stale information.

7 Implementation

We implement NetHint using 4600 lines of Rust code. 2300 additional lines of code are in NetHint server to provide NetHint to cloud tenants. The algorithms for applications to adapt transfer schedules (i.e., MapReduce, allreduce, and broadcast) are implemented using 149, 216, and 144 lines of code. We use `lpsolve` [56] for solving linear programs.

To compute the hints in our testbed, we take an endhost-based approach. We hook an eBPF program into the OS kernel. The eBPF program counts the total number of bytes going within the rack and outside the rack. A userspace program polls the counters from the eBPF program every 10 ms and maintains a moving average of the number of existing shared objects (i.e., flows, in a per-flow fairness model). The userspace program sends the number of shared objects and traffic data to the NetHint server every 100 ms. In a deployment environment where SmartNICs is available, we can also program the SmartNICs to implement this logic.

NetHint server binds to a TCP port, where VMs connect to to fetch hints. NetHint server uses a single thread to respond to NetHint queries. A single thread is enough for our design because queries are not frequent.

For an application to use NetHint, we need to modify the application. For traditional collective communication, the transfer schedule is static and decided before runtime. Recent collective communication designs have shown that transfer schedules can be dynamically decided at runtime [93]. NetHint can help these dynamic collective communication designs to decide on an efficient transfer schedule based on network characteristics. These dynamic collective communication designs can query and adapt transfer schedule every k iterations before issuing data transfer operation. For task placement, the global scheduler of a distributed system (e.g., master in MapReduce [19]) queries the NetHint server and uses both the task information and the NetHint information to decide task placement. For our evaluation purpose, we build a dynamic scheduler for collective communication and a task scheduler for MapReduce tasks according to the descriptions above.

8 Evaluation

8.1 Setup and Workloads

We evaluate NetHint using an on-premise testbed and large-scale simulations. Our setting is that hosts ensure work-conserving bandwidth guarantee for VMs and the network ensures per-flow fairness. We compare NetHint with the scenarios where cloud tenants (1) do not consider network characteristics and (2) probe the network to reverse-engineer the network characteristics and then adapt transfer schedules. For user probing, we assume network information is always correctly reverse engineered. We assume the probing strategy is the following: For a tenant that owns n hosts, user probing runs in $n/2$ rounds, where each round's latency is either the latency to send 10000 packets or 1 second, whichever is smaller, to measure throughput and latency between $n/2$ pairs of hosts. [2] Similar to NetHint, user probing adopts the same strategy to periodically update the transfer schedule, but with a lower frequency due to its higher overheads. We calculate user probing's frequency using the same method described in the second paragraph of §6.

We use a mix of two types of background traffic to simulate skewed and long-tailed traffic in data centers [3, 12, 70, 89]. One slow-moving background traffic occupies 0-50% bandwidth of the link capacity on each link in a Zipfian distribution. The slow-moving background traffic occupies 10% bandwidth in total and changes every 10 seconds. The other is a fast-moving background traffic which is on all links and occupies 0-10% bandwidth of the link capacity in a uniform random fashion. The fast changing background traffic changes every 10 ms. We use the following workloads. We run each experiment 5 times and report the average speedup for each job. To quantify the overall speedup, we also measure the arithmetic average of speedups across jobs.

Distributed data-parallel deep learning. We test the allreduce completion time. The job sizes are either 16 or 32 (in terms of number of nodes) with equal probability. For each allreduce job, we set the buffer size to be 100 MB ($\approx$ the size of ResNet-50). We run 100 jobs and assume jobs arrive as a Poisson process. We choose Poisson lambda $=24$ seconds, so that the average network utilization approximates to 12%.

Serving an ensemble of ML models. We test the broadcast completion time. We use the same job size distribution described in Hoplite [93]. We run 100 jobs and assume jobs arrive as a Poisson process. We choose Poisson lambda $=8$ seconds, so that the average network utilization approximates to 12%.

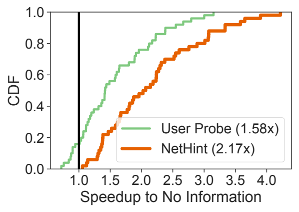
MapReduce. We test the latency of the data shuffling phase of MapReduce. We use Facebook's MapReduce trace [17], which contains 500 MapReduce jobs and their arrival time. We assume the traffic is divided evenly from a reducer to the mappers.

8.2 NetHint in Testbed Experiments

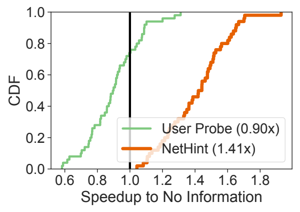
We build a 6-server testbed. Each server has a 100 Gbps Mellanox ConnectX-5 NIC and two Intel 10-core Xeon Gold 5215 CPUs (2.5 GHz). These machines are connected via an emulated 40 Gbps 2-stage FatTree network using a single 100 Gbps Mellanox SN2100 switch through self-wiring. 3 machines are in one rack, and the rest 3 machines are in the other rack. The oversubscription ratio on our network is 3. Each machine runs 4 VMs where each VM is guaranteed 10 Gbps through fair-queuing on the NICs.

Overheads. We already provide analysis of bandwidth overheads in §4.2. Now the remaining question is how much overhead NetHint incurs in terms of latency and CPU cycles.

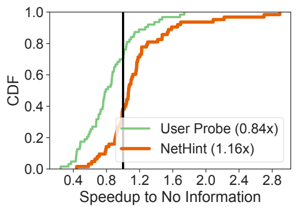
[2] We believe this is a best-case scenario for existing user probing techniques. Plink [57] sends 10000 packets per VM-pair to reverse engineer link-layer topologies. Choreo [49] uses a 3-step strategy to pinpoint congested links and its first step is measure pair-wise bandwidth. It takes 3 minutes to reverse engineer the network conditions for 10 VMs (90 VM-pairs).



(a) Distributed deep learning



(b) Ensemble model serving



(c) MapReduce

Figure 7: **Testbed results:** NetHint's speedup on testbed for allreduce in data-parallel distributed training, broadcast in ensemble ML model serving, and mapreduce shuffle compared with user probing and not using network information. Numbers in the legend shows the average of speedups compared with running applications without network information.

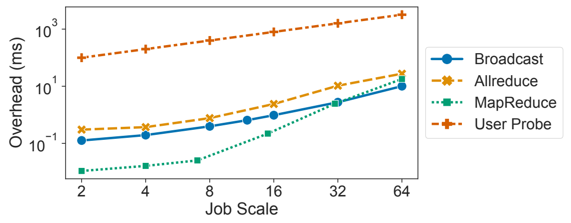


Figure 8: **Testbed results:** Latency to compute transfer schedules.

# Racks	CPU Util. (%)	Memory (MB)	Latency (ms)
6	0.06	4.53	10.60
24	0.14	5.90	10.73
96	0.41	19.28	11.91
240	0.66	78.16	13.73

Table 3: **Testbed results:** The system overhead of a NetHint server in CPU utilization, memory, and information collection latency.

Collecting statistics from eBPF program is instant, and the polling period for flow statistics is 10 ms.

To measure the overheads in large deployment, we use each CPU core in our testbed to emulate a rack by instantiating a NetHint server per-core. We use `pidstat` to measure the CPU cycles and memory footprint on NetHint server. Table 3 shows the result. When the number of racks scale up to 240 racks, the CPU time spent on NetHint servers is negligible, i.e., less than 0.66%. The memory footprint on each NetHint server is small (less than 80 MB) and scales with the number of racks mainly due to the increase in the hint size. The latency to collect network information is less than 14 ms.

We implement the algorithms described in §5. We test the computation latency of running each algorithm at different scales (number of workers). Figure 8 presents the results. The latency to make a scheduling decision remains low, ranging from 10 us to 30 ms. Compared with the computation latency, the extra latency introduced by user probing is much higher, ranging from 100 ms to 3 seconds. The round-trip latency to fetch hints takes 100 us because it is rack-local.

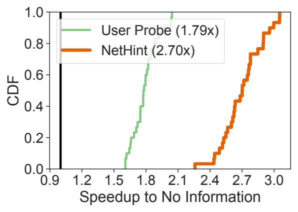
Results. NetHint improves application performance. Figure 7 shows the normalized speedup to running applications without network information. Using user probing speeds up the communication by 1.6x for distributed data-parallel deep learning and slows down the communication by 1.1x and 1.2x for serving an ensemble of ML models, and MapReduce shuffle, respectively. NetHint speeds up communication of these workloads by 2.2x, 1.4x, and 1.2x, substantially outperforming user probing. NetHint can outperform user probing because collecting hints is more lightweight than each application individually probing the network characteristics. User probing hurts many ensemble model serving and MapReduce jobs because of the probing overheads. In addition, we notice that a small portion of jobs in Figure 7c are penalized. On our testbed, the job log shows that there are on average 2.8 jobs sharing the rack bandwidth. One job arrival or departure changes the network condition for all the other jobs on the rack. However, the task placement decision cannot be changed during job execution, and thus the initial placement can be imperfect. In contrast, deep learning and model serving workloads in Figure 7 do not severely suffer from this problem, as they can timely modify the transfer schedule for each iteration based on the latest NetHint information.

8.3 NetHint in Simulations

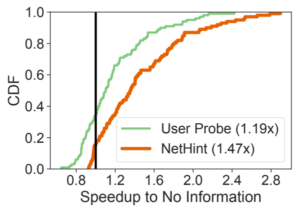
We use simulations to evaluate NetHint in large-scale deployments and in various operating environments. Our simulator is written in 5000 lines of Rust. The simulation is at flow level, and throughput of each flow is the result of solving a max-min fairness formula based on traffic demand. We simulate a CPU cluster and a GPU cluster individually. Both the CPU and GPU clusters have 150 racks. In the GPU cluster network, each rack has 6 machines with 100 Gbps NIC, and each rack has total upstream bandwidth of 200 Gbps. In the CPU cluster network, each rack has 18 machines with 100 Gbps NIC and the total upstream bandwidth is 600 Gbps. The oversubscription ratios are both 3. In the CPU cluster, each machine has 4 VMs. In the GPU cluster, each machine only has 1 VM. All VMs have bandwidth guarantee of 25 Gbps.

Results. Figure 9 shows the NetHint's speedup of the three workloads in our simulations. In summary, the trend of the simulation results matches what we have observed on the testbed. NetHint speeds up communication by 2.7x, 1.5x, and 1.2x, respectively. On allreduce, the speedup is higher than that on the testbed because the number of hosts involved in a job is larger than that on the testbed, and thus the amount of cross-rack traffic is also larger, giving NetHint more room to optimize transfer schedules.

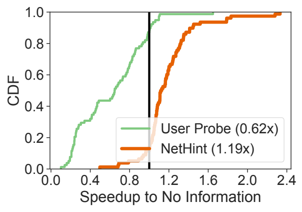
User probing incurs substantial overheads in both traffic and latency. Figure 10 shows the overheads of using NetHint and



(a) Distributed deep learning

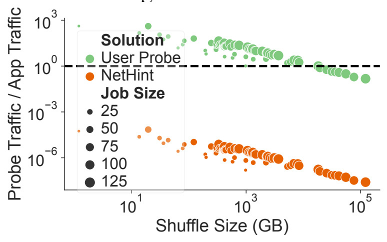


(b) Ensemble model serving

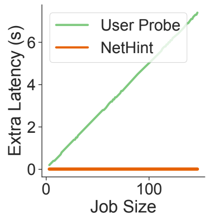


(c) MapReduce

Figure 9: **Simulation results:** Comparing NetHint with dynamic user probe in the default background traffic setting.

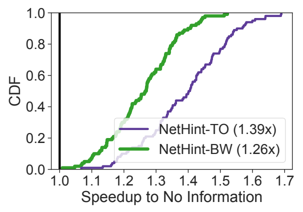


(a) Probing traffic over application traffic

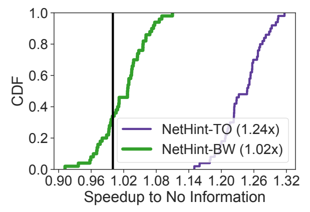


(b) Extra time

Figure 10: **Simulation results [MapReduce]:** Extra overhead for MapReduce jobs comparing NetHint and user probing.



(a) Coarse-grained workloads



(b) Non-negligible overhead to compute a transfer schedule

Figure 11: **Simulation results [Model serving]:** Using topology information alone can outperform using bandwidth information.

user probing in MapReduce. The amount of overhead depends on both MapReduce shuffle size and job size. Figure 10a shows the extra traffic introduced by NetHint and user probing over application traffic. NetHint only adds less than 0.1% extra traffic. User probing, in contrast, adds 15% to 420% extra traffic, and 90% of jobs double their traffic. This is because user probing needs to generate probe traffic, and each application has to probe independently. For large shuffle sizes, the probing traffic is less of a concern because it constitutes a smaller fraction of the total traffic. Figure 10b shows the extra latency due to probing and fetching hints for MapReduce jobs of various sizes. NetHint only adds a constant RTT-level extra latency which is negligible. User probing has a large latency overhead, which is linear in job size. This is expected because user probing needs to run for $n/2$ rounds, where n is the job size. There are a set of MapReduce jobs that are penalized substantially by user probing (as shown in Figure 9c). These are MapReduce jobs with large job sizes but with small shuffle sizes.

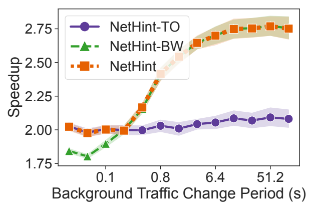
When should NetHint use topology information only? As we have described in §6, there are two situations we prefer letting NetHint use topology information only: (1) workload granularity is large, and (2) overhead of computing a transfer schedule is non-negligible. To demonstrate these situations, we set the slow-moving background traffic change frequency to every 0.2 seconds. Other environment settings remain the same as those in previous simulations.

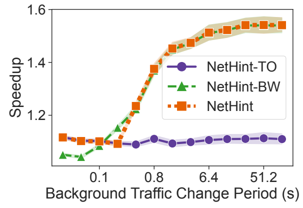
To show the case when background traffic changes faster than job completion time, we run 100 broadcast jobs with the model sizes increased to 1 GB. We let NetHint recompute a new broadcast strategy every iteration (but we still guarantee that the computational overhead is under a certain threshold $p = 10\%$). We use NetHint-TO to denote using only topology information when calculating the transfer schedule. We use NetHint-BW to denote using bandwidth information when calculating the transfer schedule. Figure 11a shows that NetHint-TO and NetHint-BW speed up the communication by 1.4x and 1.3x. NetHint-BW is slightly slower than NetHint-TO. Applying a bandwidth-aware algorithm does not bring benefit compared with using topology information only because the background traffic changes even within a single broadcast. Instead, it can slow down the job due to the additional overhead to compute data transfer schedules.

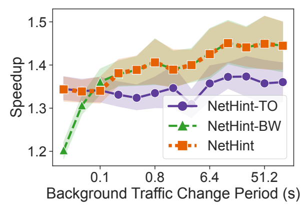
To demonstrate an extreme example for the computational overhead, we run 100 broadcasts of 64 workers with data size set to 12 MB, and we double the bandwidth capacity of ToR switch. Figure 11b shows that NetHint-TO and NetHint-BW speed up by 1.2x and 1.0x compared with no information. NetHint-BW cannot improve because the computation latency using LP is large in contrast to the broadcast latency on such a small data size. It has to adapt its traffic less frequently ($\approx 0.2s$) to ensure the compute overhead is within 10% of the total job completion time. Without being affected by inaccurate hints, NetHint-TO aims to minimize the cross-rack traffic, thus achieving better performance.

Figure 12 shows which adaptation method NetHint choose under different background traffic change periods and oversubscription ratios. The result demonstrates that NetHint chooses the best of NetHint-TO and NetHint-BW for all the three applications we use and also for both oversubscription ratio of 3 and 1.5.

Inaccurate bandwidth estimation. The bandwidth estimations in Equation 1 and Equation 2 is based on approximations, as the accurate estimation requires knowing the traffic demand for each tenant. One question to ask is whether NetHint's design fundamentally relies on the accuracy of bandwidth estimation. To answer this question, we intentionally add noise to the input of NetHint. Having additional noise of x% means the link utilization provided to NetHint is between 100-x%


(a) Deep Learning (Oversub=3)


(b) Deep Learning (Oversub=1.5)


(c) Model Serving (Oversub=3)

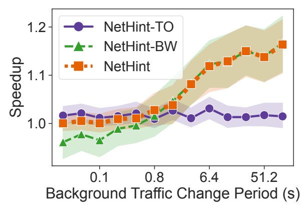

(d) MapReduce (Oversub=3)

Figure 12: **Simulation results:** Average speedup to background traffic change period under two different topology settings. The shaded area represents 95% confidence interval.

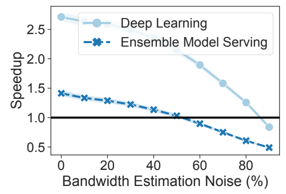


Figure 13: **Simulation results:** NetHint's speedup to not using network information when we add noise to the input of NetHint.

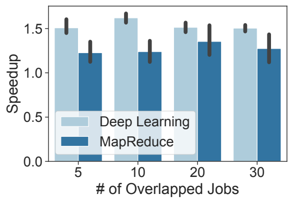
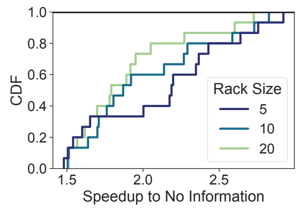


Figure 14: **Simulation results:** NetHint's performance when varying the number of overlapped jobs.


(a) Number of machines per rack

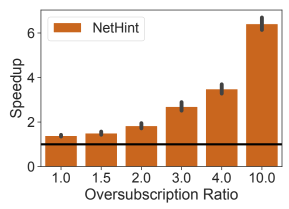

(b) Oversubscription ratios

Figure 15: **Simulation results [Distributed deep learning]:** NetHint's speedup to not using network information when we evaluate under different deployment environments.

and 100+x% of the actual utilization. We then evaluate the speedup of allreduce and broadcast jobs. Figure 13 shows the result. NetHint's speed up degrades gracefully. NetHint can still outperform not using network information when there is up to at most 50% noise.

Performance stability. To evaluate if NetHint's performance remains stable when the number of NetHint users is large, we increase the number of overlapped jobs. For deep learning, we enlarge the rack size to allow more jobs to share a ToR link and start all the jobs at the beginning. For MapReduce, we scale up the job arrival rate to create more overlapping among jobs. Figure 14 shows that NetHint can constantly achieve performance gain over not using network information.

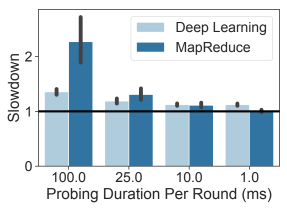
Sensitivity to network configurations. We evaluate NetHint's speedup under different network configurations in terms of the number of machines per rack and oversubscription ratios. We vary the number of machines per rack while keeping the oversubsription the same at 3. Figure 15a shows that NetHint can reduce the communication latency consistently for different rack sizes. We then vary the oversubscription ratio. Figure 15b shows that NetHint's improvement compared with not using network information increases as oversubscription ratio increases. This is because, when oversubscription ratio is high, the cross-rack communication is more likely to become the bottleneck. NetHint can mitigate this bottleneck by reducing the total amount of cross-rack traffic.

Performance gain over perfect user probing. In our evaluation, for n hosts, user probing is performed in $n/2$ rounds. In each round, it measures the bidirectional bandwidth and latency between $n/2$ pairs of hosts in parallel for a certain duration (default to 100 ms). Moreover, we show some evidence that it can be difficult to design better user probing technique to achieve similar performance as NetHint. First, we demonstrate how low the user probing duration has to be in order to achieve similar performance as NetHint. For this, we artificially reduce the probing duration while ensuring probing is accurate in simulations. Figure 16a shows the result: even when probing duration is reduced to 1 ms, NetHint still has a small performance advantage over user probing. Second, we show that such a low probing duration (i.e., 1 ms) for accurate bandwidth estimation can be difficult due to data center microbursts. We simulate data center microbursts based on measurement results in Facebook data centers [89] and calculate whether probing for x ms is sufficient to predict the average bandwidth of 100 ms. Figure 16b shows that if we measure for less than 25 ms, there is a 50% probability that the estimation error is above 75%. This is because there are gaps between microbursts, when a busy link is temporarily idle. Probing for such a short amount of time may not detect any traffic.

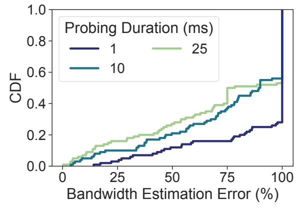
Does NetHint work for other fairness models? The rapid advancement in the programmability in emerging programmable switches makes it possible to implement other types of fairness models in the network [74, 83]. This trend makes it interesting to also understand NetHint's potential performance gains if we move to other fairness models in the future. We simulate the same allreduce jobs except that we modify our simulator for different fairness models. As shown in Figure 17, the trend of the simulation results matches what we have obtained in a per-flow based fairness setting.

9 Discussion

Herd behaviors. Tenants adapting transfer schedules with provided hints in a distributed way can potentially cause stability issues. For example, given the information of an under-utilized

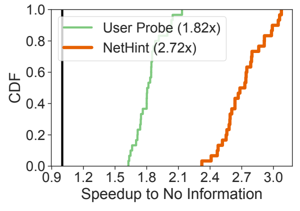


(a) Sensitivity to probing cost

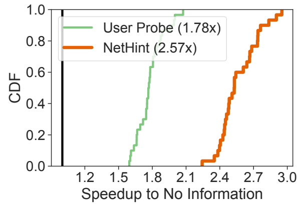


(b) Probing accuracy

Figure 16: **Simulation results:** The speedup of user probing to NetHint and the relative bandwidth estimation difference under different assumptions of probing durations. The black line in (a) represents NetHint.



(a) Per-tenant fairness



(b) Per-VM-pair fairness

Figure 17: **Simulation results [Distributed deep learning]:** Speedup for other fairness models.

link, many tenants may make identical choices to move traffic to this link, causing congestion. Such herd behavior causes load imbalance and performance oscillation in distributed load balancing problems [2,59,88]. We note that herd behavior is a common problem in some specific applications such as distributed load balancers. There are also standard techniques such as adding random jitters, and power of two choices to alleviated herd effect [59]. Whether and how NetHint should help specific applications avoid herd behavior is an interesting future direction. In the workload and setting of our evaluation, NetHint's speedup does not decrease when we increase the number of overlapped jobs (Figure 14). This infers that the performance of NetHint is not significantly affected by herd behavior.

Other competitive concerns for NetHint. NetHint exposes network utilization information to tenants. Network utilization can be a sensitive information. For example, one can infer whether a cloud has customers and whether a cloud provider does a decent job in network load balancing. NetHint makes it easy for a customer to compare network characteristics at different times. If a customer finds that the achievable bandwidth is reducing via NetHint, there may be a risk that the customer will switch to another cloud provider.

10 Related Work

Sharing network bandwidth. How to share network among many applications or cloud tenants is one of the oldest problems in computer networks. Today, network sharing is opaque to the application or cloud tenants. Within a single tenant, bandwidth sharing is through the fairness property of the underlying congestion control algorithms [24]. Across tenants, a cloud provider usually enforces strong isolation through static bandwidth allocation [68] or work-conserving bandwidth guarantee [9, 10, 50] on the NICs. It is difficult to enable either static bandwidth allocation or work-conserving bandwidth guarantee in the network because commodity switches have limited numbers of hardware queues. NetHint is complementary to these bandwidth sharing design: NetHint does not change any fairness property of the network. NetHint provides guidance for applications to use the network bandwidth better. A non-participating tenant can simply ignore the hint.

Collective communication and task placement based on network characteristics. Many related works optimize collective communication [20, 29, 45, 64, 79] or task placement [39, 49, 75, 80, 91] based on topology or bandwidth information. Similar considerations can also be applied inside OS for multi-core machines [11]. Most of these solutions assume the network topology or bandwidth information is already known. As such, NetHint can work in complementary with these solutions by providing them timely network information. Second, these works do not consider a multi-tenant environment. They assume workloads can be controlled by a logically centralized controller, while we assume each tenant's workload is controlled only by the tenant itself. Because tenants do not know other tenants' communication patterns, this knowledge needs to be provided either through cloud provider's support as proposed in this paper or using probing.

User probing. In addition to PLink and Choreo, many past works [5, 72, 81] also propose to measure network characteristics in wide-area networks to choose Internet route. NetHint is different in two aspects: (1) NetHint does not rely on active probe, and thus NetHint has low cost. NetHint simply reads counters directly from NICs or operating systems. (2) NetHint is for distributed applications that can adapt their transfer schedules rather than choosing routes in the network.

11 Conclusion

Today, the networking abstraction a cloud tenant has is a black box. This prevents a tenant's data-intensive applications from adapting the data transfer schedules to achieve high performance. We design and implement NetHint, a new paradigm for division of work between a cloud provider and its tenants. A cloud provider provides a hint, network characteristics (e.g., a virtual link-layer network topology, number of co-locating tenants, available bandwidth), directly to its tenants. Applications then adapt their transfer schedules based on these hints. We demonstrate the performance gain of NetHint on three use cases of NetHint including allreduce communication in distributed deep learning, broadcast in serving ensemble models, and scheduling tasks in MapReduce frameworks. Our evaluations show that NetHint improves the performance of these workloads by up to 2.7×, 1.5×, and 1.2×, respectively. Our source code is available at https://github.com/crazyboycjr/nethint.

Acknowledgement

We thank our shepherd John Wilkes and the anonymous NSDI reviewers for their insightful feedback. We thank Alvin R. Lebeck and Xiaowei Yang for their feedback on earlier versions of the paper. Our work is partially supported by an Amazon Research Award, a Meta Research Award, and an IBM Academic Award.

References

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[4] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, 2010.

[5] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *SOSP*, 2001.

[6] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*, 2014.

[7] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically Finding the Cause of Packet Drops. In *NSDI*, 2018.

[8] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *SIGCOMM*, 2020.

[9] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[10] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*, 2013.

[11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[12] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[13] Chandra Chekuri and Kent Quanrud. Near-linear time approximation schemes for some implicit fractional packing problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 801–820. SIAM, 2017.

[14] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*, 2014.

[15] Mosharaf Chowdhury and Ion Stoica. Efficient Coflow Scheduling Without Prior Knowledge. In *SIGCOMM*, 2015.

[16] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient Coflow Scheduling with Varys. In *SIGCOMM*, 2014.

[17] Coflow-Benchmark. `https://github.com/coflow/coflow-benchmark`, 2020.

[18] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized Congestion Control. In *SIGCOMM*, 2016.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[20] Mathijs Den Burger, Thilo Kielmann, and Henri E Bal. Balanced Multicasting: High-Throughput Communication for Grid Applications. In *SC*, 2005.

[21] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. On the Impact of Packet Spraying in Data Center Networks. In *INFOCOM*, 2013.

[22] Vanini Erico, Pan Rong, Alizadeh Mohammad, Taheri Parvin, and Edsall Tom. Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *NSDI*, 2017.

[23] Introducing data center fabric, the next-generation Facebook data center network. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network, 2020.

[24] S. Ben Fred, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts. Statistical Bandwidth Sharing: A Study of Congestion at Flow Level. In *SIGCOMM*, 2001.

[25] Harold N Gabow and KS Manu. Packing Algorithms for Arborescences (And Spanning Trees) In Capacitated Graphs. *Mathematical Programming*, 82(1):83–109, 1998.

[26] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. JUGGLER: A Practical Reordering Resilient Network Stack for Datacenters. In *EuroSys*, 2016.

[27] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *SIGCOMM*, 2016.

[28] Soudeh Ghorbani, Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Micro Load Balancing in Data Centers with DRILL. In *HotNets*, 2015.

[29] Y. Gong, B. He, and J. Zhong. Network Performance Aware MPI Collective Communication Operations in the Cloud. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):3079–3089, 2015.

[30] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, 2019.

[31] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*, 2015.

[32] Apache Hadoop. https://hadoop.apache.org/, 2020.

[33] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM*, 2017.

[34] Vipul Harsh, Sangeetha Abdu Jyothi, and P. Brighten Godfrey. Spineless Data Centers. In *HotNets*, 2020.

[35] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *MLSys*, 2019.

[36] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM*, 2015.

[37] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *SIGCOMM*, 2016.

[38] Hydro. https://github.com/hydro-project, 2020.

[39] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *SIGCOMM*, 2015.

[40] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *MLSys*, 2019.

[41] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC*, 2019.

[42] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[43] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[44] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.

[45] Nicholas T Karonis, Bronis R De Supinski, Ian Foster, William Gropp, Ewing Lusk, and John Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *IPDPS*, 2000.

[46] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *CoNEXT*, 2017.

[47] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *SOSR*, 2016.

[48] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. PicNIC: Predictable Virtualized NIC. In *SIGCOMM*, 2019.

[49] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. Choreo: Network-Aware Task Placement for Cloud Applications. In *IMC*, 2013.

[50] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. Netshare and Stochastic Netshare: Predictable Bandwidth Allocation for Data Centers. *SIGCOMM Comput. Commun. Rev.*, 2012.

[51] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-Driven Bandwidth Guarantees in Datacenters. In *SIGCOMM*, 2014.

[52] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit Switching Under the Radar with REACToR. In *NSDI*, 2014.

[53] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A Fault-Tolerant Engineered Network. In *NSDI*, 2013.

[54] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *SIGCOMM*, 2019.

[55] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*, 2016.

[56] Lpsolve. http://web.mit.edu/lpsolve_v5520/doc/index.htm, 2020.

[57] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. PLink: Discovering and Exploiting Locality for Accelerated Distributed Training on the Public Cloud. In *MLSys*, 2020.

[58] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. RotorNet: A Scalable, Low-Complexity, Optical Datacenter Network. In *SIGCOMM*, 2017.

[59] Michael Mitzenmacher. How Useful Is Old Information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, 2000.

[60] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *SIGCOMM*, 2018.

[61] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018.

[62] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making Network Stack Part of the Virtualized Infrastructure. In *USENIX ATC*, 2020.

[63] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless Datacenter Load-balancing with Beamer. In *NSDI*, 2018.

[64] Pitch Patarasuk and Xin Yuan. Bandwidth Efficient All-reduce Operation on Tree Topologies. In *IPDPS*, 2007.

[65] Y Peng, Y Zhu, Y Chen, Y Bao, B Yi, C Lan, C Wu, and C Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP*, 2019.

[66] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[67] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*, 2013.

[68] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *SIGCOMM*, 2007.

[69] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.

[70] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*, 2015.

[71] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter

Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. Technical report, KAUST, Feb 2019. http://hdl.handle.net/10754/631179.

[72] S. Savage, T. Anderson, Amit Aggarwal, David Becker, N. Cardwell, A. Collins, Eric Hoffman, John Snell, Amin Vahdat, G. Voelker, and J. Zahorjan. Detour: Informed Internet Routing and Transport. *IEEE Micro*, 19:50–59, 1999.

[73] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C. Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. Condor: Better Topologies Through Declarative Design. In *SIGCOMM*, 2015.

[74] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI*, 2018.

[75] Haiying Shen, Ankur Sarker, Lei Yu, and Feng Deng. Probabilistic Network-Aware Task Placement for MapReduce Scheduling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 241–250. IEEE, 2016.

[76] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[77] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *SOSP*, 2017.

[78] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is Big Data Performance Reproducible in Modern Cloud Networks? In *NSDI*, 2020.

[79] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *MLSys*, 2020.

[80] R. Wang, J. A. Wickboldt, R. P. Esteves, L. Shi, B. Jennings, and L. Z. Granville. Using Empirical Estimates of Effective Bandwidth in Network-Aware Placement of Virtual Machines in Datacenters. *IEEE Transactions on Network and Service Management*, 13(2):267–280, 2016.

[81] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Gener. Comput. Syst.*, 15(5–6):757–768, October 1999.

[82] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI*, 2018.

[83] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. Twenty Years After: Hierarchical Core-Stateless Fair Queueing. In *NSDI*, 2021.

[84] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

[85] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 2016.

[86] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *ATC*, 2017.

[87] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark. In *SIGCOMM*, 2016.

[88] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient Datacenter Load Balancing in the Wild. In *SIGCOMM*, 2017.

[89] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-Resolution Measurement of Data Center Microbursts. In *IMC*, 2017.

[90] Yangming Zhao, Kai Chen, Wei Bai, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. RAPIER: Integrating Routing and Scheduling for Coflow-aware Data Center Networks. In *INFOCOM*, 2015.

[91] Yangming Zhao, Chen Tian, Jingyuan Fan, Tong Guan, and Chunming Qiao. RPC: Joint Online Reducer Placement and Coflow Bandwidth Scheduling for Clusters. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018.

[92] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.

[93] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems. In *SIGCOMM*, 2021.