Saba: Rethinking Datacenter Network Allocation from Application's Perspective

M.R. Siavash Katebzadeh
University of Edinburgh
United Kingdom
m.r.katebzadeh@ed.ac.uk

Paolo Costa
Microsoft Research
United Kingdom
paolo.costa@microsoft.com

Boris Grot
University of Edinburgh
United Kingdom
boris.grot@ed.ac.uk

Abstract

Today's datacenter workloads increasingly comprise distributed data-intensive applications, including data analytics, graph processing, and machine-learning training. These applications are bandwidth-hungry and often congest the datacenter network, resulting in poor network performance, which hurts application completion time. Efforts made to address this problem generally aim to achieve max-min fairness at the flow or application level. We observe that splitting the bandwidth equally among workloads is sub-optimal for aggregate application-level performance because various workloads exhibit different sensitivity to network bandwidth: for some workloads, even a small reduction in the available bandwidth yields a significant increase in completion time; for others, the completion time is largely insensitive to the available bandwidth.

Building on this insight, we propose Saba, an application-aware bandwidth allocation framework that distributes network bandwidth based on application-level sensitivity. Saba combines ahead-of-time application profiling to determine bandwidth sensitivity with runtime bandwidth allocation using lightweight software support with no modifications to network hardware or protocols. Experiments with a 32-server hardware testbed show that Saba improves average completion time by 1.88× (and by 1.27× in a simulated 1,944-server cluster).

CCS Concepts: • **Networks** → **Network protocols**.

Keywords: Datacenter Networks, Bandwidth Allocation, Max-Min Fairness

EuroSys '23, May 8–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3587450

ACM Reference Format:
M.R. Siavash Katebzadeh, Paolo Costa, and Boris Grot. 2023. Saba: Rethinking Datacenter Network Allocation from Application's Perspective. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3587450

1 Introduction

Today's private datacenters host a diverse range of data-intensive applications, including machine learning training [7, 22, 38, 51, 60, 62], SQL queries [9, 30, 58], graph processing [21, 39, 41], and big-data analytics [47, 52]. Many of these applications are distributed and leverage parallel frameworks, such as Hadoop, Spark, Flink, and TensorFlow [1, 8, 17, 53, 61]. They typically adopt a bulk communication model with hundreds of connections transferring data between servers across multiple processing stages. This communication model creates a high load on the network, leading to congestion and queueing delays that affect applications' completion time.

To address this issue, over the last decade, there has been a slew of proposals in the literature on how to improve congestion control in datacenters and efficiently share network bandwidth among competing flows and applications. Some of these proposals focus on providing per-flow fairness [18, 20, 34] with various network-level objectives, such as minimizing per-packet latency [25] or flow completion time [5, 6, 44] using network-level properties, such as flow size or deadlines [5, 6, 13, 25, 44, 55]. Other works aim to provide isolation among tenants or applications when running in shared datacenters [11, 12, 23, 27, 32, 48, 57].

A common trait of all of these proposals is that they aim to achieve some variant of max-min fairness at the flow or application level. In this paper, we challenge this conventional wisdom and argue that max-min fairness is not the right metric to optimize for as different applications exhibit different degrees of sensitivity to the amount of network bandwidth, and, hence, splitting the bandwidth equally among them may lead to sub-optimal performance. Instead, we propose a new metric called *bandwidth sensitivity*, which captures the impact of the network bandwidth on the completion time for a specific application. Our analysis using a broad collection of workloads shows that different applications exhibit various degrees of sensitivity. For example, given a 56Gbps network link, reducing the link capacity to 25% leads to a

3.4× increase in the completion time of a Logistic Regression (LR) workload, while the completion time of a PageRank (PR) workload increases by a factor of 1.4×.

This observation is at the core of Saba, a novel application-aware bandwidth allocation scheme that distributes network bandwidth to applications proportionally to their bandwidth sensitivity. Saba computes the bandwidth sensitivity of applications in advance and leverages this information at runtime to derive the weights used in switches' priority queues to enforce the desired bandwidth allocations in a work-conserving fashion.

We put particular effort into the design of Saba to make it practical to use in existing datacenter deployments by limiting the amount of resources needed during profiling and the number of queues used in switches. Saba does not mandate any changes to deployed congestion-control protocols and it is fully compatible with existing switches and NICs while requiring only a lightweight shim layer (~350 LOCs) installed at the end hosts.

We evaluate Saba in a 32-server InfiniBand cluster across a broad set of workloads. We also simulate a Saba deployment with 1,944 servers to assess its performance at scale. Our experimental and simulation results show that Saba improves the average performance of the co-running applications sharing the network, compared to both existing and *ideal* implementations of per-flow max-min fairness.

The main contributions of this paper are the following:

- We show that enforcing per-flow max-min fairness as an application-agnostic bandwidth allocation scheme can lead to poor aggregate application performance when multiple applications share the network.
- We introduce the notion of bandwidth sensitivity as a guiding principle to allocate bandwidth among applications and show how this can be learned through profiling.
- We present Saba, our application-aware bandwidth allocation framework, which relies on bandwidth sensitivity for bandwidth allocation. We compare Saba against a baseline using InfiniBand congestion control and demonstrate that Saba can reduce the completion time for bandwidth-sensitive jobs by up to 3.94× while only marginally impacting a few of the bandwidth-insensitive jobs (1-5% slowdown), improving the average completion time by 1.88×.
- We show in simulations that similar benefits also hold at scale and against an ideal implementation of max-min fairness, obtaining up to 1.79× speedup (3% slowdown in the worst case) with an average improvement in completion time of 1.27×.

2 Motivation

Mainstream bandwidth allocation schemes adopted in datacenters aim to achieve max-min fairness by equally partitioning network bandwidth across flows. In this section, we demonstrate that this approach can potentially harm application performance (measured as job completion time) since the impact of reduced bandwidth varies from one application to another; hence, we argue that network resources should be split according to the impact of the network on applications.

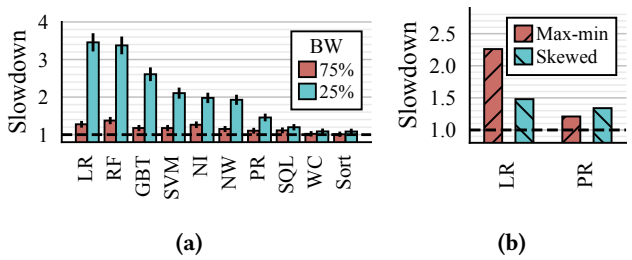


Figure 1. (a) Impact of available bandwidth on the performance of workloads (see Table 1). **(b)** Impact of bandwidth allocation scheme on the performance of two co-running workloads.

2.1 Sensitivity to bandwidth in applications

To assess the impact of network bandwidth on application performance, we profile different workloads in isolation and compute the completion time for different percentages of network bandwidth (25% and 75%) in an 8-server cluster connected with a 56Gbps switch (see §8.1 for details of the methodology). We measure the *slowdown* compared to the execution with unthrottled bandwidth.

Figure 1a shows the slowdown of various workloads under different percentages of available bandwidth. As the figure shows, the degree of slowdown varies across the workloads. For example, while LR suffers a 1.3× performance penalty when 75% of the bandwidth is available, the impact on performance for Sort and WordCount (WC) is negligible. With 25% of bandwidth, the slowdown of applications varies from 1.1× (Sort) to 3.4× (LR), with an average of 2.1×. This analysis indicates that applications are not equally sensitive to bandwidth degradation.

2.2 Using sensitivity in bandwidth allocation

Our analysis presents an opportunity to rethink the use of max-min fairness and, instead, allocate network bandwidth based on applications' sensitivity to it. The intuition is that by providing more bandwidth to the applications that are most sensitive, their completion time can be reduced without disadvantaging applications with low latency to network bandwidth. To validate this intuition, we run an experiment comparing the slowdown experienced by two workloads, LR and PR, when running together compared to the stand-alone

execution. We consider two different bandwidth allocation schemes:

1. Max-min: We use InfiniBand, which employs per-flow max-min fairness through end-to-end congestion management via Forward Explicit Congestion Notification [3, 54] and splits the bandwidth equally among competing flows [29]. Consequently, when both workloads simultaneously use the network, flows from each workload get 50% of the capacity of shared links.

2. Skewed: As illustrated in Figure 1a, LR and PR exhibit different degrees of sensitivity to bandwidth. The skewed allocation scheme leverages this information and provides more bandwidth to LR and less to PR. Specifically, the skewed scheme allocates 75% of bandwidth to LR and 25% to PR, i.e., the ratios shown in Figure 1a.

Figure 1b illustrates the slowdown of the two co-running workloads under the different allocation regimes. In the max-min configuration, LR and PR face 2.26× and 1.21× slowdown, respectively, compared to stand-alone execution. In the skewed configuration, in which more bandwidth is allocated to LR, the slowdown of LR decreases from 2.26× to 1.48× (78% improvement) while the slowdown of PR slightly increases from 1.21× to 1.34× (<13% degradation).

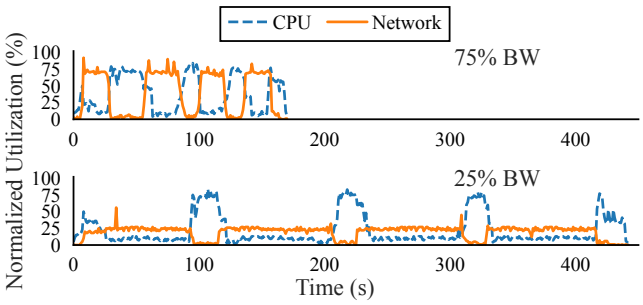
2.3 Why does the bandwidth sensitivity arise?

To answer this question, we assess the impact of bandwidth on the resource utilization of LR and PR. In this experiment, we run each application separately. Figure 2 shows the timeline of normalized resource utilization for both CPUs and network with 75% and 25% of total network bandwidth available to the application. In the figure, a low network utilization with high CPU utilization implies a *computation* phase. Likewise, a high network utilization with low CPU utilization shows that the workload is in a *communication* phase.

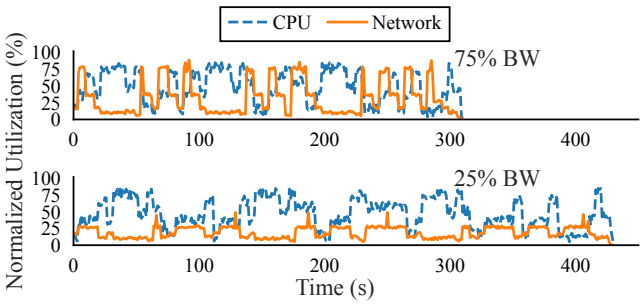
As the figure shows, the duration of the computation phases in LR remains relatively constant when the available bandwidth is decreased from 75% to 25%. Meanwhile, the duration of the communication phases increases as bandwidth is decreased from 75% to 25%. As a result, the completion time of LR increases by 2.59× (from 172s to 447s). In contrast, PR has more communication phases, but the workload is dominated by computation. Moreover, as Figure 2b shows, unlike LR, in PR, some of the data transmission is *overlapped* with computation (i.e., high network *and* CPU utilization). As a result, decreasing bandwidth has less impact on PR than LR. We observe that by decreasing the bandwidth from 75% to 25%, the completion time of PR increases by only a factor of 1.37× (from 310s to 427s).

2.4 Sensitivity-aware bandwidth allocation

The results above are important because they demonstrate that, for this kind of workloads, *i)* equally distributing the bandwidth like in traditional max-min fairness protocols



(a) LR



(b) PR

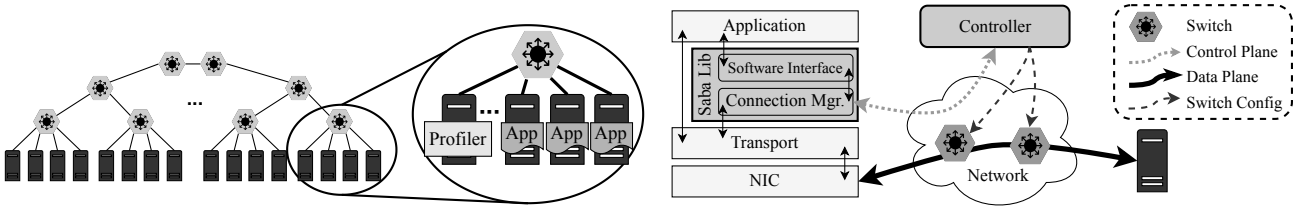
Figure 2. Impact of available bandwidth on resource utilization and completion time.

(e.g., TCP), or *ii)* focusing on reducing average *flow* completion time for individual flows (e.g., Homa [44]) or for coflows[1] (e.g., Sincronia [2]), does not necessarily result in shorter average *application* completion times. In fact, our experiments in Figure 1b show that by *unequally* distributing the bandwidth between LR and PR, and by *increasing* the flow completion time for the less sensitive application (PR), the average completion time of applications is significantly reduced. Therefore, in this paper, we argue that application sensitivity should be the primary factor driving network bandwidth allocation rather than traditional network-level, application-agnostic metrics.

Building a sensitivity-aware allocation scheme as described in §2.2, however, requires solving the following challenges, which we address in the following sections:

- Sensitivity Differentiation: An application-aware solution requires a robust approach to capture the application's sensitivity to network bandwidth.
- Dynamism: At the datacenter scale, a multitude of applications will share the network, with new applications arriving and others terminating or migrating over time. A bandwidth allocation mechanism must be able to handle such dynamism in a timely and resource-effective manner.

[1] collection of related flows



(a) The offline profiler deploys applications and profiles them.

(b) The controller and Saba library.

Figure 3. An overview of the main components of Saba.

- Practicality: To facilitate adoption and maximize generality, a bandwidth allocation scheme should not require changes to deployed hardware and/or network protocols.

3 Saba Overview

We introduce Saba, an application-aware bandwidth allocation scheme that aims to maximize performance across non-interactive applications that share the network in a datacenter. At the heart of Saba is a metric called *bandwidth sensitivity*, which reflects the effect of network bandwidth on application performance. We define bandwidth sensitivity for a given application as *the degree of performance degradation caused by a reduction in available bandwidth*. Saba's key idea is that providing more bandwidth to applications that are more sensitive to bandwidth can reduce their completion time, without compromising the performance of bandwidth-insensitive applications.

Saba consists of three main components: an offline profiler, a controller, and a library (Figure 3). The *offline profiler* determines the bandwidth sensitivity of applications by profiling them in advance (Figure 3a). The *controller* uses the bandwidth sensitivity information provided by the profiler to calculate the bandwidth share of applications in a way that minimizes the average slowdown across applications and orchestrates the network switches to enforce bandwidth. Saba expects compliant applications to be registered at launch. The Saba *library* provides a software interface for applications to communicate with the controller and conduct the registration (Figure 3b).

While Saba dynamically partitions network bandwidth for Saba-compliant applications, non-Saba-compliant applications (e.g., control or latency-critical services) can co-exist on the same network. To support this co-existence, datacenter operators can statically allocate a queue for non-Saba-compliant applications on switches and reserve a portion of the network bandwidth for them. This queue-based separation effectively isolates flows of non-Saba-compliant applications, preventing interference between them and Saba-compliant applications. Thus, it allows Saba-compliant applications to benefit from the dynamic bandwidth allocation without negatively impacting the performance of other applications running on the same network.

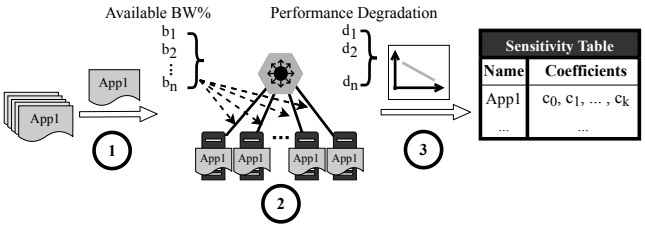


Figure 4. Details of the offline profiler.

The following sections provide details of the aforementioned components of Saba (*offline profiler*, *controller* and *library*), as well as details of our implementation.

4 Offline Profiler

Saba borrows the idea of using a-priori profiling of applications to make resource allocation decisions [4, 56, 59]. Saba's offline profiler performs ahead-of-time profiling on applications to measure their bandwidth sensitivity based on the performance degradation caused by limited network bandwidth. The profiler uses *application completion time* as the metric of performance. Completion time is commonly used to evaluate the performance of data-intensive applications [33] and can easily be determined at the system level by recording the start time and end time of an application.

4.1 Profiling Process

Figure 4 depicts the profiling process managed by the profiler. To profile a given application *App1*, the profiler first deploys the application on multiple nodes ①. In order to measure the performance degradation of the application caused by changes in available bandwidth, the profiler runs the application n times. In each run, the profiler limits the bandwidth of NICs of all nodes to a certain percentage of link capacity from $BW = \{b_1, b_2, \ldots, b_n\}$ and measures the completion time of the application ②. Next, for each measured completion time, the profiler determines the performance degradation by calculating the *slowdown*, i.e., the ratio of completion time under a given percentage of bandwidth to the completion time with unthrottled bandwidth. The output is $D = \{d_1, d_2, \ldots, d_n\}$, a

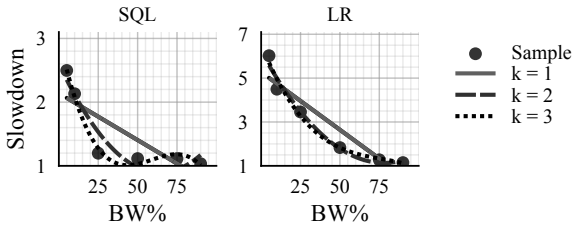


Figure 5. Sensitivity models of SQL and LR workloads with various degrees of polynomial (k).

set of performance degradation values, each of which represents the impact of the corresponding percentage of bandwidth on the performance of the application. The profiler uses $Samples = \{(b_1, d_1), (b_2, d_2), \ldots, (b_n, d_n)\}$ to generate a *polynomial regression model* and establish the relationship between the bandwidth and the slowdown of the application ③. We refer to this regression model as the *sensitivity model*. The sensitivity model of *App 1* can be represented as follows:

$$D_{App1}(b) = c_0 + c_1 b + c_2 b^2 + \cdots + c_k b^k = \sum_{i=0}^{k} c_i b^i \quad (1)$$

where k is the *degree of polynomial*. The profiler determines the value of the coefficients, $\hat{C} = \{c_0, \cdots, c_k\}$, by fitting the polynomial to the samples, and records the coefficients in the *sensitivity table*. Saba uses this table in its controller for bandwidth allocation (§5).

4.2 Accuracy of Sensitivity Model

In order to evaluate the goodness-of-fit and accuracy of a sensitivity model, we use R^2 (coefficient of determination) [37]. R^2 quantifies the fraction of the slowdown trend that the model is able to explain. $R^2 = 1$ implies that the model explains all the observed slowdowns in response to bandwidth. A model with $R^2 = 0$ does not explain any of the observed slowdown trend. The accuracy of the sensitivity model generated by the profiler depends on the degree of the polynomial, and on the differences between settings at runtime as compared to profile time. These settings include the size of the dataset and the number of nodes. We next discuss the effect of each of these parameters on R^2.

Degree of polynomial: The degree of polynomial determines the ability of the model to capture any non-linearity in the relationship between bandwidth and performance degradation of an application. We assess the degree of non-linearity for the studied workloads by profiling them as discussed in §4.1 (see the complete methodology in §8.1).

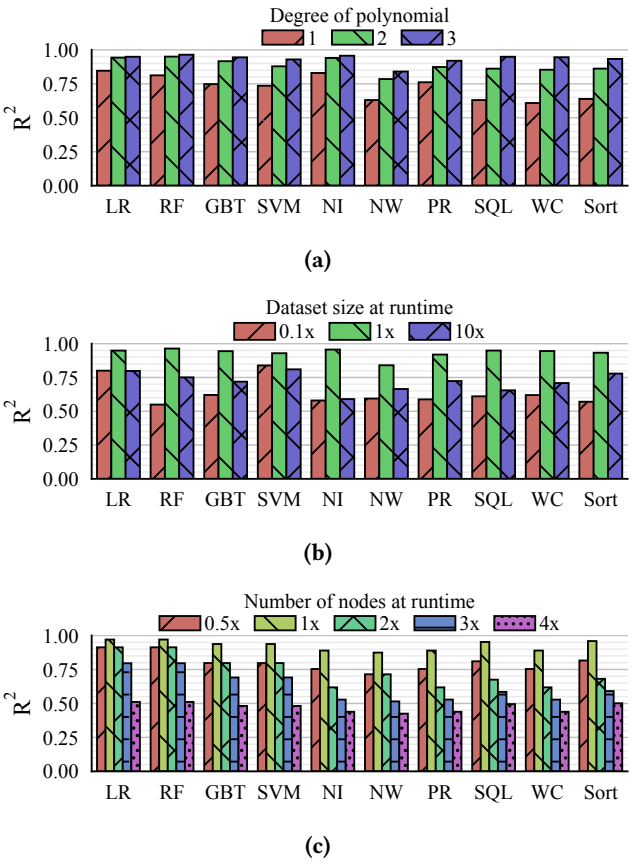


Figure 6. Impact of degree of polynomial (a), dataset size (b), and number of nodes (c) on the accuracy of sensitivity models.

Figure 5 plots the profiling samples for two studied workloads, SQL and LR, along with the sensitivity models for three different degrees of polynomial.

As Figure 5 shows, the non-linearity of the samples varies across the two workloads. For SQL, the non-linearity is high – as bandwidth is decreased from 100% to 25%, performance degrades by a mere 1.2×; however, as the bandwidth is further reduced to 10%, performance drops by 2.2×. In contrast, LR experiences a higher performance degradation throughout the bandwidth range but exhibits a more linear correlation between performance and bandwidth. Performance of LR degrades by 1.3×, 3.4×, and 4.5× as bandwidth is decreased to 75%, 25%, and 10%, respectively. The figure clearly shows that a first-degree polynomial model is unable to accommodate the data for SQL, and a third-degree polynomial is needed for a good fit; while a second-degree polynomial perfectly fits the data for LR.

We next study the accuracy of the sensitivity models based on different degrees of the polynomial. Figure 6a shows that as the degree of polynomials increases, R^2 of the sensitivity

models also increases. The sensitivity models of all workloads show accuracy above 0.60 using the first-degree polynomial ($k = 1$). We observe that some workloads enjoy a big jump in their accuracy by using higher degrees of polynomial in their sensitivity model; e.g., R^2 for the sensitivity model of SQL increases from 0.63 to 0.96 as we increase the degree of polynomial of the model from 1 to 3. For other workloads, lower degrees of polynomial provide models with high accuracy; e.g., the sensitivity model of LR generated with $k = 1$ attains an accuracy of 0.84. Using $k = 2$ increases the accuracy of LR's model to 0.94, while $k = 3$ provides only a small additional improvement to 0.95. We conclude that the degree of polynomial directly affects the accuracy of sensitivity models.

Application dataset size: In practice, it may be challenging to know the target dataset in the profiling phase. It may also be desirable to run profiling with a smaller dataset size than in deployment to reduce profiling time and cost. Thus, the size of the dataset at runtime may differ from that used by the profiler. To analyze how the accuracy of each model depends on the dataset size, we conduct a study to estimate the accuracy of the sensitivity model of each workload when the size of datasets used at runtime differs from the profiling datasets. In this study, all models are generated with the polynomial of degree three ($k = 3$).

Figure 6b shows the impact of the runtime dataset size on the accuracy of sensitivity models of the studied workloads in extreme cases in which the dataset size at runtime is ten times smaller (0.1x) or larger (10x) than the dataset used by the profiler (1x). We observe that while using datasets with sizes of 0.1x and 10x reduces the accuracy of the models, all models obtain R^2 above 0.55. SVM experiences the most negligible impact as the accuracy of its model reduces from 0.92 (dataset size 1x) to 0.83 (dataset size 0.1x) and 0.81 (dataset size 10x). Nutch Indexing (NI) experiences the highest impact as the accuracy of its sensitivity model decreases from 0.95 (dataset size 1x) to 0.57 (dataset size 0.1x) and 0.59 (dataset size 10x). We conclude that for the studied workloads, an order of magnitude difference in dataset size between profiling and runtime attains R^2 above 0.55, indicating that the sensitivity model retains good predictive power despite the change in dataset size.

Number of nodes: Similar to the dataset size, the number of nodes running a distributed application may not be the same as the number of nodes used by the profiler. For instance, limiting the number of nodes used in the profiling phase may be desirable to contain the costs of profiling. To study the impact of the number of nodes at runtime on the accuracy of sensitivity models, we compare the R^2 of models of the workloads across various numbers of nodes at runtime, ranging from 0.5x to 4x of the number of nodes used by the profiler (8 nodes). All models use $k = 3$.

As Figure 6c shows, the sensitivity models of all workloads maintain an accuracy above $R^2 = 0.50$ when the number of nodes at runtime ranges from 0.5x to 3x. The sensitivity model of NWeight (NW), delivers the lowest observed accuracy $R^2 = 0.51$ when tripling the number of nodes. When increasing the number of nodes at runtime to 4x compared to profiling, we observe that R^2 drops for most models to below 0.50; the exceptions are LR, Random Forest (RF), and Sort. We conclude that the number of nodes is a crucial factor governing the accuracy of the sensitivity models.

We quantify the performance of Saba with varying degrees of polynomial, dataset size, and the number of nodes in §8.2.

5 Controller

Saba relies on a controller to perform bandwidth allocation and orchestrate switches for bandwidth enforcement. To conduct the allocation and enforcement, the controller requires the following information: 1) which applications are Saba-compliant, and 2) the source and destination of each connection for each Saba-compliant application. The controller needs information about the source and destination of a given connection to determine the switches along their path[2]. Applications explicitly or transparently send the above information to the controller via a software interface provided by the Saba library (details are in §6). The controller collects the above information from applications and determines the bandwidth share of each application at runtime. To calculate the bandwidth share of applications, Saba uses the bandwidth sensitivity information in the sensitivity table provided by the profiler. The controller assigns the allocated bandwidth to applications and configures the switches to enforce the bandwidth shares.

5.1 Bandwidth Calculation and Assignment

Saba allocates bandwidth for applications that have registered themselves via the Saba library and are actively using the network. By tracking the active applications through the Saba library, the controller has global information about the paths of Saba-compliant flows passing through switches in the network (see §5.4 for details on scalability). The controller uses this information combined with the profiling result in the sensitivity table and determines the percentage of bandwidth to be allocated to the flows from each application at each switch output port in a way that *minimizes the total slowdown across applications.*

For a given set of applications $\hat{A} = \{a_1, a_2, \ldots, a_n\}$ sending flows to a given switch output port, weight w_i represents the percentage of bandwidth allocated to application a_i at that port. The goal is to find a set of weights to minimize the total slowdown across applications. To do so, the controller uses the sensitivity models generated by the profiler to predict the slowdown of each application. $\hat{D} = \{D_1, D_2, \ldots, D_n\}$ is the set of sensitivity models corresponding to $\hat{A}$, each of which

[2]If the underlying network layer supports multipathing, the controller determines switches along all paths between the source and destination.

generated via Eq 1. The controller calculates the weights $\hat{W} = \{w_1, \ldots, w_n\}$ for applications at the given switch output port as follows:

$$\hat{W} = \arg\min_{\vec{W}} \sum_{i=1}^{n} D_i(w_i)$$
$$\text{subject to} \sum_{i=1}^{n} w_i = C_{Saba} \tag{2}$$

where C_{Saba} is the percentage of link capacity that the operator reserves for Saba-compliant applications. The controller uses these weights to configure the given output port of the switch and enforce bandwidth.

5.2 Bandwidth Enforcement

Saba enforces the allocated bandwidth at network switches and leverages the available rate-limiting mechanism used in the transport layer. This approach decouples the bandwidth allocation and enforcement from congestion management, leading to a cleaner design. Thus, Saba does not need to calculate and limit the transmission rates at the endpoints and leaves the rate-limiting to the congestion-control protocol.

Bandwidth enforcement at switches works in Saba without any modification to existing switches, as long as the network layer supports the following requirements: 1) service differentiation through Priority Levels (PLs), and 2) per-port queues in switches. Service differentiation is required to differentiate flows coming from different applications. Per-port queues in switches are needed to enforce bandwidth by assigning flows with weights to queues. Moreover, Saba assumes that switches implement the Weighted Fair Queuing (WFQ) scheduling algorithm to schedule the packets inside the per-port queues in proportion to the weights of queues. WFQ is work-conserving, meaning that other applications may utilize the remaining bandwidth quota if one application does not use some or all of its share. Furthermore, WFQ is not subject to starvation, meaning that all flows progress and are eventually transmitted, which is an advantage for a bandwidth allocation scheme. Fortunately, modern switches used in datacenters support both service differentiation and per-port queues and implement variations of WFQ [10, 44].

In Saba, the controller assigns a PL to flows coming from each application with the help of the Saba library (§6), and maps each PL to a queue in the switch port. The controller configures each switch output port with a set of calculated weights $\hat{W} = \{w_1, \ldots, w_n\}$ determined via Eq (2) and assigns each weight to the corresponding queue, and each switch services flows inside queues using the WFQ scheduling algorithm. Note that this approach assumes that switches have an unlimited number of queues, thus using an idealized one-to-one mapping of applications to queues. However, existing switches have a limited number of queues [10, 44]. We next discuss how Saba addresses this issue.

5.3 Mapping Applications to Queues

While an ideal implementation of Saba should support a one-to-one mapping between applications and queues, modern network architectures support only a limited number of PLs and queues. The number of PLs is determined by the Quality of Service specification of network technology, whereas the number of queues in NICs and switches varies across different hardware implementations. For instance, InfiniBand and Ethernet support 16 and 8 PLs, respectively [10, 44], and a typical datacenter-grade (InfiniBand or Ethernet) switch supports 4-8 queues [6, 10, 44]. To overcome this limitation, Saba performs two layers of mapping to first translate applications to queues by mapping applications to PLs and secondly mapping PLs to queues.

5.3.1 Application-to-PL mapping. At the datacenter scale with hundreds to thousands of running applications, a one-to-one mapping between applications and PLs is infeasible due to the limited number of PLs. To address this limitation, Saba groups applications according to their bandwidth sensitivity using the K-means clustering algorithm [40]. The controller takes a set of registered applications and the coefficients of their sensitivity models as input, creating S groups from them, where S is the number of PLs. The centroid of each group represents the sensitivity of that group. Saba assigns each group a PL and uses the sensitivity of the groups in the PL-to-queue mapping.

5.3.2 PL-to-queue mapping. The controller needs to map PLs to queues to complete the application-to-queue translation. This task, however, is complicated by the fact that flows in a given PL may share different links along their paths, thus resulting in a different set of flows mapping to the PL in different switches. Consequently, the weight of a PL may vary across the switches. Additionally, the number of queues in different switches varies as the capability of switches is not necessarily the same. Thus, PL clustering must be performed individually at each switch output port to map PLs to queues.

To address this problem, Saba must maintain multiple PL clusters and PL-to-queue mappings and choose the appropriate mapping for each switch port at runtime. To this end, Saba introduces a hierarchical approach: (1) to cluster PLs, Saba uses a fast hierarchical clustering [45] scheme to preserve the information of all possible combinations of PL clustering hierarchically; (2) at runtime, Saba finds the best clustering from the hierarchy for each switch output port and uses it for bandwidth allocation. We next detail each of these tasks.

1. PL clustering: In the first level, the controller assigns each PL to a separate cluster. The controller generates a new cluster in each iteration by merging the two closest clusters from the previous level. The coefficients of the model for the new cluster are the coordinates of the euclidean midpoint of the corresponding coefficients of the two clusters. The controller repeats this step until the number of the remaining clusters equals the minimum number of queues in switches.
2. Finding the best clustering: Assuming that a switch supports Q per-port queues, given a set of PLs whose flows pass through a switch output port, the controller finds the best cluster at that port as follows:

(a) Start from level 1 of the hierarchy.
(b) In the current level, if all PLs are grouped into at most Q clusters, go to (c); otherwise, go to the next level and repeat (b).
(c) Map each cluster of PLs to a queue.

Once PLs are mapped to queues, the controller assigns the sum of the bandwidth allocated to applications (Eq 2) associated with each queue as the weight of that queue.

5.4 Centralized vs Distributed Controller

So far, the discussion has implicitly assumed a centralized controller that maintains the global state of application-to-PL and PL clustering operations, as well as the state of each switch, including flows passing through the switch and the current switch configuration. Such a centralized controller updates the application-to-PL mapping and performs PL clustering when a new application is registered or a running application is deregistered. Furthermore, every time a connection is created or destroyed, the centralized controller performs a new PL-to-queue mapping and updates the configuration of switches on the path of that connection. Naturally, a centralized controller represents a single point of failure. In addition, calculating the bandwidth for every application at the scale of a large cluster or a datacenter may result in the centralized controller bottlenecking performance.

An alternative to the centralized controller is a distributed one. Eq 2 indicates that the bandwidth calculation for applications on a given output port is independent of other switches, presenting an opportunity to distribute the controller's logic. In such a distributed design, each controller is responsible for a group of switches and maintains the record of applications sending flows to the associated switches. In order to enforce the bandwidth, the controllers fetch the application-to-PL mapping and the PL clusters from a database. The Saba library informs the closest controller when an application requests a new connection. This controller performs bandwidth allocation and enforcement while communicating with the next controller on the path of the connection to inform it about the added connection. Each controller does the same until all switches on the path of the new connection are configured.

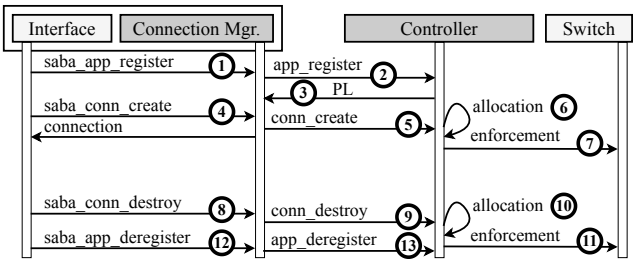


Figure 7. High-level overview of interactions between the software interface, the connection manager, the controller, and a network switch.

As noted above, the distributed controller design requires a database containing the outcomes of application-to-PL mapping and PL clustering operations. *The profiler* updates the database after performing the application-to-PL and PL clustering operations whenever a new application is profiled. Existing replication techniques can be used to replicate the database to increase reliability, availability, and scalability. To maximize performance, database instances can be co-located with the controller nodes.

6 Saba Library

Applications that wish to be Saba-compliant must register themselves at runtime via the Saba library and provide the controller with information about their networking connections. This library consists of two components: the connection manager and the software interface.
Connection manager: The connection manager performs two tasks: communicating with the controller and creating connections. For a given application, the connection manager communicates with the controller to register the application and receives a PL from the controller. When the application requests the creation of a connection, the connection manager creates a new connection and assigns the PL to the connection. The connection manager already possesses the PL received during registration. Therefore, setting up the connection does not introduce any additional overhead. During data exchange (send or receive), packets from all connections associated with the application carry the assigned PL. When the application creates or destroys a connection, the connection manager informs the controller about the creation or destruction of the connection, leading to new bandwidth allocations in the controller.
Software Interface: Saba features a simple software interface to communicate with the controller. Figure 7 illustrates the interactions between the interface, the controller, and a given switch when each function from the interface is called. An application requests registration at start time ①. Saba library informs the controller via the connection manager ②. In response to the registration request, the controller returns

a PL (generated by the application-to-PL clustering) to the connection manager to be used for future connections ③. In order to create a connection, the application sends a request ④. In the Saba library, the connection manager uses the acquired PL to create the connection and returns the connection to the application. Meanwhile, the connection manager informs the controller about the new connection ⑤. The controller considers the new connection and its associated PL for a new allocation ⑥ and enforces the allocation by updating the weights of the queues in the switches ⑦. Once the application no longer needs the connection, it destroys the connection ⑧. The connection manager informs the controller about the finished connection ⑨. This leads to a new allocation ⑩, and the controller updates the weights of queues in the switches ⑪. Before quitting, the application requests for deregistration ⑫, and the connection manager informs the controller to deregister the application ⑬.

7 Implementation

This section provides the details of the implementation of the three components of Saba, *offline profiler*, *controller*, and *Saba library*. While we have implemented Saba on top of an InfiniBand architecture, our design does not make any assumptions about the underlying network layers; e.g., our implementation can be easily ported to Ethernet networks.

7.1 Profiler

The profiler executes the application multiple times on a set of dedicated nodes; in each run, it adjusts the amount of network bandwidth available to the application and measures the application's completion time. Our current implementation considers the following percentages of link bandwidth: 5%, 10%, 25%, 50%, 75% 90% and 100%, which are enforced by a token bucket [35] rate limiter in the InfiniBand driver [46].

7.2 Controller

Path Detection: As explained in §5, the controller receives information about the sources and destinations of each connection from the Saba library. Saba leverages the *infiniband-diags* package provided by OpenFabric [46] and gets the forwarding tables of switches in the network to detect the path of each connection.

Weight Calculation: Saba uses Sequential Least Squares Programming (SLSQP) algorithm from NLopt library [28] to solve the optimization problem in Eq 2 and calculate the weights for the flows passing through a given port.

Bandwidth Enforcement: As explained in §5.2, the controller requires the following from the network to enforce bandwidth: *i)* service differentiation and *ii)* per-port queues in switches with the WFQ scheduling policy. InfiniBand supports service differentiation in its transport protocol and features per-port queues. Further, InfiniBand switches implement a Weighted Round Robin scheduling discipline to approximate WFQ.

InfiniBand offers service differentiation by introducing two concepts: *Service Levels* and *Virtual Lanes* [10].

Service Levels (SLs): InfiniBand supports 16 priority levels, called Service Levels (SLs). SLs are exposed to the developer and can be used to create connections. Once a connection has been created using an SL, the header of packets from the connection will carry the SL through the network.

Virtual Lanes (VLs): InfiniBand divides a physical link into different logical communication links, called Virtual Lanes (VLs), and allocates a queue for each VL at the output ports of switches and NICs. For each VL, InfiniBand provides buffering, flow-control, and congestion management. Switches and NICs handle packets inside VLs in each scheduling turn according to a table that maps SLs with their associated weights to VLs. This table is configurable at every switch and NIC by the datacenter operator. With the current InfiniBand specification, each switch or NIC must support between 2 and 16 VLs [10].

Saba uses SLs to differentiate applications and enforces bandwidth by dynamically setting the VLs' weights at switches.

7.3 Saba Library

The connection manager, implemented with just 350 LOC, uses RPC operations for all control-plane activities. The connection manager creates the low-level InfiniBand connections using *ibverbs* library and assigns SLs to them. In order to register workloads and communicate with the controller, we add Saba API to the Spark and Flink frameworks. To do so, we modified the existing job submission and RDMA-enabled shuffle manager [43] in Spark to invoke Saba's interface functions. The individual workloads required *no* modification to support Saba.

8 Evaluation

We evaluate Saba using experiments on (1) a 32-server InfiniBand testbed with a suite of workloads and (2) a simulated 1,944-server cluster with a set of synthetic workloads.

8.1 Methodology

The main goal of the experiments is to evaluate the impact of Saba on the performance of workloads as compared to a baseline implementing max-min fairness.

Baseline: We use InfiniBand as our baseline, which approximates max-min fairness for each queue in its end-to-end congestion management via Forward Explicit Congestion Notification [3, 54]. In the simulation experiments, we also implemented an idealized version of max-min fairness, which provides an upper bound on the performance achievable by *any* congestion-control protocol targeting max-min fairness.

Table 1. Dataset size of workloads in profiling.

	Workload	Dataset Size
①	LR (Logistic Regression)	10k samples
	RF (Random Forest)	20k samples
	GBT (Gradient Boosted Trees)	1k samples
	SVM (Support Vector Machine)	150k samples
②	NW (NWeight)	# of graph edges: 4250M
③	NI (Nutch Indexing)	100G samples
	PR (PageRank)	50M pages
④	SQL (Join)	Two Tables # of records: 5G & 120M
⑤	WC (WordCount)	300GB
	Sort	280GB

Metric: Our metric of interest is speedup, defined as *the ratio of the performance of a given workload on the Saba-enabled network to the performance of the workload on the baseline system*. Throughout this section, the average speedup reports the geometric mean of the results.

Testbed: We conduct our experiments on a cluster of 32 servers. Each server runs Ubuntu 18.04 and is equipped with two 8-core Intel Xeon E5-2650v2 (Ivy Bridge) CPUs at 2.60GHz. Each CPU has 20 MB of L3 cache and two hardware threads per core, though we disable SMT in our experiments. Each server has 64 GB of system memory and a single-port 56Gb InfiniBand NIC (ConnectX-3) connected on socket 0. NICs are interconnected via a Mellanox SX6036G InfiniBand switch, which supports 9 VLs. We use 8 VLs for Saba-compliant applications. We also use an additional server to run the *centralized* controller. At profiling time, the same server runs the profiler. In all experiments, we reserve 100% of the link capacity to be managed by Saba (i.e., C_{Saba} = 100%).

On the application side, we use the Spark and Flink frameworks. We use ten workloads from Intel's industry benchmarks [26] running on top of Spark and Flink. These workloads include Machine Learning ①, Graph ②, Websearch ③, SQL ④, and Micro ⑤ benchmarks. The workloads and the evaluated dataset sizes are summarized in Table 1.

Simulation: To assess Saba at a larger scale, we implement it in Mellanox's InfiniBand simulator based on OMNeT++ [42]. We configure the simulator to support 56Gbps link capacity per port to match our hardware testbed. Each port supports 16 VLs, each with a dedicated queue. We simulate a representative network configuration with a Spine-Leaf topology and three levels of switches [50]: 54 spine, 102 leaf, and 108 top-of-rack switches. Each top-of-rack switch connects 18 servers, for a total of 1,944 servers in the network. Similar to our testbed setup, we set C_{Saba} to 100%.

We generate 20 distinct synthetic workloads in the simulator. Each workload emulates the computation and communication stages, which is a common pattern in parallel frameworks such as Spark and Flink. The amount of computation, communication, and the number of stages varies across the workloads to emulate varying degrees of bandwidth sensitivity. In the simulation, each server runs one workload. In a topology with 1,944 servers, each of the 20 workloads has 97 instances, which are randomly distributed across the network.

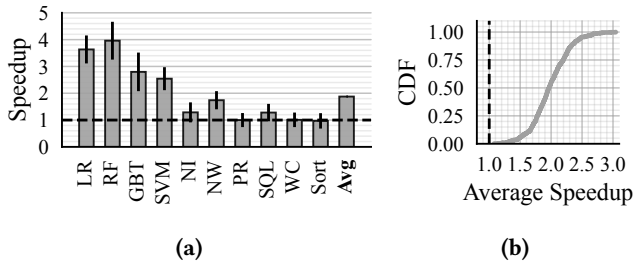


Figure 8. Testbed results. (a) shows the speedup of workloads with Saba over the baseline. (b) plots the CDF of the average speedup of 500 cluster setups.

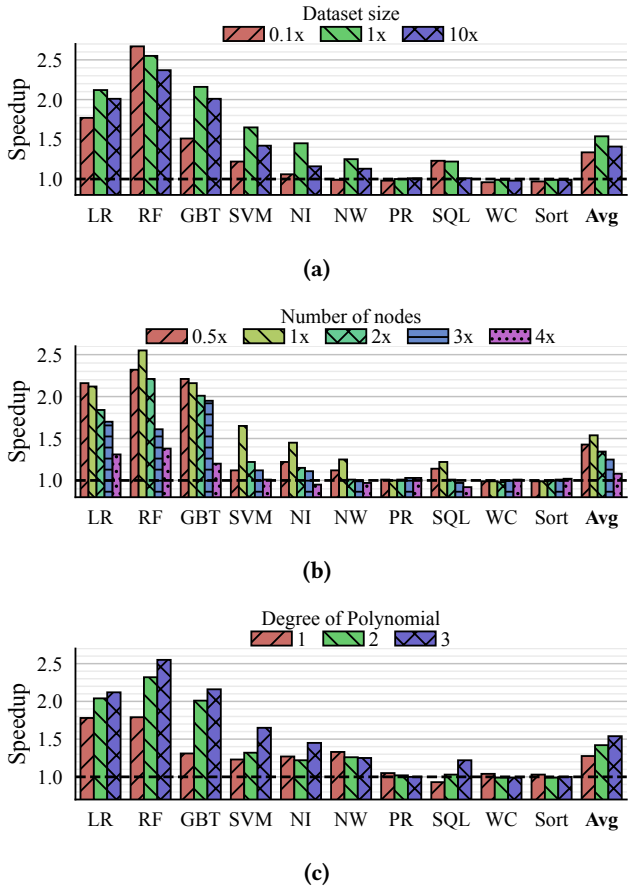


Figure 9. Impact of dataset size (a), the number of nodes (b), and degree of polynomial (c) on speedup of workloads.

8.2 Main results

In this section, we evaluate the performance of Saba in an environment with uneven distribution of traffic on each node, which mimics realistic scenarios in private datacenters. To create such an environment, we generate 500 *cluster setups*. In each cluster setup, 16 jobs are randomly selected by drawing, with replacement, from the set of workloads listed in Table 1. All workloads are profiled in advance with the degree of polynomial of 3 as described in §4.1. The dataset size of each job is randomly selected from 0.1x, 1x, and 10x of the dataset used by the profiler. The number of instances of a job is also randomly selected from 0.5x to 4x of the number of nodes used by the profiler (8 nodes). On each server, one core is assigned to each job, and memory is equally partitioned among all workloads. Instances of jobs are randomly distributed among servers with two constraints: 1) at most one instance of a given job is assigned to a server, and 2) each server accommodates at most 16 jobs. For each cluster setup, we run all jobs together two times, once with Saba managing the bandwidth of jobs and once with the baseline, and we measure the completion time for each job.

Figure 8a displays the average speedup over the baseline for each workload. As the figure shows, Saba improves the average performance across workloads by 1.88× compared to the baseline. The largest performance improvement is observed on workloads with high bandwidth sensitivity. E.g., the performance of RF is increased by 3.9×, and LR by 3.6×. For 2 (out of 10) workloads that have low sensitivity to network bandwidth (Sort and PR), their performance is slightly degraded by 5% and 1% over the baseline, respectively. The reason for the performance drop is that Saba distributes the bandwidth in favor of bandwidth-sensitive workloads. With this objective, workloads that are highly sensitive to bandwidth get larger shares of network bandwidth, while workloads with low sensitivity receive smaller shares. Such an approach significantly improves the performance of bandwidth-sensitive workloads as well as the overall performance of all workloads. However, this redistribution of bandwidth may lead to mild performance degradation for some workloads, as demonstrated by our results.

Figure 8b shows the CDF of the average speedup of 500 cluster setups. As the figure shows, the average speedup ranges from 0.94× to 2.92×. In only 2 (out of 500) cluster setups, Saba results in a performance slowdown compared to the baseline.

8.3 Sensitivity studies

In §4.2, we observed that the degree of polynomial, dataset size, and the number of nodes impact the accuracy of sensitivity models. In this section, we evaluate the impact of these parameters on the performance of Saba. In all scenarios, all workloads are profiled in advance (§4.1). In studies 1 and 2, the degree of polynomial (k) used by the profiler is 3. We vary the degree of polynomial in study 3.

1) Impact of dataset size: As described in §4.2, the accuracy of the sensitivity models declines as the difference in dataset size at profiling and runtime increases. We compare the performance of Saba across various dataset sizes, ranging from 0.1x to 10x the size of the datasets used in the profiling phase. To this end, we create a homogeneous setup in which the number of nodes is the same number of nodes used in the profiling phase (8 nodes) and all nodes run all workloads together. We run one instance of each workload on every server with a core assigned to each workload and memory equally partitioned among all workloads.

Results are shown in Figure 9a. As the figure shows, the applications benefit the most (1.54× speedup) when the runtime dataset size matches the dataset size in the profiling (1x). However, even when datasets are ten times smaller (0.1x) or larger (10x), Saba is still able to obtain performance improvement across workloads. When the workloads use the 0.1x and 10x datasets, Saba improves the average performance by 1.33× and 1.40×, respectively, compared to the baseline.

2) Impact of number of nodes: Similar to dataset size, the accuracy of a sensitivity model decreases as the number of nodes running a distributed application at runtime drifts from the number of nodes in the profiling phase (explained in §4.2). We compare the performance of Saba across a number of nodes, ranging from 0.5x to 4x the number of nodes used by the profiler (8 nodes). Similar to study 1, we run one instance of each workload on every server. In this study, workloads use datasets of the same size as in the profiling.

As Figure 9b shows, Saba achieves a 1.42× average speedup over baseline when the number of nodes is reduced to half. Increasing the number of nodes to 2x and 3x compared to the profiling results in 1.34× and 1.26× average speedup, respectively. When increasing the number of nodes to 4x compared to profiling, we observe that Saba gains only 1.09× average speedup; however, workloads such as SQL, NW, and NI experience 8%, 6%, and 3% drop in their respective performance. This result was expected as §4.2 explained that when the number of nodes at runtime is 4x, the accuracy of sensitivity models significantly drops. We conclude that when the number of nodes used in deployment exceeds the number in the profiling configuration by over 3x, the benefits of Saba significantly diminish.

3) Impact of degree of polynomial: As explained in §4.2, the degree of polynomial plays an important role in the accuracy of the sensitivity models. We compare the performance of Saba while varying the degree of polynomial used by the profiler from 1 to 3. Similar to study 2, we run one instance of each workload on every server. In this study, workloads use datasets of the same size as in the profiling phase and the number of nodes is the same number of nodes used in the profiling phase (8 nodes).

Figure 9c displays the impact of the degree of polynomial on the performance of Saba. As expected, Saba benefits from more accurate sensitivity models. Some workloads benefit from the higher degrees of polynomial. For instance, when the sensitivity model of the SQL workload is generated using second- and third-degree polynomial, SQL experiences 1.03× and 1.22× improvement, respectively, over the baseline. Overall, with the first- and second-degree polynomial, Saba achieves 1.27× and 1.42× average speedup, respectively.

8.4 Simulation Results

To study the performance at a larger scale, we expand the deployment size by using the simulator and running the set of synthetic workloads as explained in §8.1.

Profiling: We calculate the bandwidth sensitivity of workloads by profiling them. For each of the simulated workloads, the profiler deploys instances of the workload on a rack-scale simulated system with 18 nodes (thus mimicking a real deployment). The profiler uses the third-degree polynomial in the following studies to generate a sensitivity model for each workload. In all studies, Saba uses a centralized controller, except for study 7.

4) Saba vs. Ideal Max-Min Fairness:

Ideal max-min fairness: In the ideal implementation of max-min fairness, each workload is assigned to a dedicated queue, and packets from queues are serviced using the Round-Robin algorithm. In this scheme, in each turn, the scheduler in the switch selects a queue and chooses the packet at the head of the queue. Assuming that all packets have the same size if such a scheme transmits one packet in each turn, it achieves the upper bound of max-min fairness [24].

We evaluate the performance of synthetic workloads at a large scale with ideal max-min fairness and compare that with Saba. We run all workloads together in the simulation. Similar to §8.2, Saba uses 8 per-port queues in the switches.

Figure 10 presents the results. As the figure shows, almost all workloads achieve higher performance using Saba compared to using ideal max-min fairness. The maximum speedup achieved by a workload is 1.79×; while the performance of two workloads is penalized by 3%. The average speedup across workloads for Saba and ideal max-min fairness is 1.27× and 1.14×, respectively. This shows that per-flow max-min fairness is inherently unable to directly target the application-level performance, as it tries to achieve bandwidth fairness at the flow level, but ignores the fact that some workloads are more sensitive than others. In contrast, Saba allocates the bandwidth of each network link based on the bandwidth sensitivity of the workloads using it. Thus, workloads with higher sensitivity get more bandwidth and see a performance improvement over max-min fair allocation.

5) Saba vs. Homa: In this study, we compare Saba against the recently-proposed Homa [44], which is considered the state-of-the-art networking protocol designed for datacenters. Similar to Saba, Homa leverages the priority queues available in network switches. Homa prioritizes short flows to achieve optimal flow-level completion time. We use Homa's OMNet++ simulator with the same topology and set of synthetic workloads described in §8.1.

As Figure 10 shows, Homa achieves 1.12× speedup over the baseline. Saba outperforms Homa by an additional 15%. The reason that bandwidth-sensitive workloads benefit from Saba more than Homa is the fact that Homa differentiates flows based on their size. E.g., in this setup, Homa assigns all flows longer than a certain size (10KB) to the *same* priority queue, without differentiating their associated workloads; thus, Homa does not allocate bandwidth in favor of bandwidth-sensitive workloads and the application-level performance of workloads is ignored. Saba, however, differentiates workloads at runtime based on their application-level sensitivity to bandwidth, resulting in improved performance.

6) Saba vs. Sincronia: To bridge the gap between flow-based bandwidth allocation schemes and application-level bandwidth requirements, a networking abstraction, called coflow, has been proposed recently [15]. Coflow represents a collection of related flows to convey application-specific communication requirements. In this study, we compare Saba against Sincronia[2], which is the state-of-the-art clairvoyant coflow scheduler. Sincronia tries to minimize the coflow completion time by ordering all unfinished coflows and assigning priority levels to the flows according to their coflow order. Sincronia requires flow sizes to be known a priori. While such a requirement is not feasible in datacenters [63], it provides Sincronia with near-optimal coflow completion time. To enforce the allocated rates, Sincronia leverages the underlying priority-enabled transport layer.

As Figure 10 shows, Sincronia achieves 1.19× speedup over the baseline. Saba outperforms Sincronia by an additional 8%. Indeed, the coflow abstraction allows workloads to more accurately express their bandwidth requirements to the network fabric; however, it does not take the sensitivity of workloads into account and the overall application-level performance of workloads is ignored. In addition, Sincronia, like other coflow-based approaches, needs applications to be modified and invoke coflow API [15]; Saba, however, requires *no* modification to applications.

7) Centralized vs. Distributed: In this study, we evaluate the impact of the centralized versus distributed controller on Saba's performance. We repeat Study 4 with the distributed controller. As explained in §5.4, in Saba, with the distributed controller, the profiler performs the application-to-PLs and hierarchical clustering operations offline. Thus, the controller may not have the most accurate mappings, leading to a trade-off between performance and scalability.

As Figure 11a demonstrates, Saba with the distributed controller is able to achieve a 1.23× speedup over the baseline, falling just 4% short of Saba with the centralized controller. We conclude that using the distributed controller slightly reduces the effectiveness of Saba while improving scalability.

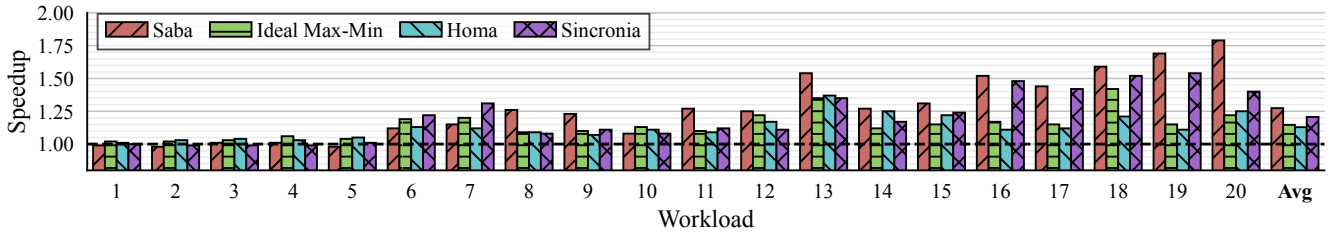


Figure 10. Speedup of Saba, ideal max-min, Homa, and Sincronia, all over the baseline.

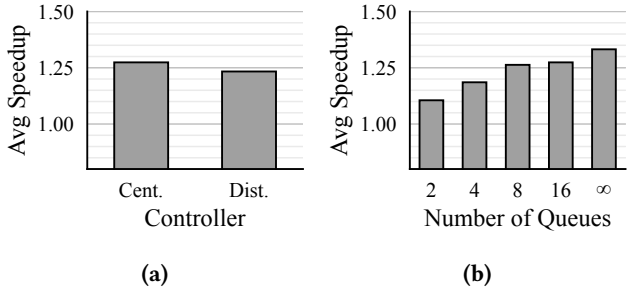


Figure 11. (a) the average speedup with centralized versus distributed controllers. (b) impact of the number of queues on the performance of Saba.

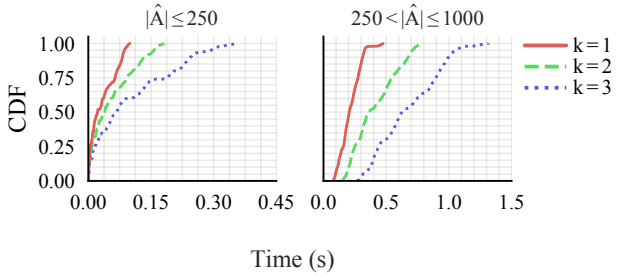


Figure 12. The overhead of a centralized controller.

8) Impact of number of queues: The number of queues per output port varies among switches used in today's datacenters. 8 queues are common, though more capable switches may support 16 queues per port [44]. We study the impact of the number of queues in network switches on the performance of Saba. To do so, we repeat study 4 and use switch configurations with 2, 4, 8, and 16 queues per port. We also study a configuration with unlimited queues, where each workload is assigned to a dedicated queue; this configuration provides an upper bound on Saba's performance.

Figure 11b plots the results of the study. Compared to the baseline, Saba delivers a 1.12× average performance improvement with just 2 queues at each output port. With 8 queues, Saba achieves a speedup of 1.27×, approaching the ideal speedup of 1.33× with an unlimited number of queues. This result shows that 8 queues, which is common in today's datacenter switches, is sufficient for Saba to achieve close to its optimal performance.

8.5 Overhead of the controller

In study 3, we observed that modeling the bandwidth sensitivity of workloads with higher degrees of polynomial results in higher performance. In this study, we investigate the added overhead by higher degrees of polynomial in bandwidth calculation on a large scale in the simulator. To this end, we evaluate the *calculation time* of a centralized controller, i.e., the time the controller takes to compute the bandwidth share of applications for all switches. We generate 30,000 scenarios, in which the size of the active application set varies from 1 to 1,000. In each scenario, 32 instances of each application are randomly distributed among nodes.

Figure 12 displays the CDF of measured calculation time with various sizes of active application sets ($|\hat{A}|$). The result shows that for sets of applications with sizes up to 250, the calculation time of the controller at 99th percentile is 0.09s, 0.16s, and 0.31s for $k = 1$, $k = 2$ and $k = 3$, respectively. By increasing the size of applications set up to 1,000, the calculation time of the controller at 99th is 0.43s, 0.72s, and 1.13s for $k = 1$, $k = 2$ and $k = 3$, respectively. Despite the higher accuracy achieved by sensitivity models with higher degrees of polynomial, the increased overhead in bandwidth calculation reduces the responsiveness of the controller. While the degree of polynomial must be carefully tuned according to the number of applications in the deployment of Saba, the calculation time of the controller even with $k = 3$ is negligible as compared to the runtime of workloads. To put the calculation time in perspective, the studied workloads take from several minutes to hours to finish their job, while in an extreme case, the calculation time of a centralized controller for 1,000 applications takes 1.13 seconds. Moreover, the datacenter operator can distribute the controller, or use a multi-threaded implementation of the controller and run it on multiple cores or accelerators such as FPGAs to reduce the calculation time.

9 Related Work

Datacenter bandwidth allocation protocols: Mainstream proposals for bandwidth allocation in datacenter networks can be broadly classified into two categories. In the first one, there are protocols like DCTCP [5], NDP [25], and Swift [31] that aim to achieve max-min fairness while keeping low queue utilization. In the second one, there are solutions like Homa [44], pFabric [6], pHost [19], and Sincronia [2] that explicitly target reducing flow completion time as the primary metric, possibly using policies such as shortest-flow-first [6, 44]. However, as we discussed in §2.2, they all optimize for network-level metrics and ignore the impact of bandwidth sensitivity for different applications, which can result in poor aggregate application performance. In contrast, Saba estimates the impact of bandwidth on applications and leverages this information to distribute network bandwidth and achieve shorter application completion times.

Application-aware bandwidth allocation: There have been a few attempts to use the application-level network demands in bandwidth allocation, many of which are specialized for specific types of applications and require software modification. Coflow [2, 14–16] introduces a new network abstraction and network API for data-parallel applications. AppSch [36] relies on a priori knowledge of the flow size of all flows in applications and allocates paths for their flows. Saba requires no modification to applications and no prior knowledge of flow size. Rajasekaran et al. observe that unfairly (unequally) sharing the network among ML jobs could lead to shorter training time due to the on-off pattern of DNN training [49]. In contrast, Saba does not make any assumption about the specific pattern of communication and computation, and proposes a general methodology to allocate bandwidth across a heterogeneous set of workloads.

Performance prediction: Performance prediction through application modeling has been explored in recent works. Ernest [56] models the structure of machine learning jobs to predict their performance; Saba, however, does not make any assumptions about the internal structure of applications. CherryPick [4] builds performance models and leverages information of history jobs, aiming at finding a cloud configuration (CPU, memory, number of nodes, etc) for a single application that minimizes the cost of executing the application. Saba focuses on bandwidth allocation for co-running applications that share the network.

10 Conclusion

In this paper, we demonstrate the shortcomings in bandwidth allocation approaches that are based on max-min fairness or shortest-flow first on a per-flow basis. We show that such allocation schemes do not efficiently utilize the network in shared environments like datacenters, as they are unable to identify the bandwidth demands of applications. We introduce Saba, an application-aware bandwidth allocation scheme, which determines the sensitivity of applications to bandwidth, and allocates bandwidth to applications according to their bandwidth sensitivity. Our evaluation shows that Saba improves the performance of co-located workloads compared to existing and ideal implementations of max-min fairness.

11 Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and feedback. This research was generously supported by the University of Edinburgh and by EASE Lab's industry partners and sponsors including Huawei, Intel and Microsoft.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*. 265–283.

[2] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: Near-Optimal Network Design for Coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3230543.3230569

[3] Fatma Alali, Fabrice Mizero, Malathi Veeraraghavan, and John M. Dennis. 2017. A measurement study of congestion in an InfiniBand network. In *2017 Network Traffic Measurement and Analysis Conference (TMA)*. 1–9. https://doi.org/10.23919/TMA.2017.8002911

[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 469–482.

[5] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP).. In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: minimal near-optimal datacenter transport.. In *Proceedings of the ACM SIGCOMM 2013 Conference*. 435–446.

[7] Rohan Anil, Gabriel Pereyra, Alexandre Passos, Róbert Ormándi, George E. Dahl, and Geoffrey E. Hinton. 2018. Large scale distributed neural network training through online distillation.. In *ICLR (Poster)*.

[8] Apache. [n. d.]. *Flink*. http://flink.apache.org

[9] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark.. In *SIGMOD Conference*. 1383–1394.

[10] InfiniBand Trade Association. 2015. *InfiniBand architecture specification volume 1 and 2*. Technical Report.

[11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Antony I. T. Rowstron. 2011. Towards predictable datacenter networks.. In *Proceedings of the ACM SIGCOMM 2011 Conference*. 242–253.

[12] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. 2013. Chatty Tenants and the

Cloud Network Sharing Problem.. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*. 171–184.

[13] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. 2016. Scheduling Mix-flows in Commodity Datacenters with Karuna.. In *Proceedings of the ACM SIGCOMM 2016 Conference*. 174–187.

[14] Mosharaf Chowdhury, Samir Khuller, Manish Purohit, Sheng Yang, and Jie You. 2019. Near Optimal Coflow Scheduling in Networks.. In *SPAA19*. 123–134.

[15] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: a networking abstraction for cluster applications.. In *Proceedings of The 11st ACM Workshop on Hot Topics in Networks (HotNets-XI)*. 31–36.

[16] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with Varys.. In *Proceedings of the ACM SIGCOMM 2014 Conference*. 443–454.

[17] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters.. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*. 137–150.

[18] Alan J. Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm.. In *Proceedings of the ACM SIGCOMM 1989 Conference*. 1–12.

[19] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. pHost: distributed near-optimal datacenter transport over commodity network fabric.. In *Proceedings of the 2015 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*. 1:1–1:12.

[20] S. Jamaloddin Golestani. 1995. Network Delay Analysis of a Class of Fair Queueing Algorithms. *IEEE J. Sel. Areas Commun.* 13, 6 (1995), 1057–1070.

[21] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI)*. 599–613.

[22] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR* abs/1706.02677 (2017).

[23] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. 2010. SecondNet: a data center network virtualization architecture with bandwidth guarantees.. In *Proceedings of the 2010 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*. 15.

[24] Ellen L. Hahne. 1991. Round-Robin Scheduling for Max-Min Fairness in Data Networks. *IEEE J. Sel. Areas Commun.* 9, 7 (1991), 1024–1039.

[25] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance.. In *Proceedings of the ACM SIGCOMM 2017 Conference*. 29–42.

[26] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis.. In *ICDE Workshops*. 41–51.

[27] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert G. Greenberg, and Changhoon Kim. 2013. EyeQ: Practical Network Performance Isolation at the Edge.. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*. 297–311.

[28] Steven G. Johnson. 2011. *The NLopt nonlinear-optimization package*. http://ab-initio.mit.edu/nlopt

[29] M. R. Siavash Katebzadeh, Paolo Costa, and Boris Grot. 2020. Evaluation of an InfiniBand Switch: Choose Latency or Bandwidth, but Not Both.. In *ISPASS20*. 180–191.

[30] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop.. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*.

[31] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Mike Ryan, David J. Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conferenced*.

[32] Katrina LaCurts, Jeffrey C. Mogul, Hari Balakrishnan, and Yoshio Turner. 2014. Cicada: Introducing Predictive Guarantees for Cloud Networks.. In *Proceedings of the 6th workshop on Hot topics in Cloud Computing (HotCloud)*.

[33] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. 2020. Sol: Fast Distributed Computation Over Slow Networks.. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*. 273–288.

[34] Jeng Farn Lee, Meng Chang Chen, and Yeali S. Sun. 2007. WF. *Comput. Networks* 51, 6 (2007), 1403–1420.

[35] Tsern-Huei Lee. 2004. Correlated token bucket shapers for multiple traffic classes. In *IEEE 60th Vehicular Technology Conference, 2004. VTC2004-Fall. 2004*, Vol. 7. 4672–4676 Vol. 7. https://doi.org/10.1109/VETECF.2004.1404977

[36] Andrew Lester. 2014. Application-Aware Bandwidth Scheduling for Data Center Networks. 7, 3 (2014), 194–205.

[37] Colin Lewis-Beck and Michael Lewis-Beck. 2015. *Applied regression: An introduction*. Vol. 22. Sage publications.

[38] Shigang Li, Tal Ben-Nun, Salvatore Di Girolamo, Dan Alistarh, and Torsten Hoefler. 2020. Taming unbalanced training workloads in deep learning with partial collective operations.. In *PPoPP20*. 45–61.

[39] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2014. GraphLab: A New Framework For Parallel Machine Learning. *CoRR* abs/1408.2041 (2014).

[40] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA, 281–297.

[41] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing.. In *SIGMOD Conference*. 135–146.

[42] Mellanox. [n. d.]. *IB flit simulator*. https://oss.iol.unh.edu/research/ib_flit_sim

[43] Mellanox. [n. d.]. *SparkRDMA*. https://github.com/Mellanox/SparkRDMA

[44] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. 2018. Homa: a receiver-driven low-latency transport protocol using network priorities.. In *Proceedings of the ACM SIGCOMM 2018 Conference*. 221–235.

[45] Daniel Müllner. 2013. fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python. *Journal of Statistical Software* 53, 1 (2013), 1–18.

[46] OFED. [n. d.]. *OFED*. https://www.openfabrics.org/ofed-for-linux/

[47] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. 2016. Big Data Analytics over Encrypted Datasets with Seabed.. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*. 587–602.

[48] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. 2013. ElasticSwitch: practical work-conserving bandwidth guarantees for cloud computing.. In *Proceedings of the ACM SIGCOMM 2013 Conference*. 351–362.

[49] Sudarsanan Rajasekaran, Manya Ghobadi, Gautam Kumar, and Aditya Akella. 2022. Congestion control in machine learning clusters. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. 235–242.

[50] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network.. In *Proceedings of the ACM SIGCOMM 2015 Conference*. 123–137.

[51] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation.. In *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*. 785–808.

[52] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX.. In *IEEE Symposium on Security and Privacy*. 38–54.

[53] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System.. In *MSST10*. 1–10.

[54] Chakchai So-In, Raj Jain, and Jinjing Jiang. 2008. Enhanced Forward Explicit Congestion Notification (E-FECN) scheme for datacenter Ethernet networks. In *2008 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*. 542–546.

[55] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. 2012. Deadline-aware datacenter tcp (D2TCP).. In *Proceedings of the ACM SIGCOMM 2012 Conference*. 115–126.

[56] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 363–378.

[57] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Rao Kompella. 2012. The only constant is change: incorporating time-varying network reservations in data centers.. In *Proceedings of the ACM SIGCOMM 2012 Conference*. 199–210.

[58] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale.. In *SIGMOD Conference*. 13–24.

[59] Hailong Yang, Alex D. Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers.. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*. 607–618.

[60] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. 2018. Image Classification at Supercomputer Scale. *CoRR* abs/1811.06992 (2018).

[61] 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. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[62] Haishan Zhu, David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Mattan Erez. 2019. Kelp: QoS for Accelerated Machine Learning Systems.. In *HPCA19*. 172–184.

[63] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. 2019. Is advance knowledge of flow sizes a plausible assumption? *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019* (2019), 565–580. https://github.com/vojislavdjukic/flux.