Superways: A Datacenter Topology for Incast-heavy workloads

Hamed Rezaei
hrezae2@uic.edu
University of Illinois at Chicago

Balajee Vamanan
bvamanan@uic.edu
University of Illinois at Chicago

ABSTRACT

Several important datacenter applications cause incast congestion, which severely degrades flow completion times of short flows and throughput of long flows. Further, because most flows are short and the incast duration is shorter than typical round-trip times, reactive mechanisms that rely on congestion control are not effective. While modern datacenter topologies provide high bisection bandwidth to support all-to-all traffic, incast is fundamentally a many-to-one traffic pattern, and therefore, requires deep buffers or high bandwidth at the network edge.

We propose *Superways*, a heterogeneous datacenter topology that provides higher bandwidth for *some* servers to absorb incasts, as incasts occur only at a small number of servers that aggregate responses from other senders. Our design is based on the key observation that a small subset of servers which aggregate responses are likely to be network bound, whereas most other servers that communicate only with random servers are not. *Superways* can be implemented over many of the existing datacenter topologies and can be expanded flexibly without incurring high cost and cabling complexity. We also provide a heuristic for scheduling jobs in our topology to fully utilize the extra capacity. Using a real CloudLab implementation and using ns-3 simulations, we show that *Superways* significantly improves flow completion times and throughput over existing datacenter topologies. We also analyze cost and cabling complexity, and discuss how to expand our topology.

CCS CONCEPTS

• **Networks** → **Layering**; **Network management**; **Network protocol design**.

KEYWORDS

datacenter networks, incast control, network topology

ACM Reference Format:

Hamed Rezaei and Balajee Vamanan. 2021. ***Superways*: A Datacenter Topology for Incast-heavy workloads. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3449966**

WWW '21, April 19–23, 2021, Ljubljana, Slovenia
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3449966

1 INTRODUCTION

Datacenters provide on-demand access to vast amounts of Internet data. The applications that run in datacenters can be broadly classified as foreground applications, which perform distributed lookups to service user queries, and background applications, which periodically reorganize, backup, and replicate data. For each query, foreground applications fetch data from a large number of servers. The queries, therefore, cannot complete before receiving replies from most of the servers. Consequently, their response time is determined by the tail (e.g., 99%-ile) of network flow completion times (FCT) [31]. Further, it is well known that foreground applications generate mostly short flows (e.g., 1KB to 10KB) [52, 64] and background applications generate mostly long flows (e.g., 1MB to 100MB) that are sensitive to network throughput [17, 18].

Because foreground applications concurrently fetch data from a set of servers, they cause synchronized responses from servers that cause severe queue buildup at the switch port connected to receiver server. This phenomenon, called *incast*, is known to dilate tail FCTs (i.e., degrade response times) and cause packet drops (i.e., reduce goodput) [18, 28]. Because datacenter switches use shallow buffers [23, 25, 63], it is likely that short, incast flows overrun the port buffer. Our experiments (Section 5) confirm this behavior.

Incasts are synchronized bursts of many-to-one short flows that fundamentally cause an over-subscription of the receiver (aggregator) link. Recent measurements on datacenters show that the duration of most incasts (i.e., $<50\mu s$) is shorter than typical round-trip times (RTT) (i.e., 100 μs) [64]. Most congestion control approaches [18, 21, 29, 35, 60] require at least a one RTT to react, which is absolutely late. Packet scheduling approaches prioritize short flows [20], or flows with near deadlines [44], but do not help when short, incast flows that have similar sizes (or deadlines) contend for limited buffer capacity, which is the common case.

Today's incast-heavy applications (e.g., Web search, social networks) and high-bandwidth network topologies (e.g., fat-trees [40]) imply that receiver congestion is more severe than congestion in the network core, as reported by Facebook [52], Google [55], and Microsoft [38]. While existing datacenter topologies improve *all-to-all* throughput by providing high bisection bandwidth (e.g., [17, 58]), which helps applications such as those applications that perform data-mining tasks, these topologies do not alleviate incast. *Our insight is that effectively handling incasts, which are short-lived many-to-one traffic bursts, require either deep buffers or high bandwidth at the network edge, to absorb these bursts.* Designing large buffers that operate at high speed is hard [22, 37] and the associated silicon-level changes would likely impede early deployment [19]. Further, large buffers cause high latency and could even render some congestion control algorithms unstable [42]. Recent works such as [18], [20], and [29] show that using large buffers exacerbates tail flow completion times as it increases waiting time of packets by queuing them behind large flows.

We make the *key* insight that Online Data-Intensive (OLDI) applications (i.e., foreground applications) partition work such that a few *aggregator* processes gather data from a large number of *worker* processes. A direct implication of this behavior is that the aggregators tend to be network bound while they are not CPU bound at all. Our analysis of CPU and network utilizations of aggregator and worker processes using the open source *Apache Solr* search engine [59] confirms this assumption (see Figure 13 and Figure 14). Motivated by this insight, we propose *Superways*, a heterogeneous network topology optimized for incast-heavy applications. *Superways* is a datacenter topology that provides additional links *only* for those servers that act as incast aggregators (i.e., the server that receives synchronized responses). Because increasing the bandwidth of *all* edge links is expensive and not all edge links are equally likely to experience incast, our proposal provides additional bandwidth only for incast aggregators.

Unlike existing homogeneous topologies where jobs can be scheduled on any random server, our heterogeneous design has a direct implication for the cluster scheduler. Toward that end, we propose a *greedy* heuristic that maps incast aggregators to the topology using their incast degrees (i.e., the number of concurrent connections), which may be obtained using offline profiling or using domain knowledge of applications. Our heuristic simply attempts to map the most incast-heavy job first and proceeds in the order of decreasing marginal utility (i.e., an application that stands to gain the most with an additional link gets picked first). We elaborate on this heuristic in Section 3.

While there are some existing topologies that provide additional links for servers (e.g., Subways [41], Bcube [33]), their homogeneous design is a serious limitation when you consider the heterogeneity in the link congestion; only *a small fraction* of incast aggregators need more receive bandwidth yet these proposals provision expensive receive bandwidth *equally* in *all* nodes. In other words, for a given *cost*, these schemes are not as effective as *Superways* because they do not exploit the inherent heterogeneity in the network traffic and the mismatch between CPU and network utilizations between incast aggregators and worker servers. We summarize our contributions as follows:

- We propose *Superways*, a heterogeneous datacenter topology that is optimized for incast-heavy workloads, which cause receiver congestion in the *common case*.
- We present a greedy heuristic for placement of applications with varying incast degrees in our heterogeneous topology.
- We implemented *Superways* on a real testbed (CloudLab [50]) and show that our proposal improves tail FCT and throughput by factors of 1.83x and 1.38x over a leaf-spine topology.
- Our large-scale simulations show that *Superways* achieves substantial improvements in both tail FCT and throughput over existing proposals (leaf-spine, Bcube [33], Jellyfish [58], and Subways [41] topologies): *Superways* improves tail FCT by a factor of 1.8x (reduce by 45%) and throughput by a factor of 1.2x, over BCube, Jellyfish, Leaf-spine, and Subways, on average.

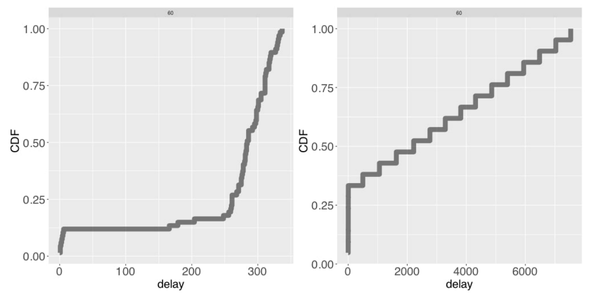


Figure 1: Homa's performance with and without incast of extremely short flows

2 MOTIVATION

There has been a renewed interest in congestion control for datacenters over the last few years. Recent studies [52] on datacenter traffic reveal that most flows are extremely small. Because most congestion control schemes require multiple round trips to adjust rates, they are not effective when flows are very small. Recently, a receiver-driven congestion control called Homa [45] has been proposed, which improves the tail latency compared to its predecessors. Homa approximates SRPT by scheduling packet transmissions from the receivers. In fact, Homa divides each flow to unscheduled and scheduled parts, and sends the scheduled part if and only if the receiver asks to do so when it receives the unscheduled part. However, since datacenter flows are extremely small, it is difficult to divide them and almost all the data will be transmitted as a unscheduled (i.e., no grant needed) data portion. Therefore, many packets will be dropped when incast of short flows happens. We simulated a small scale datacenter in OMNET++ [61] to check performance of Homa when incasts of extremely short flows exist. Figure 1 shows the result of this experiment at 60% load. The left hand side figure shows Homa's performance when incast does not exist and the figure on the right shows its performance when incast of short flows exists. X-axis shows the total delay in milliseconds. We see from the figure that some packets that fall in tail are dropped even more than twice (RTO=3). Thus, receiver-driven methods such as Homa can help but only after the first RTT, which is late in such networks that require very low latency (i.e., zero packet drop).

The current trend of increasing intensity of short flows and incast imply that congestion control schemes are unlikely to solve this problem. Therefore, modern datacenters must resort to other solutions such as large (shared) buffers or use higher capacity topologies such as Subways [41] that provides higher bandwidth for endhosts. Modern shared-buffer switches are equipped with dynamic sharing technologies such as Dynamic Threshold (DT) [30] or Enhanced Dynamic Threshold (EDT) [53]. Although these methods perform well in case of mild congestions, they face various issues when incast congestion happens. For example, DT always keeps a specific amounts of buffer reserved for other inactive ports, while a port that experiences incast might need it. To further evaluate DT's performance, we simulated a datacenter network using reported numbers in previous studies (e.g., [52] and [64]) to measure the performance of DT in case of incast. we explain the workload details later in section 5. Figure 2 shows the performance of DT in terms of tail FCT. We set the Retransmission timeout to 3 seconds. Figure 2 shows

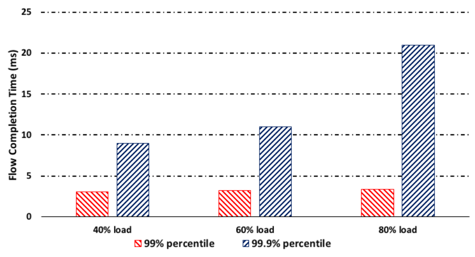
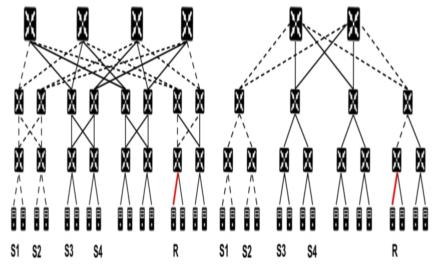


Figure 2: DT's performance in shared buffers



(a) Over-subscription factor of 1 (b) Over-subscription factor of 2

Figure 3: Fat-tree with and without over subscription

that DT suffers from excessive packet drops at high loads (60% to 80%) due to its design that keeps the buffer underutilized. Therefore, while DT is a great solution for mildly congested networks, it will not help in busty environments such as datacenter networks. EDT, however, performs better; but it may hurt fairness as it assigns the whole buffer to a single port when a burst of packets arrives at that port. In such cases that more than one server is experiencing incast, EDT performs even worse.

We argued that neither congestion control methods nor existing dynamic buffer management schemes are able to solve incast problem. But what about datacenter topologies?

Many topologies have been proposed for datacenter networks [17, 57, 58] in past few years; most are designed to improve throughput by increasing the network bisection bandwidth. One of the popular designs is the *hierarchical* design, which uses multiple levels of switches to increase bandwidth between a pair of racks. Hierarchical design increases the available bandwidth between server pairs by providing multiple paths from source to destination. The exact number of paths between racks depends on the over-subscription ratio of the topology. Higher over-subscription ratio means the network has less uplink capacity compared to downlink capacity and there is more chance for forwarding multiple flows over the same path. Figure 3 shows two fat-tree topologies; one with over-subscription ratio of one (Figure 3 (a)) and the other one with over-subscription ratio of two (Figure 3 (b)). As we see in Figure 3, flows are more likely to share the same path in Figure 3 (b) compared to Figure 3 (a).

High bisection bandwidth is not a solution for incast problem. Consider an incast scenario in which 4 servers (S1, S2, S3, and S4) are sending data to a single receiver (R), simultaneously. Since the bottleneck is the last hop switch that connects to the receiver (shown in red), although chance of core layer congestion is much less in the non-blocking topology (i.e., Figure 3 (a)), both non-blocking and oversubscribed topologies perform poorly when incast happens. On the other hand, topologies such as Bcube and Subways that provide high bandwidth at the network edge are prohibitively expensive, which makes them almost impossible to implement.

Hierarchical topologies come with some issues (e.g., cost inefficiencies, low throughput, etc) that motivated researchers to work on topologies such as Jellyfish [58], Proteus [56], and [57] that use random graphs rather than trees. Jellyfish [58] uses random graphs to connect servers and switches. Jellyfish's design is unstructured, which improves throughput of long flows. However, in case of incast, it will not help as incast happens at the network edge rather than network core.

3 *SUPERWAYS*

The high-level idea of *Superways* is to provision more bandwidth for incast aggregators. We argue that providing more bandwidth for incast aggregators is achievable because incast aggregators contribute to small fraction of total number of servers in a datacenter network. Our analysis on publicly available Facebook datacenter traffic dataset shows that around 3% of servers are likely incast applications, as they receive a large number of flows in a short period of time. These servers receive 18 times more flows compared to the rest of servers (i.e., 97% of servers), on average.

Above analysis shows that the number of incast applications in a real datacenter is reasonably small so that providing additional links only for incast applications is feasible. A naive approach is to provision one link from leaf switch to incast aggregator for each worker server, so that all worker servers can send at full rate. However, this would be prohibitively expensive. Further, incast degree would vary drastically across applications, and perhaps may even vary with time. To tackle this problem, we leverage the key observation that most incast flows tend to be short, and are unlikely to utilize the full line rate. Therefore, we propose provisioning receive bandwidth based on a number of factors, including the number of links, average size of flows, and buffer sizes of routers. We discuss the details of our link provisioning method later in Section 3.1.

Although providing additional links for servers is not a new idea, *Superways* brings two novelties: (1) it selectively provides additional links **only** for those servers that require more bandwidth (i.e., those servers that host incast applications), and (2) it greedily finds the optimal number of additional links proportional to incast degrees of incast applications. At its core, *Superways* co-locates incast applications on certain physical servers (e.g., multiple virtual machines each of which holds one or more incast applications) and then greedily finds the optimal number of additional links that are required to absorb all those incast applications' packets combined. For example, we select a physical server in one of the racks to move a certain number of incast applications to this server, and then we calculate the number of additional links required for serving all the incast applications' traffic combined.

The question is: is this feasible to relocate applications in the network? Fortunately, due to rapidly growing use of containerized applications in datacenters, incast applications can be deployed in containers, and therefore, managing these applications (e.g., moving them to another cluster) is easy to do using application container management tools such as Kubernetes [26]. Nowadays, almost all modern datacenters use containerized applications, which is a great advantage for topologies such as *Superways* that require freedom in server placement. Moreover, recent studies such as [39] show that although traffic characteristics such as load and packet sizes change frequently, large scale network characteristics such as workload changes infrequently. Thus, there is no need to constantly relocate the incast applications when *Superways* is implemented.

In contrast to other datacenter topologies that provide additional links between servers and leaf switches, *Superways* connects incast applications directly to spine switches (or inner switches in random graph topologies) to make a trade-off between cost and performance. By connecting to spine switches, *Superways* does not saturate uplink capacity (i.e., links from leaf to spine switches), which improves throughput and tail latency of the flows. This is critically important because provisioning additional links between servers and leaf switches will increase the over-subscription ratio of the network, which increases the number of in-network packets. Note that the network capacity remains unchanged. This issue leads to unpredictable consequences such as excessive packet drops and high queuing delays. That is why previous studies that provision additional links from servers to leaf switches require more leaf switches to absorb the extra packets. Also, providing additional links from servers to spine switches may seem more expensive compared to other methods that connect to leaf switches; however, since spine switches are more powerful in terms of link speed and number of links, *Superways* cuts the overall cost by decreasing the number of required cables, number of server NICs, and number of additional switches. Note that link speed directly affects the number of required links. For example, a zero-buffer switch port that operates at 40 Gbps, supports 4 senders that are sending at 10 Gbps, without loss of throughput and without any packet drops. We will further discuss the reasons for connecting to spine switches in section 5.

3.1 Placement heuristic

Knowing the incast degree of each of incast applications is key in finding the number of additional links. This could be achieved through monitoring the network traffic or previous knowledge about the number of machines that each incast application will query. Although each incast application is expected to query a fixed number of servers, it may query a different number of servers in some cases, depending on the task that is requested by the end user. We argue for considering average incast degree of each of incast applications. We will discuss our decision in details later in section 5. Once we measured the average incast degree of each incast application, we sort them in a *descending* order to see which of incast applications needs more additional link(s) (i.e., more bandwidth+buffer space) and which ones require less or even no extra links.

We start assigning the incast application with *highest* average incast degree to a physical server. This can be done through moving the VM that hosts this incast application to an elected physical server, or installing the incast application on the elected server in a containerized environment. The physical server that hosts one or more incast applications is called *incast server* throughout this paper. We start from one of the racks that is closest to the rack of spine switches, and then we pick one of the servers in this rack as our elected server, which will be hosting the first incast application. In random graph topologies, we use the same approach as we need to keep the wiring lengths as short as possible.

Once we placed the first incast application with highest incast degree on the designated incast server, we have multiple rounds of calculations to find the minimum number of additional links. Table 1 shows the parameters that we use in these calculations. Below we will explain some of these parameters.

Due to various delays at kernels (e.g., handling interrupts, etc), all incast senders do not start sending their data at the same time. Therefore, we consider a jitter factor in our formulas (shown by α). α should be set to a value close to 1 (e.g., 0.8 - 0.9). In our experiments, we observed that about 80% of the sender servers send their messages at the same time.

We know from previous studies that incast senders of a particular incast application can be anywhere in the network. We call those incast servers that reside on a different rack *remote incast senders* (e.g., S3 through S6 in Figure 4). We need to differentiate remote and local incast senders because local incast senders could be able to use the current link (i.e., leaf switch to server) without requiring any extra links. Also, even remote incast senders might be able to use the current link without any packet drops. This will definitely change *Superways*' design. Therefore, we use β to show the ratio of local incast senders to the total number of incast senders (β = 1/3 in Figure 4).

Average flow size varies depending on the workload in different datacenter networks. However, as shown in figure 6 of [52], about 60% of the flows are 1 KB or below in Web search workloads. Our analysis on Facebook datacenter traffic confirms that flows in Web search workloads are mostly small so that the average flow size for all flows is close to 1500 bytes. Although this number may vary among different datacenters and different workloads, the nature of current datacenter workloads implies that most flows are in the range of 1 KB to 10 KB in size. Similar numbers have been reported in previous studies such as [32].

In the first round of calculations, we use formula 1 to check if incast senders that reside on the same rack (i.e., local incast senders: S1 and S2 in Figure 4) exceed the buffer capacity of the existing link from leaf switch to incast server. Since buffer size of each port plays a key role in handling incast (i.e., reduces packet drops by storing packets in a queue), we consider both bandwidth and buffer size in our calculations.

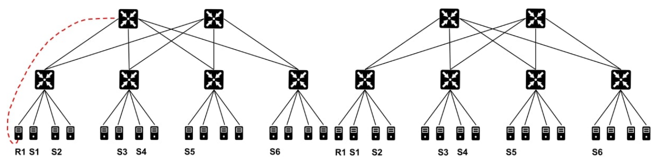
$$U_l = \frac{\alpha \times (\beta \times d) \times s}{B_l} \tag{1}$$

U_l is the ratio of used buffer to the total buffer capacity of the link that connects the leaf switch to an incast server, and therefore, could be either less or greater than one. We analyze both cases in the following sections.

3.1.1 U_l is greater than one. If U_l is greater than one, then it means local incast senders of that particular incast application will overflow the leaf switch's buffer. For example, although incast degree of R in Figure 4b is 6, if S1 and S2 (local incast senders) start sending their data at the same time, they will overflow the leaf switch's

Table 1: Parameters and descriptions

Parameter	Description
α	Jitter factor
β	Ratio of local incast senders to all senders
θ	Ratio of spine link speed to leaf link speed
B_s	Spine switch per-port buffer
B_l	Leaf switch per-port buffer
d	Incast degree
s	Average flow size in Bytes
U_l	Fraction of occupied buffer - leaf switch
r_s	Per-port residual buffer - spine switch
r_l	Per-port residual buffer - leaf switch
N	Maximum number of parallel senders
L	Number of additional links



(a) Superways/leaf-spine **(b) Leaf-spine**

Figure 4: Superways/leaf-spine and regular leaf-spine

buffer assigned to the port that connects to incast server. Therefore, we use formula 2 to calculate the maximum number of flows (shown by N) that can pass through the *leaf* switch towards the incast server without any packet drops.

$$\frac{\alpha \times N \times s}{B_l} = 1 \tag{2}$$

Because local incast senders already exceed the leaf switch's port buffer, some of them should use the additional link(s) that we will provide for the remote incast senders. Therefore, we need to update the ratio of local incast senders to the incast degree in order to consider some of them as remote incast senders. We use formula 3, to calculate the updated number of local incast senders that can use the current leaf switch's link capacity:

$$\beta = \frac{\lfloor N \rfloor}{d} \tag{3}$$

We use flooring to provide a safe margin for our calculations. At this point, we have the updated number of remote incast senders that should use the additional link(s). Next, we calculate the number of additional links (from incast server to spine switch) that is required by the remote incast senders (1-β):

$$L = \frac{\alpha \times ((1-\beta) \times d) \times s}{\theta \times B_s} \tag{4}$$

We consider the value of $\lceil L \rceil$ as the minimum number of additional links that is sufficient to avoid packet drops. We do not consider the existing link (i.e., from spine switch to leaf switch) as a usable link for incast senders. Instead, we leave this link for other servers that are located in the same rack. Note that due to different link speeds at spine and leaf switches, we need to add another factor (θ) to our formula. This factor is always equal or greater than one, as spine switches are equipped with faster NICs. The larger the θ the better because that particular spine port can support more incast applications.

Depending on the value of L, we might have a large value of residual buffer that could be used by other incast applications. For example, If $L = 1.2$, then we need to provide 2 additional links to absorb all incast packets; but, 80% of the second link's buffer capacity is still available for other incast applications. Therefore, we calculate the residual buffer of the spine link as follows:

$$r_s \leftarrow B_s \times (\lceil L \rceil - L) \tag{5}$$

At this point, we need to find an incast application that is able to use this residual buffer. This is the classic Knapsack problem: we pick the incast application with highest incast degree that fits in the residual buffer. We start from the top of the sorted list of incast degrees and calculate the value of $\frac{\alpha \times d \times s}{\theta \times r_s}$ for each incast application. Once this value becomes less than one, we assign the corresponding incast application to the current incast server. Note that all local and remote senders of the recently added incast application should use the additional link(s); because there is no more available buffer in the link from leaf switch to incast server.

We update the value of r_s after assigning the second incast application to the incast server (using formulas 4 and 5). If still any of incast applications can use the updated residual buffer, we use the same approach to install it on this incast server. On the other hand, if the residual buffer is not enough for any of incast applications, we move forward to the next server in this rack and repeat the whole process on a new incast server.

3.1.2 *U_l is less than one.*

If U_l is less than one, then it means current local senders do not overflow the leaf switch's buffer. In other words, depending on the size of residual buffer, one (or even more) of the remote incast senders might be able to use the leaf switch's port. We continue our calculations by measuring this residual buffer:

$$r_l = B_l \times (1 - U_l) \tag{6}$$

Now we check if all the remote incast senders fit in the residual buffer or not:

$$U_l = \frac{\alpha \times ((1-\beta) \times d) \times s}{r_l} \tag{7}$$

There are two possibilities depending on the value of U_l:

- U_l is still less than one: This means all local and remote incast senders can use the existing link connected from leaf switch to server, without any packet drops. Next, we calculate the residual buffer as follows:

$$r_l = \lceil U_l \rceil - U_l \tag{8}$$

  Lastly, we check the sorted list of incast degrees to find the incast application that can use the residual buffer. Again, we greedily pick the application with highest incast degree that fits in the residual buffer. Once found, we co-locate it with existing incast application on the current incast server. If the residual buffer is not enough for any of those incast applications, we leave it unassigned. This helps in having extra capacity when a burst of packets arrives at the switch port.

- U_l is larger than one: We need to provide additional links from incast server to a spine switch to absorb remote incast senders' packets. However, we should fully utilize the leaf switch's residual buffer first. Therefore, we repeat the previous calculations (formulas 2 and 3) to see how many of remote incast senders can use the link from leaf switch to incast server, and how many of them should use the additional links. Next, we use formulas 4 and 5 to find the optimal number of additional links, and then we calculate the residual buffer of these additional links that can be used by other incast applications:

If the residual buffer is enough to absorb incast packets of any of the incast applications (depending on their incast degree), we bin-pack it on the current incast server. However, if there is no more residual buffer, we move to the next server in the current rack and select it as our new incast server. Above calculations will continue until all incast applications are assigned to their incast servers.

If the number of incast applications is very large, we may need to proceed to the next rack (which is the second closest to spines rack) and pick a server as the new incast server.

3.2 Routing and load balancing

Superways creates a heterogeneous topology, and therefore, requires a load balancing method other than equal-cost load balancing schemes such as Equal Cost Multi Paths (ECMP). This is because there will be at least one shortcut route between an incast server and its senders (i.e., workers), which means the final cost of all possible routes is no longer the same. However, the routing mechanism will not change as we need incast senders to use the shortcut routes, which will be picked by existing routing protocols such as OSPF, by default (due to their lower cost compared to regular routes). Nevertheless, this holds for remote incast senders only and routing for local incast senders could be different. Below we provide the details of routing and load balancing for local and remote incast senders.

3.2.1 Load balancing for remote incast senders. Assuming a cost based routing protocol, packets originated from remote incast senders are always forwarded through the additional link(s) due to their lower cost compared to other paths. As an example, in Figure 4a, imagine S5 is sending a message to R1. S5's packets will be forwarded through the red link because of its lower cost compared to the normal paths (black links). If more than one additional link is provided, all the additional links can be bundled to a fat link, and therefore, still no further actions are required for load balancing remote senders' packets. In this case, an equal-cost load balancing method such as ECMP (or any forms of packet spraying) would work as well.
If our calculations require some of the remote servers to use the leftover bandwidth of the initial link (i.e., existing link between server and leaf switch), the network operator needs to set a static route on the spine switch to route the flows accordingly. However, the leftover capacity of the initial link is likely very limited (due to high incast degrees in modern datacenters) so that we do not expect that many remote incast senders use the initial link's capacity.

3.2.2 Load balancing for local incast senders. New studies on modern datacenters' traffic reveal that datacenter traffic is no longer rack local (i.e., only 13% of flows remain in the rack [52]). Thus, packets originated from local incast senders may or may not need further load balancing actions as it is unlikely to have too many local incast senders. In other words, if *some* of the local senders must use the additional link(s) (i.e., initial link's capacity is not enough even for absorbing local senders' packets - see section 3), we need to change the normal route to incast applications only for these servers. There are two solutions for this issue:

- **Source routing:** Incast senders can explicitly inform the switches about each packet's route. While source routing is easy to implement, it may increase latency (by adding processing delay), which is critical in datacenter networks.
- **Static routing:** We may need to add a couple of static routes to leaf switches to re-route the flows that must use the additional link(s). The static route should match the source and destination of each packet. Note that we need to check both the source and destination because the static route should affect *incast senders* only. Since most existing switches support OpenFlow, we can apply these simple configurations remotely on the SDN controller. As we mentioned earlier, we do not expect to have many local incast senders, and therefore, we may need to add only a handful of routes into the routing table of switches.

In the case that local incast senders do not need to use the extra capacity, no further action is required as existing link from leaf switch to server is perfectly handling the incoming traffic.

3.3 Topology management

In this section, we address physical considerations of implementing *Superways*, such as topology upgrade and wiring complexities.

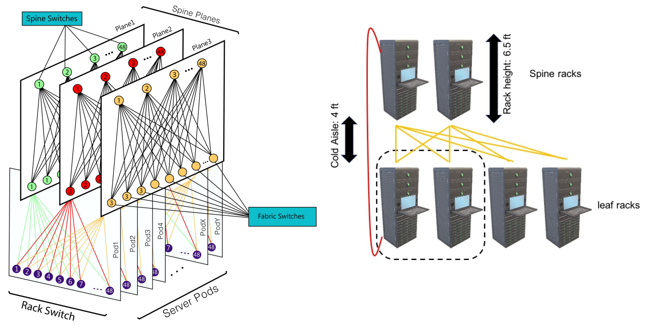
3.3.1 Topology upgrades. *Superways*' upgrading complexity is very similar to that of the underlying topology. The reason is *Superways* only targets a small number of servers in the topology, which does not lead to huge complexity in topology upgrades. In *Superways*, some specific servers are chosen to host bandwidth hungry applications, and then more links are provisioned for these servers proportional to their load and their fan-in degree. Thus, it would be a good idea to have a designated area (e.g., some racks or some pods in extreme cases) in the datacenter network, and place incast applications on servers of this area. As a result of this isolation, whenever a new incast application is placed in a datacenter network, it will be installed on one of these servers based on the calculations described in section 3.1.
There are two main reasons for colonizing incast applications: (1) incast applications will be able to use the potential extra capacity left by other incast applications, and (2) simplifying topology management. The latter is crucial as more incast applications may be added to the datacenter network over the course of time, and colonizing incast applications helps us in lowering the degree of heterogeneity of the topology. In other words, if there are many incast servers spread across the topology, while they have extra connections to spine switches, there will be many small islands in the topology that further complicates routing and load balancing, requires longer cables, and complicates topology management.

Therefore, we suggest to focus on one area of the topology and move every incast aggregator to this area. To see a clear picture of this process, refer to Figure 5a. This figure shows a schematic view of a typical tree topology deployed in Facebook [9]. We can pick as many racks as we need and then connect the servers inside those racks to the spine switches in one of the spine planes (e.g., spine plane 1). For wiring simplicity, we always pick closest racks to one of the spine planes. In Figure 5a, as an instance, servers in rack 1 of pod 1 and switches in spine plane 1 are good candidates to host incast applications as they are the closest to the spine plane 1.

Someone may note that colonizing some servers on a specific rack (or some racks) may not always be doable as there are some services that have placement restrictions. However, recent studies on datacenter networks show that servers do not tend to host rack local services [52], and therefore, most of them are not location sensitive. Other studies such as [35] confirm this fact that incast degrees can be much higher than number of ports in a leaf switch, and therefore, there must be many more worker servers outside of this rack.

Due to high growth rate of using online services (e.g., social networks), more incast applications are expected in modern datacenters. Therefore, if incast servers in the current designated rack cannot handle new incast applications, the datacenter operator can select another rack as the n-th designated rack and repeat the above-mentioned process for application placement. Since there are more than one spine switch in a spine plane, there is no problem with connecting to the next spine in the same plane, if there are no more available ports in the current spine switch. Note that the next spine switch should be the second closest spine switch to our designated rack. In some rare cases, spine switches in the closest spine plane may not have any available ports to connect to incast servers. In this case, we need to place extra spine switches with higher number of ports in that plane (i.e., spine rack) to connect to incast servers. Note that this spine switch should be connected to all leaf switches in that pod (see Figure 5a). Due to high number of ports in spine switches (i.e., 128 - 256), however, we expect to have enough free ports at spine switches given that there are small number of incast applications in a datacenter network.

3.3.2 Wiring length. *Superways* does not require long wires if closest racks to spine switches are selected to host incast applications. Figure 5b shows an example of *Superways*' architecture. Closest racks to spine switches are shown by dotted lines, which are the best candidates for hosting incast servers. Assuming a rack height of 6.5 ft and cold aisle width of 4 ft [24], the length of a wire that connects an incast server in a leaf rack to a spine switch in the spine rack is less than 15 ft, on average. The red link in Figure 5 shows this extra link. In the extreme case (i.e., providing an extra link from a server at the bottom of leaves' rack to spine switch at the top of spines' rack), we need a wire that is between 20 to 25 ft long, which is not a long cable given that it is still less than average cable length in production datacenter networks [5]. If we need to move to another rack due to increased number of incast servers, we still connect these servers to a new spine switch in the closest spine rack (e.g., second spine rack in Figure 5), and therefore, the average cable length in *Superways* will not increase.



(a) A schematic view of Facebook datacenter **(b) Wire length in a *Superways* datacenter**

Figure 5: *Superways* vs a typical datacenter topology

3.3.3 Link aggregation. If incast degrees are high, more additional links are required to absorb the burst of packets. Thus, we suggest to use a link aggregation protocol to create a fat link out of the additional links, and create a high bandwidth link that is able to process more packets at a given time. Link aggregation simplifies load balancing decisions and increases bandwidth. The latter is critically important because all co-located incast applications are not likely to receive their incast replies at the same time, and therefore, an incast aggregator can aggregate responses much faster due to high bandwidth at the spine switch. In other words, if multiple bursts arrive at a single incast application, while other incast applications installed on the same server are not active at this moment (i.e., they are not receiving a burst of packets), the overloaded incast application can use a big fraction of the fat link's capacity that was provisioned for *all* incast applications installed on that server. Therefore, extra capacity will be available for those incast applications that are actively receiving back-to-back incasts or experience higher incast degrees than usual. Having back-to-back bursts is quite likely as reported in [64]. This study reveals that incast interarrival time could be extremely short so that many of bursts arrive only 50 microseconds after the previous burst.

In summary, since not all incast applications receive their packets at the same time (i.e., all extra links are not 100% utilized at all times), considering *average* incast degree of each incast application would be sufficient to ensure that our calculations guarantee the enough number of additional links. Our simulation and testbed results confirm this assumption.

4 EXPERIMENTAL METHODOLOGY

In this section, we present our evaluation methodology, including details of the topology and the workload.

4.1 Topology

We use ns-3 [51] simulator to simulate a datacenter network. We implemented all previously discussed topologies including leaf-spine, Bcube [33], Jellyfish [58], and Subways [41] in our simulations. In these topologies, we connect 400 servers through 20 leaf (outer switches in Jellyfish) switches and 10 spine (inner switches in Jellyfish) switches. In tree topologies (e.g., leaf-spine, Subways, and Bcube), each of leaf switches have 20 downlinks to servers and 10 uplinks to spine switches. Also, the network fabric interconnects leaf and spine switches in a full mesh manner. In Jellyfish, however, we connect 20 outer switches to 10 inner switches as per [58]. The link speed between servers and leaf (outer) switches is 10 Gbps and links between leaf switches and spine (inner) switches are operating at 40 Gbps. The longest end-to-end Round-Trip Time (RTT) across the fabric is 80 microseconds, which is close to that of real datacenter networks. Also, congestion control mechanism in all topologies is DCTCP, and per port buffer size in leaf and spine switches is 240 KB (*Superways* ns3 code is publicly available at [16]).

4.2 Workload and Traffic

We use the workload reported in recent studies on Facebook's datacenter network [52, 64] to create a realistic traffic in our simulations. Our short flows' sizes are in the range of 1 KB to 10 KB, while nearly 70% of the flows are 1 KB in size. Also, as reported in [52], our long flows' sizes are in the range of 100 KB to 10 MB, while half of the long flows are 1 MB or below. All servers are spread across the network uniformly randomly, and 20 of the servers are incast aggregators (i.e., 5% of all servers). Also, we used a combination of short and long flows to produce incast. In our experiments, incast degree varies in the range of 48 to 96 among all incast applications, and incast inter-arrival times match the reported numbers in [64].

5 EVALUATION

In this section, we present the results of our evaluation using (1) ns-3 [51] simulations (Section 5.1–Section 5.2), (2) a real implementation using CloudLab [50] (Section 5.5), and (3) a cost analysis (Section 5.3). Our performance evaluation consists of 5 parts:

- **Flow Completion Time (FCT)**: In datacenter networks, 99^{th} percentile FCT is the most important metric for measuring short flows' performance [18]. Thus, we do not provide detailed results of median flow completion times. However, our experiments show that when *Superways* is implemented on top of the aforementioned topologies, it results in up to 21% improvement in median flow completion time, on average, for all loads.
- **Throughput**: We measure the average throughput of long flows (including those that participate in incast) when *Superways* is implemented on existing datacenter topologies to evaluate the effectiveness of *Superways* on throughput of long flows.
- **Cost analysis**: We analyze cost of implementing *Superways* and Subways in a small scale datacenter. We only compare to Subways because while is similar to *Superways*, it is the most recent proposal as well.
- **Why connecting to spines**: We will discuss the reasons that convinced us to connect the extra links to spine switches rather than leaf switches.
- **Real testbed**: Finally, we will verify our simulation results in CloudLab [50]. We show the performance of *Superways* in a real Web search workload, using Apache solr text indexing servers.

5.1 Flow Completion Time

Our experiments show that *Superways* significantly reduces the tail (99^{th} percentile) flow completion times of short flows, when it is implemented on top of other datacenter topologies. We implemented *Superways* on top of a leaf-spine topology to see how performance of leaf-spine will improve compared to its regular form. We see the results of this experiment in Figure 6. We show the average tail flow completion time of short flows (among various incast degrees) along Y-axis, versus load on X-axis. As we see in the figure, when *Superways* is implemented on top of a leaf-spine topology, it not only outperforms normal leaf-spine, but it significantly outperforms other complex and expensive datacenter topologies as well. Also, Figure 6 shows the performance of DT, which is the state-of-the-art shared buffer management technology in existing datacenter switches. As we see in the figure, when DT is implemented on all switches in a leaf-spine topology, it works well at lower loads only, which is not the case in many modern datacenter networks.

We implemented *Superways* on top of other datacenter topologies to see how their performance improves. The result of this experiment is shown in Figure 7. We show the reduction in tail flow completion time along Y-axis versus load on X-axis. As we see in this figure, by implementing *Superways* on top of Jellyfish (*Superways*/Jellyfish), flow completion time of short flows will reduce by 97%, on average, for all loads, compared to regular Jellyfish. Also, tail flow completion times in *Superways*/Bcube and *Superways*/Subways will reduce by 96% and 95% on average, for all loads, compared to their regular topology. Although Subways and Bcube both provide additional links for servers, their tail flow completion time reduction rate is close to that of leaf-spine and Jellyfish; which shows Bcube and Subways fail to solve incast problem.

5.2 Throughput

Superways forwards incast packets on a handful of dedicated links so that they do not collide with other non-incast flows. In other words, long flows will use those links that are not congested, and therefore, their throughput will not be affected by colliding with a burst of short flows.

In Figure 8, we show the average (among various incast degrees) throughput of long flows along Y-axis versus load on X-axis. As we see in the figure, by implementing *Superways* on top of simple leaf-spine, throughput of long flows improves by several orders of magnitude such that it achieves higher performance compared to all other topologies at higher loads (60%-80%).

Next, we studied the impact of *Superways* on throughput, when it is implemented with other datacenter topologies. The result of this experiment is shown in Figure 9. While *Superways* improves long flows throughput in Bcube and Subways by 9% and 20% respectively, leaf-spine topology benefits a lot from deploying *Superways*. Long flows' throughput in *Superways*/leaf-spine improves by about 55%, on average, for all loads. *Superways* improves long flows' throughput in Jellyfish by about 17% on average, for all loads, which shows the throughput efficiency of *Superways* in both tree and graph topologies.

5.3 Cost analysis

In this section, we study cost of implementing *Superways* and Subways on top of a leaf-spine topology. Similar to previous studies on cost comparisons (e.g., [47]), we use price quotes from different vendors to get a clear picture of the equipment cost. We used price quotes from vendors such as FS [10], Dell [8], and retailer websites

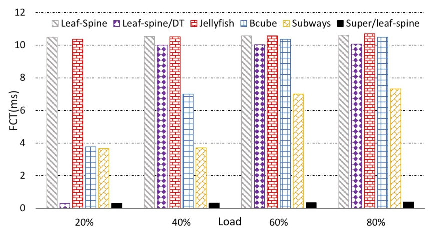


Figure 6: 99^{th} percentile flow completion times

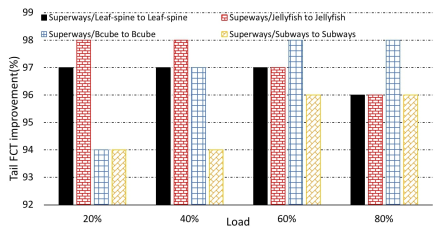


Figure 7: Reduction in 99^{th} percentile FCT after implementing *Superways*

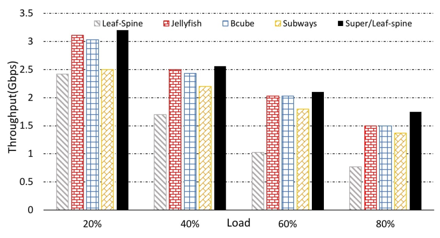


Figure 8: Long flows' average throughput

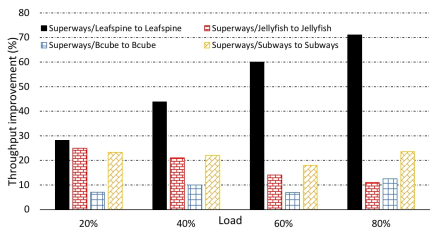


Figure 9: Speedup in long flows' throughput after implementing *Superways*

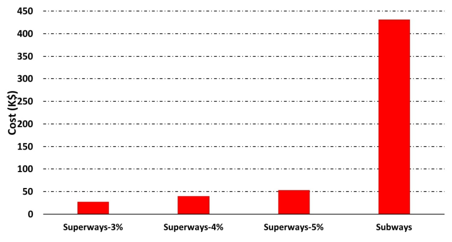


Figure 10: Cost comparison between Subways and *Superways*

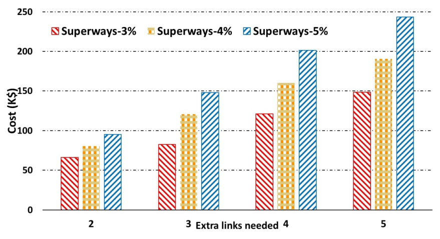


Figure 11: Cost of *Superways* with high incast degrees and shallow buffers

such as Amazon [1], Compsource [7], and Cablewholesale [4]. We consider vendors such as Dell [8], Juniper [14], Cisco [6], Brocade [3], and Huawei [12] for switches, and intel [13], HP [11], Rosewill [15], and Arista [2] for our NICs. Although prices vary among different vendors, we considered those equipment that satisfy the minimum requirements.

Our studies show that a 48 port 10 Gbps leaf switch would cost nearly \$5K, and a 32 port 40 Gbps spine switch costs nearly \$11K, on average. Also, we found the value of \$80 for a single-port 10 Gbps NIC. Bellow we use these prices to provide an exhaustive analysis on cost of implementation of *Superways* and Subways.

5.3.1 Cost analysis: Subways. We envision a datacenter network similar to the one in [41]. We consider a small datacenter network with 48 leaf switches and 24 spine switches that are connected in a leaf-spine topology. We assume each leaf switch is connected to 48 servers and each rack contains 48 servers plus the leaf switch that connects to them. We implement Subways type-1 (p=2), which requires an additional link from every server to its neighboring rack. Assuming a 42U rack with 6.5 ft height and cold aisle of 3.9 ft, on average, each server requires a 5.2 ft cable to connect to the neighbouring rack's leaf switch. Our price analysis on cable cost shows that a 10 ft long cat7 cable will cost about \$5, which shows that we need to pay \$11520 (i.e., $48 \times 48 \times \$5$) for the extra cables. Although Subways proposes a new way of wiring that decreases the cable cost, it does not significantly reduce the overall implementation cost, because cabling costs contribute to a small fraction of total cost of implementation. Subways needs more leaf switches to keep the over-subscription ratio of the network unchanged, and therefore, we need to pay $48 \times \$5,000 = \$240,000$ for extra leaf switches. Finally, assuming Subways type-1 with only one extra link per each server, we need to buy one more NIC for all servers, which costs $48 \times 48 \times \$80 = \$184320$. In total, we need to pay extra \$435,840 to implement Subways on a leaf-spine topology with 2304 servers.

5.3.2 Cost analysis: Superways. *Superways*' cost is highly dependent on number of incast applications, incast degrees, spines' link speed, and buffer sizes at leaf and spine switches. Bellow, we analyze *Superways*' cost in different scenarios when each incast application requires different number of extra links to absorb all incoming packets. Assuming that 3% of all servers host incast applications (70 out of 2304), and each incast application requires *one* more link to absorb all packets, we need to provide 70 extra links to connect incast servers to spine switches. Assuming 42U racks and a rectangular server room with two rows of racks and the rack containing spine switches located at the end of one of the rows, we need 50 ft long cables to connect the incast servers to spine switches (worst case scenario). Our price analysis on cable cost shows that a 50 ft cat7 cable will cost about \$20, which shows that we need to pay $70 \times \$20 = \$1400$ for extra cables. Since we need to add one extra NIC for those servers that hold incast applications, total cost of purchasing new NICs is $70 \times \$80 = \$5600$. In the extreme case that existing spine switches do not have enough available ports, we need to buy two more spine switches to connect to incast servers. The total cost of two extra spine switches is nearly \$22K. Therefore, if each incast server requires one extra link, the total implementation cost of *Superways* will be around \$28820.
Figure 10 shows the cost comparison between *Superways* and Subways type-1. This figure shows the total cost of *Superways* for 3 different cases when 3%, 4%, and 5% of all servers are incast applications (aggregators). Note that performance of all four schemes is the same because each server (each incast server in *Superways*) has only one extra link. As we see in Figure 10, even if 5% of all servers are hosting incast applications, which is a relatively large number, *Superways* is still about 9x cheaper than Subways. Also,

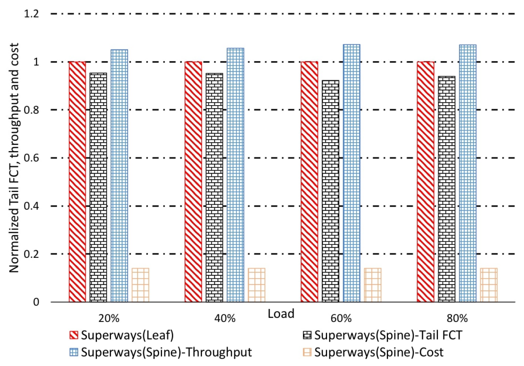


Figure 12: connecting extra links to leaf switches vs connecting extra links to spine switches.

Figure 11 shows different cases of *Superways* when more than one extra link is required per each incast server. This is the case when incast degrees are extremely high, link speeds of both leaf and spine switches is the same (e.g., all are operating at 10 Gbps, which is not high), and buffers are shallow. Although having too many links for each incast server is a rare case, even with high number of extra links, *Superways* still costs much less than Subways, which is due to its heterogeneous design that focuses on one area of the network only. In other words, *Superways* might need more than one link for each incast server depending on the burstiness of traffic, but, since it does not need too many extra switches and it only requires *some* servers to have extra links, the total cost of implementation remains low compared to other topologies with redundant links.

5.4 Why connect to spine switches?

There are both performance and cost related factors involved in *Superways*' design. By connecting to spine switches, which are more powerful in terms of processing speed and bandwidth, the amount of time that incast packets stay in a port's queue will reduce compared to leaf switches, and therefore, their average and tail FCT improves. Moreover, by providing such dedicated links from spine switches to incast applications, long flows will no longer collide with a batch of short incast flows on the spine switches, and therefore, they will not suffer delay specially when packet prioritization is enabled in the network (i.e, those short incast flows will always get the highest priority). Finally, due to large difference in link bandwidth of spine and leaf switches, the number of links, number of server side NICs, and total number of required switches will significantly reduce, which significantly reduces the overall cost of implementation if we connect the extra links to spine servers.

Figure 12 shows a comprehensive picture of comparing the overall performance of *Superways* when we connect the extra links to spine switches versus connecting to leaf switches. As we see from the figure, while connecting to spine switches improves both throughput and tail FCT by up to 8%, it reduces the cost of implementation by about 7x, which is a considerable amounts of money in medium to large scale datacenters. Thus, connecting the extra links to spine switches would be the best option as it improves performance and reduces cost of implementation, while adding a little complexity to the routing mechanism (i.e., we can easily update the routing table on the SDN controller through OpenFlow messages).

Table 2: Improvements in flow completion time and throughput - Real testbed and simulations

Topology	Testbed results		Simulation results	
	Tail FCT	Throughput	Tail FCT	Throughput
Super/leaf-spine	85.12%	38.4%	96.75%	50.7%
Super/Jellyfish	81.4%	16.3%	97.25%	17.1%
Super/Bcube	83.7%	7.1%	96.75%	11.5%
Super/Subways	80.04%	15.5%	95%	20.5%

5.5 Real testbed

We implemented a leaf-spine topology in CloudLab [50] to validate our simulation results and to evaluate metrics such as CPU usage and network I/O utilization, which cannot be done on a simulator. Similar to Figure 3(b), Our real testbed emulates 16 servers, 4 leaf switches, and 2 spine switches. The over-subscription ratio in the leaf-spine topology is 2:1, and the leaf switches are connected to spine switches in a full mesh manner. Also, using the same number of servers and switches, we implemented Bcube, Subways, and Jellyfish to validate the efficiency of *Superways* when it is implemented on top of these topologies. Bcube, Subways, and Jellyfish are implemented as per [33], [41], and [58]. Our servers run Ubuntu 16.04 (kernel version 3.3) and our switches run Open vSwitch version 2.31. We rate limit all switches to 1 Mbps, and, also, we set the transmit queue size of spine and leaf switches to 10KB.

To create a realistic workload, we used the reported numbers in [52] and [64]. Our short and long flows are 4 KB and 1 MB in size, and our incasts are a mix of short and long flows. While we used iperf3 to produce all-to-all traffic between random servers, we installed *Apache Solr* [59] version 8.2.0 on certain servers to emulate a real Web search traffic (including incast). These Apache Solr servers generate light weight queries that ask other servers about a specific event that they already saved in their database. We created incast traffic so that all incast senders send synchronous replies to the Apache Solr servers.

5.5.1 Flow completion time and throughput. We provisioned three incast applications (running Apache Solr server v8.2.0) with incast degrees of 9, 3, and 3 that are co-located on one of our servers according to calculations described in section 3. We repeated the experiments 10 times to accurately measure the performance of *Superways* in all topologies. Table 2 shows the improvement in 99^{th} and 50^{th} percentile flow completion times of short flows when *Superways* is implemented on top of other topologies. As we see from the table, our testbed results are close to simulation outputs, which endorses the efficiency of *Superways* in improving tail FCT of short flows. We measured the throughput of long flows as well. Since long flows will no longer collide with bursts of short flows at the network core, their throughput improves by up to 38.4%, as we see in table 2.

5.5.2 CPU utilization vs bandwidth utilization. Co-locating more than one incast application on a single server may arise concerns about CPU usage of that particular server. We designed an experiment to see how CPU utilization varies among those servers that participate in all-to-all traffic, and that of incast aggregators. Figure 13 shows the result of this experiment. The red line shows the

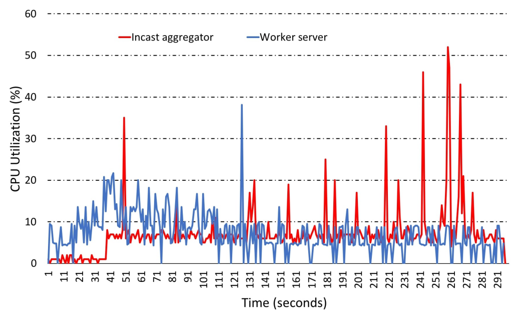


Figure 13: CPU utilization of an incast aggregator compared to a worker server

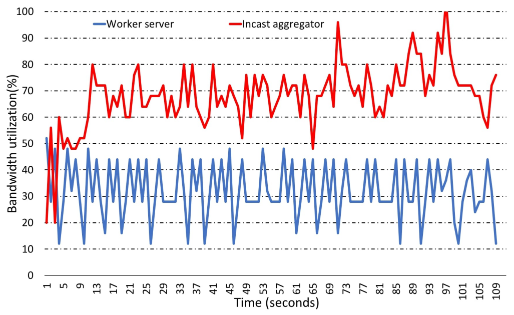


Figure 14: Bandwidth utilization of an incast aggregator compared to a worker server

CPU utilization of a server with Apache Solr V8.2.0 installed on it (incast application) and the blue line shows CPU utilization of a random server that transmits both short and long flows using iperf3. As we see in the figure, CPU utilization of both servers is almost the same, so that the average CPU utilization of the incast aggregator is 8.19%, while this value is 7.08% for the other server. Therefore, co-locating incast applications on a single server will not over-utilize the CPU as incast applications do not run compute-intensive tasks. On the other hand, we claimed throughout the paper that incast aggregators require more bandwidth compared to other servers due to their high number of incoming packets. To verify this assumption, we measured the bandwidth utilization for an incast aggregator and a random server during a short period of time. Figure 14 shows this comparison. As we see in this figure, the incast aggregator utilizes bandwidth much more than the other server so that while average bandwidth utilization is 17% for an ordinary server, this value is 68% for an incast aggregator. Thus, incast applications are indeed network bound and providing more bandwidth for them is necessary to guarantee the highest performance.

6 RELATED WORK

There is a large body of related work in datacenter networks that directly or indirectly deal with incast. D3 [62], PDQ [36], and D2TCP [60] optimize the completion times of short flows using either proactive and reactive rate allocation. They all rely on flow information such prior knowledge about flow size and flow deadline, that are not easy to achieve in real datacenters. DCTCP [18] is a pioneering work that improves the accuracy and response times of traditional TCP and ECN [48]. It throttles its sending rate based on the fraction of congested packets.

Homa [45], NDP [35], and ExpressPass [29] employ receiver-driven techniques, and therefore, address incasts. All these techniques require at least one RTT to deal with incasts. Recall that most incasts last shorter than one RTT [64]. While some of these proposals would allow the receiver to schedule the rest of packets, they do not control packet scheduling in the first RTT; the first window of packets are always sent at line rate. Recent measurement studies show that the average flow size of workloads such as Web Search is of the order of one kilobyte [52, 64], which would easily fit within 1 RTT in modern datacenters (e.g., a 10 Gbps link with an RTT of 100 μs would send more than 100 KB of data in 1 RTT). Therefore, aforementioned schemes always miss the sent packets in the first RTT. Note that most datacenter flows finish in only one RTT.

Other approaches detect incasts at the routers [21, 54] by requiring end-host to set rates. However, they suffer from false-positives and false-negatives in incast detection. Pacing [19, 43, 49] packets at the sender alleviates incasts to some extent but do not completely avoid the rate mismatch at the receiver due to incasts. Homa [45], another receiver-driven protocol, sends the first part of a flow without any restrictions but the latter parts of the flow must be explicitly scheduled by the receiver. Similar to NDP, Homa would suffer from incasts during the first RTT because most flows are short and would fit within the first RTT. Therefore, the congestion control schemes do not effectively deal with incasts.

There are a number of recent papers on topology (e.g., Subways [41], Bcube[33], Portland [46], VL2 [32], and fat-tree [17]). We present *Superways*, which is complementary to all these topologies, in that providing extra links only for those servers that receive a bulk of messages from multiple senders simultaneously. We elaborately discuss Bcube [33], Jellyfish [58], and Subways [41] in the body of our paper. DCell [34] adds multiple network interfaces to servers to use them as switches to forward packets. DCell, however, requires a large number of cables, which limits its scalability to large datacenters. Similarly, VL2 [32] leverages servers for packet forwarding, and therefore suffers from CPU and bandwidth overheads. A recently proposed method called Larry [27] uses idle links on neighbor ToR switches to transmit packets to core switches, in case of uplink congestion in the ToR switches. However, Larry is not able to handle incast because it focuses on rack uplink congestions only.

7 CONCLUSION

In this paper, we presented *Superways*, a heterogeneous datacenter topology that is designed for incast-heavy workloads. Based on our key observation that in most incast applications a few aggregator processes are bottlenecked by receiver bandwidth, our proposal provides additional bandwidth (links) for a small subset of nodes. Further, we proposed a heuristic for scheduling critical aggregator processes in our heterogeneous topology. Our insight for the scheduler is to greedily find the optimal number of extra links by measuring the residual buffer of the extra links after bin-packing incast aggregators. By connecting incast aggregators to the spine switches, *Superways* decreases the packet drop rate and improves the tail flow completion times of incast flows. When combined with existing load balancing schemes, *Superways* distributes the incast flows among multiple receive links, avoiding flow collision between incast and non-incast flows. Furthermore, *Superways* improves fault tolerance and throughput while reducing the total cost of implementation compared to other schemes such as Subways. As incast-heavy applications become more prevalent, heterogeneous topologies and their associated job scheduling strategies such as our proposal would become increasingly important.

REFERENCES

[1] Amazon. http://www.Amazon.com.
[2] Arista Networks. https://www.arista.com.
[3] Brocade Inc. https://www.brocade.com.
[4] Cablewholesale. https://www.cablewholesale.com.
[5] Cablinginstall. https://www.cablinginstall.com/data-center/article/16468527/the-data-center-evolution-how-to-overcome-its-cabling-challenges.
[6] Cisco Systems. https://www.cisco.com.
[7] Compsource. https://www.compsource.com.
[8] Dell. http://www.dell.com.
[9] fb. https://www.techrepublic.com/article/facebook-fabric-an-innovative-network-topology-for-data-centers/.
[10] FS.com. http://www.fs.com.
[11] HP. https://www.hp.com.
[12] Huawei Technologies. https://www.huawei.com.
[13] intel. https://www.intel.com.
[14] Juniper Networks. https://www.juniper.net.
[15] Rosewill. https://www.rosewill.com.
[16] Superways_ns3_code. https://github.com/hrezae2/Superways-WWW-21/.
[17] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture *(SIGCOMM '08)*.
[18] Mohammad Alizadeh et al. 2010. Data center TCP (DCTCP) *(SIGCOMM '10)*.
[19] Mohammad Alizadeh et al. 2012. Less is more: trading a little bandwidth for ultra-low latency in the data center *(USENIX NSDI)*.
[20] Mohammad Alizadeh et al. 2013. pFabric: Minimal Near-optimal Datacenter Transport *(SIGCOMM '13)*. ACM.
[21] Hamidreza Almasi, Hamed Rezaei, Muhammad Usama Chaudhry, and Balajee Vamanan. 2019. Pulser: Fast Congestion Response Using Explicit Incast Notifications for Datacenter Networks *(IEEE LANMAN'19)*. 1–6.
[22] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing Router Buffers *(SIGCOMM '04)*.
[23] Arista 2020. Arista 7050QX Series 10/40G Data Center Switches. https://www.arista.com/assets/data/pdf/Datasheets/7050QX-32_32S_Datasheet.pdf.
[24] Telecommunications Industry Association. 2012. TIA standard ANSI/TIA-942-A, data center cabling standard amended.
[25] Wei Bai, Kai Chen, Shuihai Hu, Kun Tan, and Yongqiang Xiong. 2017. Congestion Control for High-Speed Extremely Shallow-Buffered Datacenter Networks *(APNet'17)*.
[26] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, omega, and kubernetes. *Queue* 14, 1 (2016), 70–93.
[27] Andromachi Chatzieleftheriou, Sergey Legtchenko, Hugh Williams, and Antony Rowstron. 2018. Larry: Practical network reconfigurability in the data center *(USENIX NSDI'18)*. 141–156.
[28] Yanpei Chen et al. 2009. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*. ACM.
[29] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of SIGCOMM*. 239–252.
[30] Abhijit K Choudhury and Ellen L Hahne. 1998. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions On Networking* 6, 2 (1998), 130–140.
[31] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013).
[32] Albert Greenberg et al. 2009. VL2: a scalable and flexible data center network *(SIGCOMM '09, 4)*. ACM.
[33] Chuanxiong Guo et al. 2009. BCube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Computer Communication Review* 39, 4 (2009), 63–74.
[34] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. 2008. Dcell: a scalable and fault-tolerant network structure for data centers. In *ACM SIGCOMM Computer Communication Review*. ACM.
[35] Mark Handley et al. 2017. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM.
[36] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing flows quickly with preemptive scheduling *(ACM SIGCOMM'12)*.
[37] Sundar Iyer, Ramana Rao Kompella, and Nick McKeown. 2008. Designing Packet Buffers for Router Linecards. *IEEE/ACM Trans. Netw.* 16, 3 (June 2008), 705–717. https://doi.org/10.1109/TNET.2008.923720
[38] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. 2009. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings of IMC*. 202–208.
[39] Jitendra Kumar and Ashutosh Kumar Singh. 2020. Cloud datacenter workload estimation using error preventive time series forecasting models. *Cluster Computing* 23, 2 (2020), 1363–1379.
[40] Charles E. Leiserson. 1985. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* 34, 10 (Oct. 1985), 892–901.
[41] Vincent Liu et al. 2015. Subways: A case for redundant, inexpensive data center edge links *(CoNEXT'15)*. ACM.
[42] S. H. Low, F. Paganini, Jiantao Wang, S. Adlakha, and J. C. Doyle. 2002. Dynamics of TCP/RED and a scalable control. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 1.
[43] Radhika Mittal et al. 2015. TIMELY: RTT-based Congestion Control for the Datacenter *(SIGCOMM '15)*. ACM.
[44] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal Packet Scheduling. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. 501–521.
[45] Behnam Montazeri et al. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities *(SIGCOMM '18)*. ACM, 221–235.
[46] Niranjan Mysore et al. [n.d.]. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM Computer Communication Review*, Vol. 39.
[47] Lucian Popa, Sylvia Ratnasamy, Gianluca Iannaccone, Arvind Krishnamurthy, and Ion Stoica. 2010. A cost comparison of datacenter network architectures. In *Proceedings of the 6th International Conference*. ACM, 16.
[48] Kadangode Ramakrishnan, Sally Floyd, and David Black. 2001. *The addition of explicit congestion notification (ECN) to IP*. Technical Report.
[49] Hamed Rezaei et al. 2019. ICON: Incast Congestion Control using Packet Pacing in Datacenter Networks *(IEEE COMSNETS)*. 125–132.
[50] Robert Ricci and Eric Eide. 2014. The CloudLab Team. *Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. USENIX* 39, 6 (2014).
[51] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.
[52] Arjun Roy et al. 2015. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*. ACM.
[53] Danfeng Shan, Wanchun Jiang, and Fengyuan Ren. 2015. Absorbing micro-burst traffic by enhancing dynamic threshold policy of data center switches. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 118–126.
[54] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo. 2018. Micro-Burst in Data Centers: Observations, Analysis, and Mitigations. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. 88–98.
[55] Arjun Singh et al. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of SIGCOMM*.
[56] Ankit Singla et al. 2010. Proteus: a topology malleable data center network *(HotNets'10)*. ACM.
[57] Ankit Singla, P Brighten Godfrey, and Alexandra Kolla. 2014. High throughput data center topology design *(NSDI'14)*.
[58] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. 2012. Jellyfish: Networking data centers randomly *(NSDI'12)*.
[59] solr 2020. Apache Solr. https://lucene.apache.org/solr/.
[60] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. 2012. Deadline-aware Datacenter TCP (D2TCP) *(SIGCOMM '12)*.
[61] Andras Varga. 2010. OMNeT++. In *Modeling and tools for network simulation*. Springer, 35–59.
[62] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: meeting deadlines in datacenter networks *(SIGCOMM '11)*. ACM, New York, NY, USA.
[63] Kyriakos Zarifis et al. 2014. DIBS: Just-in-time Congestion Mitigation for Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*.
[64] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution measurement of data center microbursts *(IMC'17)*. ACM.