Horus: Granular In-Network Task Scheduler for Cloud Datacenters

Parham Yassini*[1], Khaled Diab*[2], Saeed Zangeneh[1] and Mohamed Hefeeda[1]

[1]*School of Computing Science, Simon Fraser University, Burnaby, BC, Canada*
[2]*Hewlett Packard Labs, United States*

Abstract

Short-lived tasks are prevalent in modern interactive datacenter applications. However, designing schedulers to assign these tasks to workers distributed across the whole datacenter is challenging, because such schedulers need to make decisions at a microsecond scale, achieve high throughput, and minimize the tail response time. Current task schedulers in the literature are limited to individual racks. We present Horus, a new in-network task scheduler for short tasks that operates at the datacenter scale. Horus efficiently tracks and distributes the worker state among switches, which enables it to schedule tasks in parallel at line rate while optimizing the scheduling quality. We propose a new distributed task scheduling policy that minimizes the state and communication overheads, handles dynamic loads, and does not buffer tasks in switches. We compare Horus against the state-of-the-art in-network scheduler in a testbed with programmable switches as well as using simulations of datacenters with more than 27K hosts and thousands of switches handling diverse and dynamic workloads. Our results show that Horus efficiently scales to large datacenters, and it substantially outperforms the state-of-the-art across all performance metrics, including tail response time and throughput.

1 Introduction

The slowdown of Moore's law and the end of Dennard scaling have changed how cloud datacenters deploy and manage their hardware resources and software services. Instead of continually increasing the frequency of CPU cores, microprocessor vendors have been shipping more cores per processor with only slight frequency increases. As a result, the available number of cores in datacenters has been steadily and substantially increasing over the last several years [30]. From the software perspective, numerous interactive and user-facing datacenter applications have been deployed. Examples of such latency-sensitive applications include key-value stores [11–13], multimedia applications [16, 34], distributed interactive analytics [49, 56], network function virtualization [56], and web search [9, 21]. To take advantage of the availability of many cores and reduce deployment costs, designers of these large-scale applications have recently started to embrace various practices such as micro-services and function-as-a-service.

The emerging hardware trends and the requirements of recent large-scale applications have increased the demands for fine-grain management of the datacenter computing resources. This is sometimes referred to as the *granular computing* paradigm [52]. In this paradigm, many applications are decomposed into large numbers of *short-lived* tasks, which are executed in parallel on 100's–1000's of cores that potentially *span multiple server racks*. Each task typically has a tight *response time*, in the order of 10's–100's of microseconds [20, 52]. And since the performance of applications is affected by their slowest tasks, granular computing platforms strive to minimize the task *tail response time* [20]. Granular computing platforms are also expected to support *high scheduling throughput* as the number of concurrent tenants and their application demands are rapidly growing [69].

Granular computing can be viewed as the generalization of recent initiatives from academia and industry for offering more flexible, finer-grain, cost-effective, and shorter latency computing infrastructures. For example, Amazon Lambda [2] handles task execution times in the order of 100's of milliseconds using the Firecracker microVMs [14]. Apache OpenWhisk [1] offers a serverless framework to seamlessly execute functions while handling the provisioning of the underlying computing resources. Efforts from Microsoft [68] and others [45, 70] have introduced mechanisms to reduce the cost of cold-starts in serverless frameworks to support low-latency applications. Furthermore, recent operating system schedulers, e.g., [43, 64], offer support for microsecond-scale tasks within individual servers. Granular computing aims at pushing the boundaries even further, by efficiently supporting microsecond tasks at a *datacenter scale*. This paper contributes to the realization of granular computing.

As illustrated in Figure 1, datacenter operators deploy mul-

*Both authors contributed equally to this work.

tiple software components to manage applications and computing resources [1, 71, 73], including a resource manager, worker pools, and task schedulers. The resource manager allocates a worker pool for each application according to its required level of fault tolerance and performance using mechanisms such as [43, 68, 70]. Then, a task scheduler assigns each submitted task of an application to a worker from its worker pool. Resource managers, such as YARN [72] and Mesos [39], decouple resource allocation from task scheduling. This enables deploying multiple task schedulers targeting different application needs. This paper presents a granular task scheduler designed for latency-sensitive applications.

Designing a granular task scheduler is, however, challenging because such a scheduler needs to make decisions at a microsecond scale, achieve high throughput, and minimize the task tail response time. Minimizing the tail response time requires balancing the load across workers, which is difficult to achieve because of the substantial diversity in the task execution times and the scale of modern datacenter applications that could have thousands of workers distributed across many racks. Further, since tasks are short-lived, the load on workers is highly dynamic. Thus, naively tracking the load on workers could result in substantial communication, processing, and memory overheads on the scheduler.

Current *software* schedulers, e.g., Borg [73], Twine [71], and Atoll [70], introduce significant network and processing delays. These delays are sometimes larger than the task execution time itself, which makes software schedulers unsuitable for short-lived tasks. In addition, scheduling granular workloads requires a substantial amount of computing resources [66, 73], which is difficult to realize using traditional application-layer schedulers. For example, consider a system with 20K tasks and a task mean execution time of 100 μs. The system would need to make 200M scheduling decisions per second and handle around the same number of packets for processing the state update messages. This scale of throughput is not possible to achieve using software schedulers [66].

In-network schedulers, on the other hand, schedule tasks in the data plane as packets carrying these tasks pass through switches. Thus, they significantly reduce the scheduling latency and, in turn, the response time of tasks. However, recent in-network schedulers, e.g., [48, 50, 78], can only schedule tasks within individual racks. Therefore, they cannot meet the growing demand of large-scale applications that require executing thousands of tasks across multiple racks.

To support current latency-sensitive applications and future granular computing platforms, we propose **Horus**, the *first* datacenter-wide in-network scheduling system in the literature, to the best of our knowledge. One of our key insights in designing Horus is that scheduling operations should run at different time scales for efficiency and scalability. For example, assigning a task to a worker should be done at a microsecond scale, whereas tracking the load on workers can occur at a millisecond scale. Leveraging this insight, we divide the

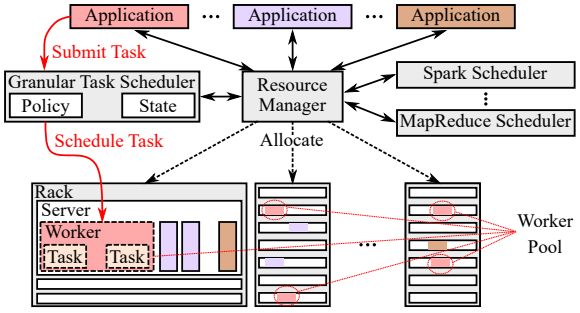


Fig. 1: Overview of resource management in datacenters.

operations of Horus into two components. The first assigns tasks to workers, and the second tracks, aggregates, and maintains the load on workers. Horus *offloads* both components to switches in the network and optimizes them independently, which enables efficient scheduling of granular tasks.

The contributions of this paper are as follows.

- We propose an in-network task scheduling architecture for latency-sensitive datacenter applications, in §3.
- We present a new scheduling policy that does not queue tasks at switches and runs at line rate, in §3.3.
- We design multiple data structures to realize the proposed policy in modern programmable switches, which have a restricted programming model and limited memory resources.
- We propose efficient mechanisms to distribute the load information among schedulers, which maintain the freshness of load values and minimize overheads on switches, in §3.4.
- We implement Horus in a testbed using a Tofino switch and compare it against the state-of-art in-network scheduler, which is RackSched [78], in §4. In single-rack settings, our results show that Horus reduces the tail response time by up to 75% and increases the throughput by up to 1.9X compared to RackSched, for the considered realistic workloads. Since RackSched does not support multiple racks, we show that Horus substantially outperforms two natural extensions of RackSched in multi-rack settings.
- We also conduct large-scale simulations for datacenters with more than 27K hosts and thousands of switches handling diverse and dynamic workloads, in §5. Our results show that Horus outperforms RackSched and its extensions across all performance metrics. We also show the robustness of Horus against failures, packet losses, and link delays.

Due to space limitations, some details and evaluation results are presented in the **Appendix**.

2 Background and Related Work

2.1 Task Scheduling in Datacenters

Granular applications, e.g., key-value stores and microservices, create many short-lived tasks with *diverse* execution

times, ranging from tens of microseconds to hundreds of milliseconds or even longer. These tasks need to be assigned to workers for execution. Recent intra-server schedulers, e.g., Shinjuku [43] and ZygOS [64], support microsecond tasks. These task schedulers are, however, limited to single servers.

A natural question is then: can we use load balancers with server schedulers to scale beyond individual servers? Zhu et al. [78] showed that this policy is ineffective and may yield long tail response time. This is because most load balancers, e.g., [19, 33, 57], typically make their decisions based on hashing various fields in the packets. Zhu et al. [78] proposed RackSched to extend task scheduling to the rack level, where the top-of-rack (ToR) switch approximates the load on servers within its rack and assigns tasks to them accordingly.

Scaling task scheduling beyond single racks is an important and challenging research problem. There are many practical scenarios where applications require and/or benefit from execution on cores across racks. For example, running applications across racks in different fault domains improves their fault tolerance and availability [4,7,22]. This is especially critical for latency-sensitive applications since most of them are user-facing. A recent study from Facebook [6] indicates that the traffic of many latency-sensitive applications is mostly not rack-local. In addition, in public datacenters, it is not uncommon that tenants' VMs are placed on different racks due to unavailable resources at the time or for improved fault tolerance [40]. Therefore, there is a need to run tasks across racks in the datacenter. However, simple extensions of rack-level schedulers that use load balancers to distribute tasks to racks and then to servers may lead to long tail response time for the same reason mentioned above: load balancers are oblivious to the current queue lengths of workers, which could be impacted by the diversity in the task execution times.

To demonstrate the limitations of using the state-of-the-art approach, which is RackSched [78], for scheduling tasks across racks, we conduct simulations with representative workloads and datacenter configurations similar to prior works [43,54,67,78]; the details of our simulations are given in §5.1. Briefly, we simulate a tree-based datacenter with 27,648 servers, each having 32 cores, and we consider large workloads and diverse task distribution times and arrival patterns. We implemented the scheduling policy of RackSched, which we refer to as RS. We complemented RS with a datacenter load balancer that uniformly (at random) distributes tasks to racks. Tasks are then scheduled to servers within racks using RS. We refer to this scheduling system as RS-LB.

In addition, to show the potential performance gains, we simulate a *global* version of the Join Shortest Queue (JSQ) policy. JSQ tracks queue lengths at individual servers, and it schedules tasks to the server with the shortest queue. As analyzed in [79], JSQ produces optimal results across different performance metrics, e.g., waiting time and throughput, and for tasks with low- and high-dispersion execution times. We simulate an *ideal/theoretical* version of JSQ that immediately updates queue lengths to show the performance bounds. The results of our simulation are presented in Figure 2 for two representative task execution time distributions: Exponential and Bimodal. The figure plots the tail (99%) response time as the normalized workload increases for different task scheduling systems. The results reveal the substantial performance gap between JSQ and RS-LB. For example, in Figure 2a, the tail response time increases rapidly as the normalized load exceeds 50% when RS-LB is used, whereas it stays low for JSQ even for a load around 90%.

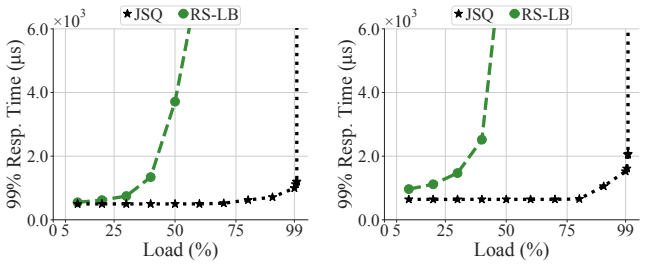


Fig. 2: Limitations of current rack-level task scheduling systems when scaled to the whole datacenter.

Although JSQ theoretically provides optimal results, it is not possible to implement in practice for large-scale datacenters, especially for microsecond tasks. This is because JSQ requires knowing the queue lengths at all servers, which takes time to either probe servers (one RTT) or wait for servers to send their updates (which may arrive asynchronously and/or delayed/aggregated). Thus, by the time the scheduler determines the server with the least load, the situation might have already changed. In addition, JSQ and similar global policies may cause task *herding*, which occurs because once a server is reported to have the least load, the scheduler may keep sending tasks to it until the server sends an update. Since updates may take a relatively long time compared to task execution times, a burst of too many tasks could have already been sent to the server, which leads to severe load imbalance and long response time. Finally, implementing JSQ would require extensive computation resources at the scale of a datacenter.

2.2 Related Work

Software Schedulers. Traditional schedulers are designed as software processes that run on one or multiple servers [37, 62, 73]. These schedulers focus on service times in the range of seconds to hours, and they can make complex decisions and support a wide range of resource allocation policies. To scale up and support higher throughput, distributed schedulers have been proposed [28, 29, 31, 46, 60]. The shortest task service time supported by distributed schedulers is still in the order of hundreds of milliseconds [31, 52, 60]. This is because of the relatively long time such schedulers take to

allocate tasks to servers. Sparrow [60] and Eagle [28], for example, maintain queues of submitted tasks at schedulers. Each scheduler then pushes reservation probes to randomly-selected workers. When a worker becomes idle, it pulls the next reserved task from that scheduler's queue, introducing a scheduling delay of at least two RTTs.

In contrast, Horus is designed for granular tasks with execution times in the order of tens of μs, which can be smaller than a single RTT [20]. Horus is also designed to support millions of scheduling decisions per second. In addition, multiple prior works, e.g., YARN [72] and Mesos [39], decouple resource allocation from task scheduling. Horus can complement such works by offering fine-grain task scheduling on the allocated resources for large-scale datacenter applications.

In-network Computing. Emerging programmable switches enable offloading of various operations and functions to the network to achieve high throughput and low latency for datacenter applications, such as caching [41], data aggregation [65], concurrency control [53], and lock management [77].

Similarly, in-network task scheduling has been considered before [48, 50, 78]. RackSched and prior in-network schedulers, however, support only single racks. In contrast, Horus scales to multiple racks across the whole datacenter network. RackSched [78] was shown to outperform prior works, and it is considered state-of-the-art. Thus, we compare a simple version of Horus against RackSched in single racks. We also extend RackSched to support scheduling across multiple racks and compare Horus against these extensions.

3 Proposed In-Network Scheduling

In this section, we first summarize the principles that guided the design of Horus. Then, we describe the proposed in-network task scheduling approach. This is followed by describing our efficient methods for distributing state among various components of Horus. Finally, we describe various deployment options for Horus. Due to space limitations, we present some details in the **Appendix**, including handling failures and packet losses (§A.3), supporting multi-packet tasks (§A.4), overhead analysis (§A.5), and pseudo code (§A.6).

3.1 Design Principles

The design of Horus is based on the following principles:

- ***P1: Load-aware Scheduling***. The load on workers in datacenters is subject to spatial and temporal variations due to resource allocation policies, application requirements, and the seasonality of workloads [71, 73]. And as shown in prior works, e.g., [78], and by our simulations in §2.1, not considering the actual worker load in the scheduling decisions may lead to long tail response times. We propose a *zero-queue* scheduling system that efficiently tracks the load of workers, minimizes the task tail response time, and avoids task herding. By not buffering tasks in switches, the memory requirements become *independent* of the task rate, which helps Horus to scale.
- ***P2: Lazy State Update***. Horus makes scheduling decisions based on the maintained state without queuing tasks. Thus, updating this state is important to reflect the latest changes. This, however, may increase the communication overhead and limit scalability. Our idea is that a switch may not need to immediately update its state *if* it can make accurate decisions using its current state. Our approach identifies when an update is needed by calculating a *drift* between actual load values and the load information available at schedulers, and it only updates the state of a scheduler when the drift may negatively impact the scheduling quality.
- ***P3: Localized State***. Horus avoids the complexity of replicating state across all switches in the datacenter by logically grouping the distributed schedulers and maintaining the state within each group. This allows each scheduler to update its view of a subset of workers without querying other schedulers. Localization of state enables Horus to further reduce the overheads on switches, achieve high throughput, and handle failures efficiently.

3.2 Overview and Workflow

Overview. Horus is a *distributed*, in-network, granular task scheduling architecture designed for datacenters. It can be viewed as one of the components in the software suite managing computing resources in datacenters, as illustrated in Figure 1. For example, Horus can be integrated with existing platforms such as OpenWhisk [1], and it can coexist with schedulers of long-lived tasks such as Borg [73].

As shown in Figure 3, Horus consists of a set of schedulers, a centralized controller, an agent per server, and APIs.

Schedulers in Horus are distributed to handle high task rates and tolerate failures. Horus decomposes task scheduling into two components. The first maintains and aggregates the load information of workers, whereas the second executes the scheduling policy to assign tasks to workers using this maintained information. Both components run in the data plane of the switches. Schedulers do not require specific network topology, and thus, Horus can easily be deployed on different datacenter networks. For clarity of the presentation, however, we focus on the widely-deployed leaf-spine topology [15, 42], which is shown in Figure 3. In this case, Horus schedulers run as data plane programs on leaf and spine switches; no schedulers run on core switches. Leaf schedulers track and use the load information about workers in their racks (**P1**), and they aggregate and efficiently distribute this information to spine schedulers (**P2**).

The ***centralized controller*** realizes various functions, such as addressing and handling failures. It assigns a fixed ID to each scheduler, and it interacts with the resource manager to

retrieve the placement information for the workers of each application. Using this information, the controller assigns a leaf scheduler to each rack that has workers. It then divides leaf schedulers into disjoint groups, where each group is assigned a spine scheduler and forms a logical tree. Horus uses a simple approach that aligns groups with datacenter pods, where a pod usually has 32-64 racks. In this case, the worker state is localized and maintained by schedulers within each pod (**P3**). This approach is efficient because packets exchanged between the spine and leaf schedulers within a pod traverse only one hop, which reduces packet latency and load on links, compared to the case where the spine is in a different pod.

Horus ***agents*** are lightweight processes that run on servers to track the load of workers. They also run a health check mechanism with the control plane of the leaf schedulers, which enables Horus to detect and react to worker failures. Agents are not involved in the scheduling decisions.

Horus offers ***APIs*** to datacenter applications to seamlessly submit tasks for execution and receive their results. To submit a task, Horus attaches a layer-4 header to packets, which makes Horus compatible with various routing protocols. The header includes a unique ID for each task, `taskID`. The uniqueness of task IDs is ensured by concatenating the application ID and a monotonically increasing sequence number. The application ID is computed by hashing the application pathname and adding a random number to ensure uniqueness. A sequence number of 32 bits is sufficient for all practical scenarios. Since tasks are expected to finish within micro or milliseconds, by the time the sequence number wraps around, if it ever does, earlier tasks would have been long completed.

Horus is designed for granular tasks, which are mostly contained within single packets as they typically carry parameters and paths to data. For example, in key-value stores [11–13], segments of the data are typically pre-distributed to workers, and tasks carry the queries to be executed on the data. For the common case of single-packet tasks, Horus does not maintain any per-task state at schedulers. Horus does support multi-packet tasks and maintains task-worker affinity using ideas similar to prior works, e.g., [57], as discussed in §A.4.

Horus Workflow. Workers of latency-sensitive applications are pre-deployed on CPU cores and initialized to be ready to execute tasks. A worker runs in a virtualized environment such as a container or microVM [14], and is allocated one or more cores. Horus assigns tasks to worker queues and does not dictate any intra-worker scheduling policy for distributing the tasks across the worker's cores. All workers of an application are assigned an anycast IP address by the controller. Tasks use the anycast address as the destination address in their packets. Tasks are scheduled in a recursive manner. When a task is submitted for execution, its packets are randomly forwarded to one of the spine schedulers assigned to this application (Step ❶ in Figure 3). Since the controller knows the distribution of workers across racks, the random selection of spine schedulers is weighted in proportion to the number of workers per pod.

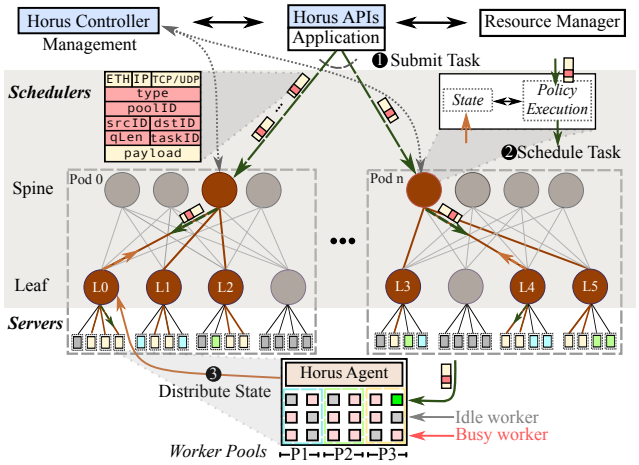


Fig. 3: The proposed Horus scheduling architecture.

This ensures load balancing across pods, which is important in case workers are non-uniformly distributed. The spine scheduler runs its policy to select a downstream leaf scheduler to handle the task, which in turn assigns the task to one of the workers in its rack (Step ❷).

After a worker completes a task, the agent running on that worker includes the latest load information into the header of the reply packet and sends it to the leaf scheduler in the rack (Step ❸). Upon receiving a reply packet, the leaf scheduler updates the load information in its memory and sends an update message to upstream spine schedulers, *if needed*, using our aggregated state update method in §3.4.

3.3 Scheduling Tasks in the Network

Switch Model. We consider the common switch model used by several switch vendors [10, 24, 61], in which a packet goes through a pipeline of multiple stages, where each stage has processing and memory components. This switch model supports line-rate packet processing by realizing two main design choices: (i) atomic memory updates and (ii) bounded packet latency. The first allows a packet to access *up to one* memory location at each stage, and the second limits the number of processing stages. As a result, the total available processing and memory resources for the stages are limited. To mitigate the impact of these design choices, a packet may have to be *recirculated* from the egress to the ingress for additional processing or to access the same memory block again at the cost of increased delay [74]. While this restrictive model enables line-rate packet processing, it makes it difficult to implement in-network task scheduling.

Proposed Scheduling Policy. In Horus, all spine and leaf schedulers employ the same scheduling policy and maintain the same data structures. Thus, we abstractly present the scheduling policy as follows. A scheduler assigns an arriving task to a lower-layer *node*. A node for a spine scheduler

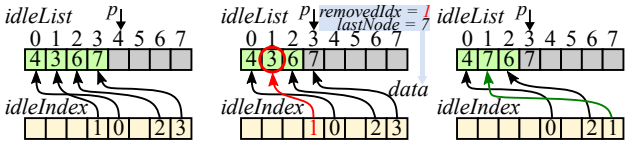
is a rack of servers, whereas it is a worker for a leaf scheduler. Optimally scheduling a task requires knowing the load of all downstream nodes and assigning the task to the least loaded one, i.e., implementing a JSQ-like policy. As we discussed in §2, JSQ is difficult to realize at the datacenter scale because it imposes high communication and processing overheads, and it may introduce task herding, where a burst of many tasks is sent to a node leading to periods of significant load imbalance.

Horus strives to *approximate* the JSQ scheduling policy at the datacenter scale while considering the dynamic nature of workloads and the restrictions of programmable switches. Specifically, Horus divides the scheduling decisions into two cases: (i) when some idle nodes are available and (ii) when all nodes are busy. In the first case, when a task arrives at a scheduler and the scheduler is aware of some idle nodes, it will send the task to one of them. For spine schedulers, an idle node is a rack that has at least one idle worker. In this case, the spine scheduler will send the task to the leaf scheduler of that rack, which in turn will select an idle worker within the rack, resulting in zero queuing time and minimizing the response time under light load. The challenge here is to track idle nodes at a large scale and in a way that can be implemented in programmable switches. We present the details of our solution later in this section.

In the second case, when a task arrives at a scheduler and all nodes are busy, the scheduler takes 2 *random* samples from the queue lengths of nodes and selects the least loaded node among the sampled values. This is referred to as the power-of-2 policy and is known to reduce the response time [58]. In addition, the randomization in taking samples prevents task herding, since it is unlikely that the same node will be repeatedly chosen for several consecutive tasks. Randomization is critically important for scheduling *granular* tasks as they are more susceptible to task herding. This is because granular tasks have short execution times, and load updates from workers may take relatively long times to reach various schedulers distributed across the datacenter. Implementing the power-of-2 policy at scale and in programmable switches with limited resources and strict constraints is challenging. We present new data structures to realize this policy later in this section and efficient methods to distribute load information among schedulers in §3.4.

Scheduling Tasks to Idle Nodes. We design a data structure, called *idleNodes*, for schedulers to track the IDs of idle nodes. Our data structure supports fast addition, removal, and retrieval of nodes with constant time in the switch data plane. It also requires a small number of memory accesses and minimal dependencies among memory blocks, which reduces the number of allocated processing stages in switches.

An insight that we used is that a scheduler needs *only* to know whether there exists an idle node in the list. It does not need to identify the temporal order of when nodes became idle. Building on this insight, we design a data structure that guarantees the following ***invariant***:



(a) Main structure (b) Remove: pass #1 (c) Remove: pass #2

Fig. 4: The proposed *idleNodes* data structure, which is designed to support fast operations in programmable switches.

If there are idle nodes, they will be contiguously stored at the top of the list.

Maintaining this invariant allows the scheduler to quickly find idle nodes. Notice that the data structure has to satisfy the invariant even when any arbitrary node is removed, i.e., becomes busy.

Figure 4a illustrates the main components of the proposed data structure. It has an *idleList* array of N items of node IDs, where N is the number of downstream nodes. It also has a pointer p that points to the first non-idle item in the list. p is initialized to zero. In addition, the data structure has an *idleIndex* array, which stores the indices of idle nodes. In Figure 4a, nodes 4, 3, 6, and 7 are idle, and *idleIndex* contains their locations in *idleList*.

Adding a node to *idleNodes* occurs when a scheduler receives an *idleAdd* message about a node n becoming idle. The scheduler writes the ID of n in *idleList[p]* and p in *idleIndex[n]*. It then increments p. Current programmable switches support an atomic *read-modify-write* operation in a single stage. This means that a scheduler can read the current p, increment it, and write it back in one stage. Thus, adding an idle node requires three processing stages only, one to update each of p, *idleList*, and *idleIndex*.

To schedule an incoming task to an idle node, a scheduler retrieves the node at *idleList[p-1]*. If this is a leaf scheduler, the retrieved node is removed from *idleList* by decrementing p and clearing the corresponding location of the retrieved node in *idleIndex*. This is because a node in this case represents a single worker, which will no longer be idle after sending the task to it. On the other hand, a node in a spine scheduler represents a whole rack, and sending a task to an idle leaf does not necessarily mean that the rack no longer has idle workers. Therefore, a spine scheduler does not remove the retrieved node *idleList[p-1]*. An idle node is removed from a spine scheduler only when it receives an explicit *idleRemove* message from the leaf scheduler represented by this node.

For both the spine and leaf schedulers, an incoming task is scheduled to an idle node *without* any delay or buffering and at line rate, since all operations on *idleNodes* are performed in a few consecutive processing stages in the switch. In addition, it is straightforward to show that the operations of adding a new idle node at *idleList[p]* and removing the idle node at *idleList[p-1]* maintain the invariant mentioned above.

Finally, the *idleNodes* data structure should support remov-

ing any arbitrary node n, while still maintaining the invariant. This is needed when a scheduler receives an *idleRemove* message for n. A scheduler removes node n by replacing it with the last idle node in the list. This ensures maintaining the invariant that all idle nodes are contiguously placed at the top of the list. It also allows the scheduler to conserve memory and use a constant number of processing stages. This operation needs to be performed in two passes because current programmable switches do not allow a packet to access the same memory location more than once in the same pass. In the first pass, the scheduler examines *idleIndex* to get the index of n in *idleList*, which is referred to as *removedIdx*. The scheduler also decrements p to retrieve the ID of the last idle node, which is called *lastNode*. In the second pass, the scheduler resubmits the packet with the additional data *lastNode* and *removedIdx*, where *lastNode* is then written into *idleIndex[removedIdx]*. An example is shown in Figures 4b and 4c.

We note that current programmable switches do not preserve the order of processing of resubmitted packets. For example, before processing the second pass of an *idleRemove* packet A, the first pass of another packet B could be processed by the switch, which may result in incorrect values of the indices. To prevent this potential race condition, we use a single memory location as a logical lock. The lock is acquired in the first pass and released in the second one. An *idleRemove* packet is dropped if it fails to acquire the lock. The sending node will resend a new *idleRemove* packet after receiving another task from the scheduler.

Scheduling Tasks to Busy Nodes. When there are no idle nodes ($p = 0$), a scheduler needs to realize the power-of-2 policy, which is more challenging than the case of idle nodes. This is because a scheduler needs to read two randomly selected indices from the *loadList*, while the switch model does not allow reading more than one item per packet from the same memory block. To address this restriction, Horus maintains two identical copies of the *loadList* in two different stages, where each array stores the load of all downstream nodes (one node per slot). Storing two copies allows the scheduler to read one random index from each copy and then compare them. When a new state update for a node arrives at a scheduler, it writes the updated load value to both copies.

After making a decision, a scheduler should update its view on the load information, which is stored in the *loadList*. This requires the scheduler to write back the updated load value to the corresponding *loadList* slot. As described earlier, the same memory block cannot be accessed twice per packet. A straightforward solution is to resubmit each packet to the pipeline and update the load state on the second pass. This, however, results in additional processing overhead and increases the scheduling latency. To address this issue, we propose a ***lazy state update*** algorithm, which resubmits a packet *only* if it will impact future scheduling decisions. That is, a scheduler keeps processing tasks using the potentially stale view of the *loadList* until it detects an update is needed.

Specifically, for a node m, we decompose its actual queue length q_m into a load value l_m and a drift value d_m, where $q_m = l_m + d_m$. We define the drift value as the number of tasks scheduled to a node that has not been reflected in its load value, and we store the drift values of all nodes in a data structure called *driftList*. Each scheduler maintains two copies of the *driftList* placed in two stages. Next, we find the *necessary condition* to resubmit a packet to update the load values. Upon receiving the first packet of a new task, a scheduler picks two random nodes m and n, and it reads their load values l_m and l_n from *loadList*. Without loss of generality, assume that $l_m < l_n$. Then, the scheduler should resubmit the packet *iff* $q_m > q_n$. That is, $l_m + d_m > l_n + d_n$. We rearrange the inequality to find the ***necessary condition*** to resubmit a packet as:

$$d_m > (l_n - l_m) + d_n. \quad (1)$$

The above condition means that as far as the drift in the load of node m is less than or equal to the difference between the load of n and the load of m plus the drift in the load of n, the scheduler will make the correct decision by choosing the node with the smaller load, which is m, by comparing their l_m and l_n stored in the *loadList*.

The proposed lazy state update algorithm works as follows. It first reads the load values of nodes m and n. It then identifies the node with the smaller load, say m, and it computes the difference $(l_n - l_m)$. Then, it reads d_m from the first copy of *driftList* to check how many more tasks are actually queued at m. If the drift value is lower than the difference between load values, i.e., $d_m < (l_n - l_m)$, the algorithm increments the corresponding drift value in each copy of the *driftList* for node m. Otherwise, when $d_m > (l_n - l_m)$, the algorithm resubmits the packet to update l_n and l_m. The algorithm does not include d_n in its calculations because it would violate the atomicity requirement in current programmable switches. The algorithm, however, does guarantee selecting the least loaded node among m and n. This is because $d_m > (l_n - l_m)$ still satisfies the necessary condition $d_m > (l_n - l_m) + d_n$. The less restrictive condition used by our algorithm allows implementing the power-of-2 policy in programmable switches at the cost of possibly resubmitting some extra packets than absolutely needed by the necessary condition.

An **example** illustrating this algorithm is given in §A.1.

3.4 Distributing State Among Schedulers

We design efficient mechanisms to distribute the necessary information among leaf and spine schedulers to update their states. This enables the execution of the proposed scheduling policy using fresh information with minimal overheads.

Distributing Worker State to the Leaf Layer. Since scheduling tasks to workers is only done by leaf schedulers, each leaf scheduler updates its state when selecting a worker for a task (§3.3). When a task is done executing on a worker, the agent modifies the `qLen` field in the header (Figure 3) and uses the

reply packet to report the updated load to the leaf scheduler. If the reply packet indicates the worker is idle, the leaf scheduler adds the `srcId` to the *idleList* and updates the *loadList* for the corresponding index.

Distributing Rack State to the Spine Layer. We consider two types of information to be distributed to the spine layer: (i) idleness of the rack and (ii) average load of the rack.

Idle State Update. When a leaf scheduler becomes aware of an idle worker, it sends an *idleAdd* packet to the spine scheduler it is linked with. This simple strategy balances and localizes the information about idle racks among the upper-layer spine schedulers. Once there are no more idle workers in a rack, the leaf scheduler in that rack sends an *idleRemove* packet to the linked spine to remove the leaf from its *idleList*.

Load State Update. Each spine scheduler tracks the load of a subset of racks that contain workers. Each leaf scheduler calculates the *average* load across workers in its racks and sends it to the linked spine. Directly calculating averages in programmable switches is, however, infeasible due to their restrictive programming model. We describe how we approximate averages in §A.2.

Since Horus is designed for short-lived tasks, continually sending every updated average load value to the spine would result in large communication and processing overheads on switches, without significantly modifying the average at the spine. Instead, we make each leaf scheduler locally compute the current average and maintain the previously sent average to the spine. Then, a leaf scheduler sends the update message only if the difference between the current and previous average load values is greater or equal to one, because this is the smallest integer value of load changes that could impact the scheduling decision. This ***aggregated update mechanism*** substantially reduces the number of update packets sent to spine schedulers without sacrificing the scheduling performance. In addition, we *piggyback* the state update information with the response packets sent by workers after completing tasks to minimize the communications overhead of Horus.

3.5 Horus Deployment Options

Horus does not rely on the structure of the datacenter network in its operation, and thus, it can be deployed in various networks. In addition, it does not dictate a specific routing protocol since it utilizes layer-4 headers. Furthermore, Horus can be incrementally deployed in datacenters. Suppose a fraction of the spine and leaf switches are upgraded to be programmable to support latency-sensitive applications. In this case, workers of these applications can be allocated in the racks with programmable switches, and the centralized controller in Horus can be configured to only use the programmable spine switches.

A more restrictive deployment scenario occurs when only a fraction of the spine switches are upgraded and all leaf switches are legacy. In this case, we can implement the leaf

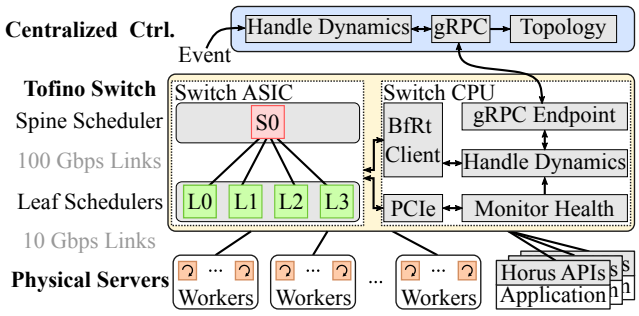


Fig. 5: Testbed setup.

scheduler on a server in each rack using efficient techniques such as kernel bypassing, as in prior works, e.g., Maglev [33]. These techniques were shown to process packets in microseconds. Workers and Horus agents in each rack would be configured to first direct their messages through the software leaf scheduler in the rack. This adds an additional delay, but it is deterministic and limited by the small RTT within racks.

Finally, the ideas of Horus can be used without any programmable switches. For example, leaf schedulers can be implemented in software as above, and the functionality of the spine schedulers can be integrated with the datacenter load balancers handling the submission of tasks.

4 Evaluation in a Testbed

4.1 Experimental Setup

Testbed. Our testbed, illustrated in Figure 5, has one 3.2 Tbps Intel Tofino switch, which has two hardware pipelines. We configure one of the hardware pipelines as a spine switch and run the spine scheduler of Horus on it. We emulate four leaf switches on the other hardware pipeline, where each switch represents a rack of servers. Leaf switches run the leaf schedulers of Horus. We connect the leaf switches to the spine switch using logical 100 Gbps links. In addition, the testbed has seven servers connected to the leaf switches through 10 Gbps links, where each server is equipped with an Intel 82599ES 10 GbE NIC. These servers are used as clients to generate tasks and as workers to execute tasks.

Horus Implementation. We have implemented a proof-of-concept of Horus consisting of leaf and spine schedulers, switch and centralized controllers, agents, and client APIs. All source code, testing scripts, and datasets are open source [3].

The leaf and spine schedulers are implemented in P4 [23] and deployed to the switch ASIC; a brief description of our P4 implementation can be found in §A.6. We implemented the switch and centralized controllers using Golang in about 6K lines of code. The controllers handle failures and dynamics, and they update the switch data structures accordingly. We implemented a set of APIs in C using DPDK to submit tasks and receive their results. We implemented the Horus agent in

about 100 lines of C code. The agent adds the worker load information to the reply packet after task execution is done.

For workers, we need to dispatch and run microsecond tasks. Recent intra-server OSes, e.g., ZygOs [64] and Shinjuku [43], support microsecond tasks and offer various scheduling policies. We modified the more recent Shinjuku [43] to dispatch tasks to worker queues based on Horus headers. In our experiments, we use one core per worker, bypassing Shinjuku's scheduling policy. We note that Horus does not dictate the use of any specific intra-server scheduling policy or OS.

Systems Compared Against. To the best of our knowledge, Horus is the first task scheduler that scales to the whole datacenter. RackSched [78] is the state-of-the-art in-network task scheduler, but it only scales to a single rack. We compare a simple version of Horus versus RackSched (referred to as RS) for single rack settings. We use the open-source P4 implementation of RackSched [8].

In addition, we consider two natural extensions of RackSched to support multiple racks. The first one integrates RackSched with a load balancer that uniformly distributes tasks to top-of-rack switches, which in turn run RackSched to assign tasks to workers within their racks. We refer to this system as RS-LB. In the second extension, we use RackSched *hierarchically*, meaning that we deploy it on both spine and leaf switches, where spine switches assign tasks to leaf switches using RackSched, which in turn assign tasks to workers also using RackSched. We refer to this system as RS-H. To ensure fair comparisons, we make each leaf scheduler send the load state update from workers to the spine scheduler immediately after each task is done.

Worker Placement. We consider two setups for worker distribution across racks: (1) **Uniform:** a total of 32 workers are uniformly distributed across all racks, where each rack has eight workers running on a physical server attached to the leaf switch (1 worker/core). The 32 workers run on four identical servers, each has Intel Xeon E-2186G CPU, 3.80 GHz, 12 cores, and 32 GB memory. (2) **Skewed:** a total of 48 workers are distributed as follows: two racks have four workers each, one rack has eight workers, and one rack has 32 workers (running on three physical servers, where one of the servers has Intel Xeon E5-2650 CPU, 2.3 GHz, 40 cores, 128 GB memory and runs 16 workers and the other two servers have the same specifications as the ones used in the Uniform setup and run 8 workers each).

Workloads. We evaluate Horus using two practical workload scenarios. The first scenario runs real tasks on the **RocksDB** engine [13], which is a high-performance key-value store developed by Facebook and is widely deployed in production [25]. The second scenario is synthetic and uses the **TPC-C** benchmark [5], which is an online transaction processing benchmark emulating e-commerce systems.

We create multiple RocksDB workloads with parameters similar to prior works [32, 43, 44, 78]. Specifically, our RocksDB workloads contain SCAN and GET tasks, where the first scans a range of objects (i.e., a relatively long task), and the second retrieves a specified number of objects (i.e., a short task). We construct the SCAN and GET tasks such that their dispersion is one order of magnitude: a SCAN request scans 5K objects with a median service time of 650 μs, whereas a GET request retrieves 60 objects where the median time of each request is 40 μs. We then employ two realistic distributions to generate concurrent long and short tasks. The first distribution is similar to workload A in the YCSB benchmark [26], and it consists of 50% GET and 50% SCAN tasks. The second one consists of 90% GET and 10% SCAN tasks, which is similar to Facebook's USR workload [17].

The TPC-C benchmark [5] consists of five tasks (or transactions) with different service times. We employ the profiled model in [32] of the benchmark to build a synthetic workload following the same task distribution and dispersion ratios. The five tasks have service times of 21.6, 22.68, 71.28, 332.64, and 378 μs, distributions of 44%, 4%, 44%, 4%, and 4%, and dispersion ratios of 1X, 1.05X, 3.3X, 15.4X, and 17.5X, respectively. We scaled the service times, compared to the profiled model in [32], so that clients in our testbed can smoothly generate tasks at high rates.

Task Arrival Model. We stress the system by generating tasks following a **Poisson** arrival process [78]. The Poisson process results in non-uniform inter-arrival delays and generates **bursts** that can cause temporary queue imbalance and impact the tail latency [32, 64].

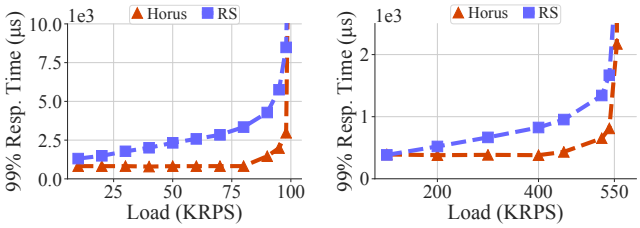
Horus schedulers have *no prior knowledge* about the service times, workloads, or arrival distributions.

Methodology and Performance Metrics. We vary the *system load* by incrementally increasing the number of tasks submitted for scheduling. The system load is measured in kilo requests per second (KRPS). We keep increasing the system load until we reach the capacity of the system, where the response time becomes unacceptably high for latency-sensitive applications (e.g., seconds or even minutes for tasks that should complete in micro or milliseconds). The *response time* is the period between submitting a task to a spine scheduler until it finishes execution on a worker.

An important metric for scheduling systems is the *achievable throughput*, which we define as the maximum system load that can be processed while meeting a given bound on a target performance metric, e.g., the tail (99th percentile) response time should not exceed 3ms.

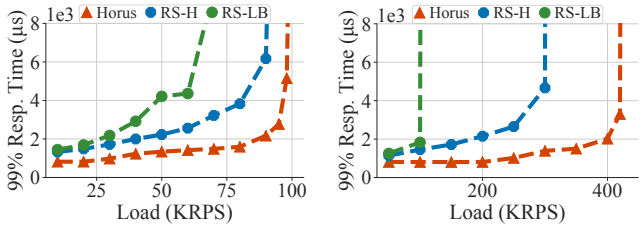
4.2 Comparison against State-of-the-Art

Horus vs. RackSched: Single Rack. We report the tail response time achieved by Horus and RackSched for the RocksDB and TPC-C workloads in Figure 6. There is no worker placement method used in this case, as all workers are located within the same rack. The results show that Horus consistently achieves much lower tail response times than RackSched, especially at high system loads. For example,



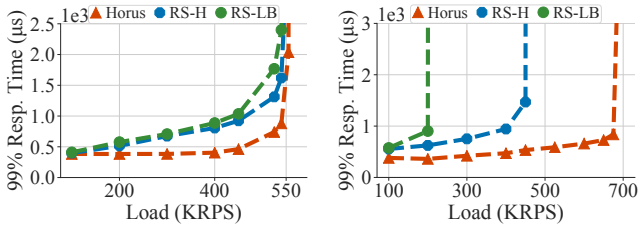
(a) RocksDB (b) TPC-C

Fig. 6: Comparing Horus vs. RackSched in single-rack setups.



(a) Uniform (b) Skewed

Fig. 7: Comparing Horus vs. RackSched extensions in multi-rack settings: Sample results from the RocksDB workloads.



(a) Uniform (b) Skewed

Fig. 8: Comparing Horus vs. RackSched extensions in multi-rack settings: Sample results from the TPC-C workloads.

for the RocksDB workload with 50% GET and 50% SCAN (Figure 6a) and at 80 KRPS system load, Horus reduces the tail response time by up to 75% compared to RackSched.

Horus achieves these significant gains because it tracks the loads on workers in a more accurate and efficient way than RackSched. Specifically, Horus identifies and uses idle workers within the rack, which results in low and constant response times at moderate loads. In contrast, RackSched does not explicitly track idle workers and relies only on the power of 2 policy of its scheduler, which may not always assign tasks to idle workers. In addition, RackSched relies on response messages from workers to update the load values. Since response messages are generated only after the completion of tasks, they may not capture the current state of workers by the time they arrive at the scheduler. In contrast, Horus updates the load values once it assigns a task to a worker, which enables it to have a real-time view of the workers' loads.

Horus vs. RackSched's Extensions: Datacenter. We compare Horus against RS-H and RS-LB for the different workloads and worker placements mentioned in §4.1. Representative samples of our results are shown in Figure 7 and Figure 8; the plots for all other scenarios are similar and given in §B. The results show that Horus consistently and substantially outperforms RS-H and RS-LB across all workloads and worker setups. For example, in the Uniform worker setup with 50% GET and 50% SCAN RocksDB requests (Figure 7a), Horus reduces the tail response time by up to 50% compared to RS-H when the system load is 80 KRPS; RS-LB could not support this load. This also means that Horus can achieve much higher throughputs than RS-H and RS-LB. For the same example in Figure 7a, if the target tail response time is 2 ms, Horus can achieve a throughput of up to 80 KRPS, whereas RS-LB and RS-H can only achieve up to 25 and 40 KRPS, respectively. That is, Horus can improve the throughput by up to 3.2X and 2X compared to RS-LB and RS-H, respectively, in this case. The results for the Skewed worker placement for the RocksDB workloads exhibit even higher gains, as shown in Figure 7b for the 90% GET and 10% SCAN workload. Similar gains are observed for TPC-C workloads with Unifrom and Skewed placements as shown in Figure 8.

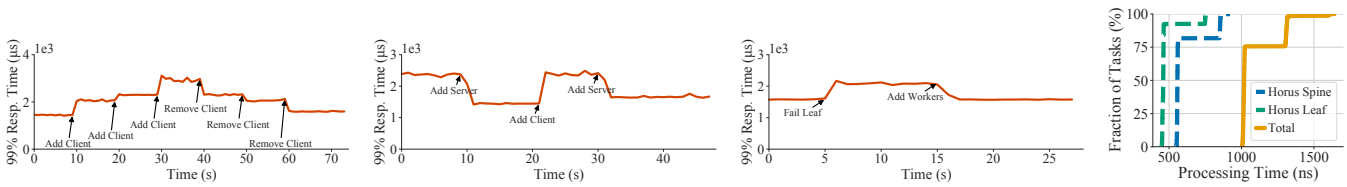
Horus achieves these gains across various workloads because its scheduling policy uses idle information to schedule tasks to any available idle workers, and it implements the efficient power-of-two method whenever there are no idle workers. In addition, Horus schedulers utilize the lazy update method to maintain up-to-date information about the worker loads, which significantly reduces the load imbalance across workers, as we show using large-scale simulations in §5.

4.3 Responsiveness and Overheads of Horus

We assess the performance of Horus in response to various dynamic events and analyze its overheads.

Dynamic Task Rate. We start with one client generating tasks at a rate of 30 KRPS. Every 10 seconds, a new client joins the system and sends additional tasks at a rate of 20 KRPS till the total task rate reaches 90 KRPS after 30s. Starting at 40s, we remove one client at a time at the same 10-second intervals. Figure 9a depicts the tail response time per second for the considered scenario. The results show that Horus can quickly react to the workload dynamics as the tail response time quickly drops after the task rate is decreased.

Dynamic Resource Scaling. In this scenario, one client starts sending tasks at a rate of 45 KRPS, where Horus schedules them to two racks of 16 workers. After 10s, we add a new server with 8 workers from another rack to the available worker pools. As Figure 9b shows, the tail response time drops to 1,416 μs after adding the server. We increase the task rate after 20s to 65 KRPS, which increases the response time. After 30s, we add another server with 8 workers, which reduces the response times to 1,644 μs. It takes between 1–2



(a) Dynamic task rate (b) Dynamic resource scaling (c) Handling failures (d) Latency

Fig. 9: Responsiveness and robustness of Horus to various dynamic events in (a)–(c), and its scheduling latency in (d).

seconds from when a resource allocation request is sent to the Horus controller till the response time is reduced.

Handling Failures. Initially, we use four leaf switches, and the client sends tasks at a rate of 65 KRPS. After 5s, we inject one leaf switch failure while the other three leaf schedulers remain active. We fail a switch by disabling its ports. Since the testbed has only one spine switch, we could not fail it. However, we analyze spine failures using simulations in §C.

As shown in Figure 9c, Horus can effectively use the remaining resources across other available racks to schedule tasks without interruption. While the spine scheduler has not been updated yet, a small fraction of the tasks sent to the failed leaf switch will be lost. At the rate of 65 KRPS, a leaf switch failure results in 1,444 failed tasks which are 2% of the total submitted tasks per second. At time 15s, we add another leaf scheduler with a rack of 8 workers. The response time is reduced within two seconds of adding the leaf scheduler.

Latency Overheads. We measure the total scheduling latency for a task at a switch by collecting the hardware timestamps at the beginning and end of the switch pipeline. Figure 9d shows the CDF of the measured scheduling latency at spine and leaf switches. As shown in the figure, the total latency is less than 1.6 μs for all tasks. We note that the small step increase observed in Figure 9d is due to the resubmitted tasks; recall that Horus resubmits a small fraction of tasks through the switch to update its state.

Other Results. We present more results in §B, including analyzing the fraction of resubmitted tasks.

5 Evaluation using Simulation

5.1 Simulation Setup

Datacenter Topology. We simulate a large datacenter with characteristics similar to the ones used in prior works, e.g., [54, 67]. Specifically, we simulate a network with a multi-rooted Clos topology composed of common 48-port switches with fully connected pods. The network has 1,152 leaf switches interconnected with the same number of spine switches. Each leaf switch manages a rack of 24 servers, leading to a network with a total of 27,648 servers. Each server has 32 cores and accommodates a maximum of 32 workers. The average per-hop delay between switches is set to 5 μs [35, 55], and the average packet loss rate is set to 1e$-$3% [38, 80].

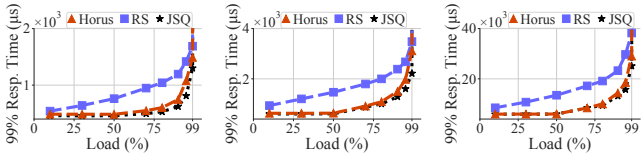
Worker Placement. We simulate 1K concurrent worker pools, where each worker pool processes tasks of a large-scale datacenter application. Similar to [54, 67], we allocate workers to pools following an exponential distribution with min=50, max=20K, and mean=685. The total number of workers is 685K. Each worker has a private task queue and runs an FCFS policy to process tasks.

Workloads. To analyze the performance in realistic settings, we generate three workloads with different distributions for the task processing time: (1) **Exp (100)** is an exponential distribution with mean=100 μs, which represents the processing time of a single type of tasks with its variability, such as tasks that occur in in-memory key-value stores and caching servers [43, 78], (2) **Bimodal** (50%–50 μs, 50%–500 μs), and (3) **Trimodal** (33.$\overline{3}$%–50 μs, 33.$\overline{3}$%–500 μs, 33.$\overline{3}$%–5000 μs), which together simulate patterns observed in a mix of simple and complex tasks such as *get/put* and *scan* operations [63]. These workloads are similar to the ones used to evaluate RackSched, which ensures fair comparisons.

Task Arrival Model. We generate tasks following a **Poisson** process to stress the system under **bursty** arrival and non-uniform patterns. We keep increasing the system load until we reach the maximum for each worker pool, which is given by $\lambda = n/\bar{s}$, where n is the number of workers in the pool and $\bar{s}$ is the mean task execution time. In the figures, we report the system load as a percentage of the maximum load.

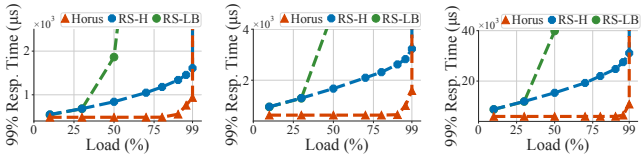
5.2 Comparison against the State-of-Art

Horus vs. RackSched: Single Rack. We compare the performance of Horus versus RackSched in single rack settings. Additionally, we compare against JSQ to show how far Horus is from the theoretical performance bounds; as we discussed in §2, JSQ is not implementable in real environments. We simulate a hypothetical switch that executes JSQ with zero-delay state updates. To be able to compute the optimal results by JSQ, we consider only 10 concurrent applications with workers deployed within the same rack. Figure 10 shows the tail response times for different workloads. Horus substantially outperforms RackSched, and its performance is close to JSQ. This is because Horus tracks the loads on workers more accurately than RackSched, as discussed before in §4.2.



(a) Exponential (b) Bimodal (c) Trimodal

Fig. 10: Comparing Horus against RackSched and JSQ in single-rack settings using simulation.



(a) Exponential (b) Bimodal (c) Trimodal

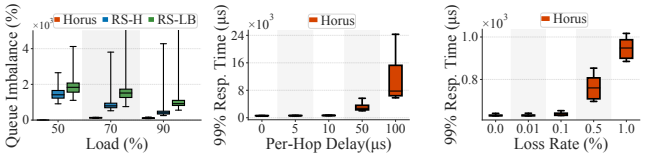
Fig. 11: Comparing Horus against RackSched's extensions in multi-rack settings using simulation.

Horus vs. RackSched's Extension: Datacenter. We compare Horus versus the extensions of RackSched described in 4. To ensure fair comparisons with RS-H, we make each leaf scheduler send the load state of the rack to every available spine scheduler. Figure 11 depicts the tail response time observed by the worker pool with the median size for Horus, RS-LB, and RS-H. The figure shows that Horus reduces the tail response time by up to 3X at moderate loads, and it achieves higher throughput for any target tail response time. The gains are more significant when the dispersion in the task execution times is high, as shown for the Bimodal (Figure 11b) and Trimodal distributions (Figure 11c) compared to the exponential distribution (Figure 11a). This is because the high diversity in the task execution time, coupled with the variability introduced by the Poisson inter-arrivals of tasks, may introduce imbalance in the queues at workers, which are better addressed by Horus.

To analyze the reasons behind the achieved gains, we measure the imbalance in the queues at workers. We define the *imbalance* as the ratio between the maximum queue length and the average queue length within each worker pool. We measure the imbalance every 50 μs and plot the results in Figure 12a, as an interquartile range (1st and 99th percentiles). The results show that Horus spreads the load more uniformly across workers because it tracks their loads more accurately.

5.3 Analysis of Horus

Impact of Network Delay. We analyze the impact of network delays on the scheduling performance of Horus. We increase the average per-hop delay between switches from 0 to 100 μs and measure the tail response time. Figure 12b shows the interquartile range of the tail response time of different applications. As the figure shows, even when the per-hop delay is 10 μs, which is twice the average in real environments [35,55], Horus maintains a low tail response time with small variation. As the per-hop delay increases further to unrealistically high values (50 and 100 μs), the delayed updates from the leaf to spine layer start to impact the scheduling quality of Horus, because such updates may deliver stale information about the worker status in different racks, as shown in Figure 12b.



(a) Queue imbalance (b) Impact of delay (c) Impact of losses

Fig. 12: Analysis of Horus.

Impact of Packet Losses. Datacenter networks have very low loss rates, up to 0.01% [38, 80]. We vary the average packet loss rate from 0 to 1%, which is 100X the loss rate in real datacenters. Figure 12c shows the tail response time of different applications versus the packet loss rate. Even when the average loss rate is 10X (i.e., 0.1%) the normal rate, Horus is not significantly impacted by packet losses. This is because Horus re-transmits the important information such as *idleAdd* and *idleRemove* messages. However, with extreme loss rates, states maintained at switches can be stale for short periods, which causes an increase in response times.

Other Results. We present more results in §C, including analyzing the impact of worker placement, scheduler failures, and the contributions of Horus components to its performance.

6 Conclusions

We presented the design, implementation, and evaluation of Horus, a granular task scheduler for multi-tenant datacenters. In contrast to traditional schedulers, Horus offloads the scheduling of latency-sensitive tasks to network switches, which enables scheduling them at high rates in real time. Horus distributes the load information of workers among network switches, and it introduces a new scheduling policy that minimizes the task response time and does not buffer tasks in switches. We presented multiple ideas and data structures to efficiently realize the scheduling policy in programmable switches. We also designed methods to propagate updated load values among schedulers. We implemented Horus in a testbed with a modern programmable switch and compared its performance against RackSched [78], the state-of-art in-network task scheduler. We also evaluated the performance of Horus in large-scale simulations. Our experimental and simulation results showed that Horus is scalable and robust, and it substantially outperforms RackSched across all performance metrics in both single- and multi-rack settings.

References

[1] Apache openwhisk. https://openwhisk.apache.org/. [Online; accessed September 2022].

[2] Aws lambda. https://aws.amazon.com/lambda/. [Online; accessed September 2022].

[3] Network and Multimedia Systems Lab (NMSL) at SFU. https://nmsl.cs.sfu.ca/.

[4] Placement groups - amazon elastic compute cloud. https://bit.ly/3zMuugf. [Online; accessed April 2023].

[5] Tpc-c. https://www.tpc.org/tpcc/. [Online; accessed September 2022].

[6] Data sharing on traffic pattern inside facebook's data center network. https://bit.ly/3tLgIqz, Jan 2017. [Online; accessed April 2023].

[7] Fault tolerance through optimal workload placement. https://bit.ly/2VMR6wQ, Sep 2020. [Online; accessed April 2023].

[8] Racksched Git Repository. https://github.com/netx-repo/RackSched, 2020. [Online; accessed September 2022].

[9] Apache Lucene search engine. https://lucene.apache.org/, 2021. [Online; accessed September 2022].

[10] Intel Tofino ASIC. https://intel.ly/3fmIP8Q, 2021. [Online; accessed September 2022].

[11] Memcached key-value store. http://memcached.org/, 2021. [Online; accessed September 2022].

[12] Redis in-memory data structure store. https://redis.io/, 2021. [Online; accessed September 2022].

[13] RocksDB. https://rocksdb.org/, 2021. [Online; accessed September 2022].

[14] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proc. of USENIX NSDI'20*, pages 419–434, Santa Clara, CA, February 2020.

[15] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proc. of ACM SIGCOMM'14*, page 503–514, Chicago, IL, August 2014.

[16] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In *Proc. of ACM SoCC'18*, pages 263–274, Carlsbad, CA, October 2018.

[17] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS'12*, page 53–64, London, United Kingdom, June 2012.

[18] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty tenants and the cloud network sharing problem. In *Proc. of USENIX NSDI'13*, pages 171–184, Lombard, IL, April 2013.

[19] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A High-Speed Load-Balancer design with guaranteed Per-Connection-Consistency. In *In Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 667–683, Santa Clara, CA, February 2020. USENIX Association.

[20] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, March 2017.

[21] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.

[22] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *Proc. of ACM SIGCOMM'12*, pages 431–442, Helsinki, Finland, August 2012.

[23] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[24] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proc. of ACM SIGCOMM'13*, pages 99–110, Hong Kong, China, August 2013.

[25] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In

Proc. of USENIX FAST'20, pages 209–223, Santa Clara, CA, February 2020.

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of ACM SoCC'10*, page 143–154, Indianapolis, IN, June 2010.

[27] Penglai Cui, Heng Pan, Zhenyu Li, Jiaoren Wu, Shengzhuo Zhang, Xingwu Yang, Hongtao Guan, and Gaogang Xie. Netfc: Enabling accurate floating-point arithmetic on programmable switches. In *Proc. of IEEE ICNP'21*, pages 1–11, Virtual Event, November 2021.

[28] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proc. of ACM SOCC'16*, pages 497–509, Santa Clara, CA, October 2016.

[29] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of USENIX ATC'15*, pages 499–510, Santa Clara, CA, July 2015.

[30] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proc. of ACM ASPLOS'14*, page 127–144, Salt Lake City, UT, February 2014.

[31] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proc. of ACM SoCC'15*, pages 97–110, Kohala Coast, HI, August 2015.

[32] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proc. of ACM SOSP'21*, pages 621–637, Virtual Event, October 2021.

[33] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *In Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 523–535, Santa Clara, CA, March 2016. USENIX Association.

[34] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *Proc. of USENIX NSDI'17*, pages 363–376, Boston, MA, March 2017.

[35] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proc. of OSDI'16*, volume 16, pages 249–264, Savannah, GA, 2016.

[36] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proc. of ACM SIGCOMM'11*, pages 350–361, Toronto, Canada, August 2011.

[37] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proc. of USENIX OSDI'16*, pages 99–115, Savannah, GA, November 2016.

[38] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. of ACM SIGCOMM'15*, pages 139–152, London, United Kingdom, August 2015.

[39] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of USENIX NSDI'11*, pages 22–22, Boston, MA, March 2011.

[40] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *Proc. of ACM SIGCOMM'15*, pages 435–448, London, United Kingdom, August 2015.

[41] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proc. of ACM SOSP'17*, pages 121–136, Shanghai, China, October 2017.

[42] Sangeetha Abdu Jyothi, Mo Dong, and P. Brighten Godfrey. Towards a flexible data center fabric with source routing. In *Proc. of ACM SOSR'15*, pages 1–8, Santa Clara, CA, June 2015.

[43] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proc. of USENIX NSDI'19*, pages 345–360, Boston, MA, February 2019.

[44] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proc. of ACM SOSP'21*, pages 605–620, Virtual Event, October 2021.

[45] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Practical Scheduling for Real-World Serverless Computing. November 2021. arXiv: 2111.07226.

[46] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. of USENIX ATC'15*, pages 485–497, Santa Clara, CA, July 2015.

[47] D. Katz and D. Ward. Bidirectional forwarding detection (bfd). RFC 5880, RFC Editor, June 2010.

[48] Ibrahim Kettaneh, Sreeharsha Udayashankar, Ashraf Abdel-hadi, Robin Grosman, and Samer Al-Kiswany. Falcon: Low latency, network-accelerated scheduling. In *Proc. of EuroP4'20*, pages 7–12, Barcelona, Spain, December 2020.

[49] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proc. of USENIX OSDI'18*, pages 427–444, Carlsbad, CA, October 2018.

[50] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *Proc. of USENIX ATC'19*, pages 863–880, Renton, WA, July 2019.

[51] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[52] Collin Lee and John Ousterhout. Granular Computing. In *Proc. of ACM HotOS'19*, pages 149–154, Bertinoro, Italy, May 2019.

[53] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proc. of ACM SOSP'17*, pages 104–120, Shanghai, China, October 2017.

[54] Xiaozhou Li and Michael J Freedman. Scaling ip multicast on datacenter topologies. In *Proc. of ACM CoNEXT'13*, pages 61–72, Santa Barbara, CA, December 2013.

[55] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. In *Proc. of OSDI'20*, pages 1171–1186, 2020.

[56] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient microservices on SmartNIC-Accelerated servers. In *Proc. of USENIX ATC'19*, pages 363–378, Renton, WA, July 2019.

[57] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. of ACM SIGCOMM '17*, pages 15–28, Los Angeles, CA, August 2017.

[58] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[59] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proc. of ACM SIGCOMM'09*, pages 39–50, Barcelona, Spain, August 2009.

[60] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proc. of ACM SOSP'13*, pages 69–84, Farminton, Pennsylvania, November 2013.

[61] Recep Ozdag. Intel® Ethernet Switch FM6000 Series - Software Defined Networking. *Intel*, page 8, 2012.

[62] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. of ACM EuroSys'18*, pages 1–14, Porto, Portugal, April 2018.

[63] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. Fast scans on key-value stores. *Proc. of the VLDB Endowment*, 10(11):1526–1537, 2017.

[64] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proc. of ACM SOSP'17*, pages 325–341, Shanghai, China, October 2017.

[65] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *Proc. of USENIX NSDI'21*, pages 785–808, Virtual Event, April 2021.

[66] Malte Schwarzkopf. Cluster scheduling for data centers: Expert-curated guides to the best of cs research: Distributed cluster scheduling. *ACM Queue*, 15(5):78–89, October 2017.

[67] Muhammad Shahbaz, Lalith Suresh, Jen Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo:

Source-routed multicast for public clouds. In *Proc. of ACM SIGCOMM'19*, pages 458–471. Beijing, China, August 2019.

[68] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proc. of USENIX ATC'20*, pages 205–218, Virtual Event, July 2020.

[69] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proc. of ACM SIGCOMM'15*, page 183–197, London, United Kingdom, August 2015.

[70] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A Scalable Low-Latency Serverless Platform. In *Proc. of ACM SoCC'21*, pages 138–152, Seattle, WA, November 2021.

[71] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *Proc. of USENIX OSDI'20*, pages 787–803, Virtual Event, November 2020.

[72] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of SOCC'13*, pages 1–16, Santa Clara, California, October 2013.

[73] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proc. ACM EuroSys'15*, pages 1–17, Bordeaux, France, April 2015.

[74] Dingming Wu, Ang Chen, T. S. Eugene Ng, Guohui Wang, and Haiyong Wang. Accelerated Service Chaining on a Single Switch ASIC. In *Proc. of ACM HotNets'19*, pages 141–149, Princeton, NJ, November 2019.

[75] Dingming Wu, Yiting Xia, Xiaoye Steven Sun, Xin Sunny Huang, Simbarashe Dzinamarira, and TS Eugene Ng. Masking failures from application performance in data center networks with shareable backup. In *Proc. of ACM SIGCOMM'18*, pages 176–190, Budapest, Hungary, August 2018.

[76] Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira, Dingming Wu, Xin Sunny Huang, and TS Eugene Ng. A tale of two topologies: Exploring convertible data center network architectures with flat-tree. In *Proc. of ACM SIGCOMM'17*, pages 295–308, Los Angeles, CA, August 2017.

[77] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proc. of ACM SIGCOMM'20*, pages 126–138, Virtual Event, August 2020.

[78] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *Proc. of USENIX OSDI'20*, pages 1225–1240, November 2020.

[79] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers(technical report). 10 2020.

[80] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proc. of ACM SIGCOMM'17*, pages 362–375, Los Angeles, CA, 2017.

Appendix A More Details of Horus

This appendix describes additional details that we could not fit into the main paper.

A.1 Example: Lazy State Update

We illustrate the implementation of the lazy state update algorithm in programmable switches in Figure 13. The figure shows the processing of a resubmitted packet. The two copies of *loadList* are maintained in stages 1 and 2, and the two copies of *driftList* are in stages 4 and 5. Each of stages 1 and 2 randomly selects a node and passes the ID of this node and its load to the subsequent stages as metadata (shown at the bottom of the figure). The selected nodes in this example are $m = 0$ from stage 1 and $n = 2$ from stage 2. Stage 3 chooses the node with the smaller load, m, and computes the difference in the load between m and n, $l_n - l_m = 1$.

In stage 4, the drift, $d_m = 3$, is found to be greater than the load difference between m and n, i.e., $d_m > (l_n - l_m)$. This means that the initial scheduling decision (assuming $q_m < q_n$) may not be accurate, since it could be underestimating the actual load of m. Thus, a flag is set to resubmit this packet to update all lists and select the correct node. The drift for the second node, d_n, is read in stage 5 from the second copy of *driftList* and passed to stage 6, which calculates the actual load of both nodes q_m and q_n as the sum of drift and load values and resubmits the packet with this data.

In the resubmission pass, the scheduler becomes aware of the actual load values of the two nodes, i.e., $q_m = 3+3 = 6$ and $q_n = 4+1 = 5$, and it selects the least loaded node, which is now $n = 2$. The scheduler then increments the *loadList* of the least loaded node. The *loadList* is updated with the new values, and the corresponding entries in the *driftList* are reset to 0. Notice that in the resubmission pass, we do not read items from the *loadList* or *driftList* as these values are injected in the resubmitted packet. This allows us to increment the load list values while respecting the strict memory access requirement of programmable switches.

Finally, we note that memory updates in programmable switches are done atomically: a packet may update memory locations at different stages of the pipeline, but the following packet will not observe such updates until they re fully completed. *loadList* and *driftList* values are only updated upon receiving Load Update packets from downstream nodes. *driftList* is only incremented* in the first pass for task packets, and *loadList* is only incremented in the resubmission pass. Therefore, even with re-ordering of resubmitted packets and other state updates, Horus does not introduce any race conditions.

*An atomic read-modify-write operation that prevents race conditions in the pipeline.

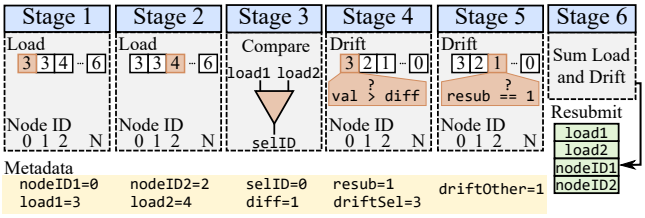


Fig. 13: Example of resubmitting a packet while scheduling tasks to busy workers.

A.2 Realizing Average Queue Length

Calculating an average value in the switch data plane is infeasible due to the lack of support for floating-point arithmetic in programmable switches. To mitigate this issue, we approximate the calculation of the average by using a *fixed-point* representation. We use a 32-bit number to represent an average load value, which is communicated among switches using the `qLen` field in Figure 3. The 32 bits are divided equally among the integer and fraction parts, which can support an accuracy of 2^{-16} that is sufficient for most practical cases. We note that Horus can support existing approximations of floating-point operations, e.g., [27], at the cost of additional switch resources.

When the resource manager allocates w workers in a rack for a worker pool, the Horus controller calculates an additive factor with a value of $1/w$ using the fixed-point representation, and updates the corresponding leaf switch with this value, which is maintained in a table. For example, when a worker pool has 8 workers in a rack, then the additive factor is 0x00002000, which is interpreted as $1/8$. Notice that this value can be maintained in switches as a 32-bit number.

The calculation of an average value is a sequence of increments/decrements of the corresponding additive factor. The leaf scheduler uses the `poolID` as a key to access the table, and uses add/subtract of the additive factor to increase/decrease the average value for every started/finished task. As an example, when the 8 workers have 11 tasks in their queues, then the average queue length is $11/8 = 1+3/8$, which can be represented as 0x00016000.

A.3 Handling Failures and Packet Losses

Horus employs simple mechanisms to notify the impacted schedulers about various failures to reduce disrupting ongoing tasks. These mechanisms are deployed at the centralized controller and switches' controllers. We rely on existing protocols [47, 59, 75] to detect failures and notify the fabric manager. We assume that the centralized controller is replicated on multiple servers using algorithms such as Paxos [51], and it receives failure events from the fabric manager.

Switch Failures. Upon receiving a leaf or spine switch failure event, the centralized controller instructs all impacted

switches' controllers to remove the failed switch from their memories and reset any state linkage information. Thus, the failed switch will not be used for scheduling incoming tasks. We note that leaf switch failures usually result in network partitioning. Therefore, no new tasks could be assigned to workers inside the rack during leaf switch failures, and Horus does not attempt to retrieve the workers' soft state stored at the failed leaf switch.

Server Failures. The leaf switch controller locally detects the failures of servers in the rack based on heartbeat packets and a fixed timeout value. This localizes the state maintenance at the rack level. After detecting a failure, the leaf controller decreases the number of available workers and instructs its data plane to remove the failed worker from its data structures.

Packet Losses. Packets carrying the submitted tasks as well as packets sent to update the state at different schedulers can be lost. Horus delegates handling the first case to the submitting applications. Since Horus is designed for short-lived (μ or millisecond sec) tasks, applications typically submit multiple tasks in parallel and ignore the failed ones, as the latency of resubmitting a failed task can be larger than the execution time of the task itself.

Horus only retransmits the lost Idle Add/Remove messages that are sent from a leaf to a spine. To detect a lost update packet, the leaf and spine schedulers use a simple protocol: Spine schedulers set the *qLen* field of the scheduled task to *idleSelected* when it selects a rack from the *idleList*. When the task arrives at the leaf, it checks the header field to determine the state of the rack in the spine's memory. If it mismatches the current state, it will resend an Idle Add/Remove packet to the spine based on the correct state. Horus frequently sends Load Update packets to maintain the state at different schedulers. Horus does not retransmit lost Load update packets, since this would be costly and ineffective for the target application environment. Rather, it relies on subsequent update packets carrying fresh information to bring the state up to date. Recall that Horus strives to *approximate* the current load on workers; it is nearly impossible to make schedulers track the exact load on every single worker, given the very short execution time of tasks and the high dynamics nature of the workload.

A.4 Handling Multi-packet Tasks

When a task is composed of multiple packets, Horus schedulers need to send these packets to the same worker. This is known as task affinity. Applications expecting to submit multi-packet tasks, set the *isLastPacket* flag to 0 for all packets of the task except the last one. Also, for the non-first packets of the task they should set the *type* in the Horus header to *taskContinuation* (Figure 3).

Similar to prior approaches, e.g., [57, 78], Horus maintains a connection table at switches. When the first packet of a task arrives at a spine or leaf scheduler, the scheduler assigns it to a node using its normal operation. Let us denote the ID of this node by *nodeID*, which can be an ID of a rack (in the case of a spine scheduler) or an ID of a worker (in the case of a leaf scheduler). If the *isLastPacket* field is not set for the first packet of task, the scheduler adds the entry $\langle hash(poolID, taskID),\ nodeID\rangle$ to its connection table. For subsequent packets with the same *taskID*, the scheduler forwards them to the same node using the connection table. We note that the leaf scheduler adds another field to each entry of the connection table: *spineID*, which specifies the ID of the spine switch from which the packet came.

Entries in the connection table are removed in one of two ways. First, when a task submission by client completes, the final packet with *type* of *taskContinuation* comes and the *isLastPacket* flag is set to 1, then the scheduler removes the corresponding connection table entry. The second way to remove an entry from the connection table is through timeout. Each entry automatically disappears after a pre-specified period of time (in the order of 10s of milliseconds). This takes care of failed tasks and lost final packets.

We note that the connection table is only maintained for multi-packet tasks, not for short-lived granular tasks that are composed of single packets.

A.5 Horus Overheads

Horus imposes multiple types of overheads. First, it attaches a small header of size 11 bytes to include information such as task ID and queue length as shown in Figure 3. Second, it maintains state at switches to enable realizing a load-aware scheduling policy. This consumes part of the memory of the programmable switches. For granular tasks, which is the main target of Horus, the maintained state at switches is *independent* of task rates, which is an important property that makes Horus scalable.*

The state, however, grows with the number of applications submitting tasks and the number of workers assigned to each of them. Let us consider one datacenter application with a total of W workers allocated across R racks, where typically $R \ll W$. Spine schedulers do not maintain state about individual workers. Rather, they maintain the average worker load in different racks. Thus, the memory requirements on spine schedulers are in the order of $O(R)$. Specifically, each spine scheduler maintains one copy of the *idleList* and two copies of each of the *loadList* and *driftList* data structures. It also maintains a table mapping each rack to the fractional fixed-point additive factor for updating the load values. The number of entries in each list is R, each is 16 bits.

Leaf schedulers maintain a state about workers in their racks, which is on average $O(W/R)$ when workers are uniformly allocated across racks, and $O(W)$ in the worst case

*For multi-packet tasks, Horus maintains a connection table to ensure task affinity. As discussed in §A.4, applications expected to submit multi-packet tasks set the `isLastPacket` field to 0 in the Horus header (Figure 3), and only state about such tasks are maintained in the connection table.

Algorithm 1 Scheduling and state updates for idle nodes

```
    - p: reg. pointing to the next available slot in idleList
    - idleList: reg. array that holds the IDs of idle nodes
    - idleIndex: reg. array holding indices of the nodes in
    idleList.
1:  // On an idleAdd pkt received
2:  function ADD(pkt)
3:      readInc(p)
4:      idleList[p] ← pkt.srcID // pkt.srcID is idle node ID
5:      idleIndex[pkt.srcID] ← p
6:  function SCHEDULETASKIDLE(pkt)
7:      readDec(p) // Only a leaf scheduler decrements p
8:      selectedNode ← idleList[p − 1]
9:      Update pkt with IP of selectedNode and Forward
10: (A) On an idleRemove pkt received: First Path
11: function REMOVE(pkt)
12:     readDec(p)
13:     lastNodeID ← idleList[p]
14:     removedNodeIdx ← idleIndex[pkt.srcID]
15:     resubmit(lastNodeID, removedNodeIdx)
16: (B) On an idleRemove pkt received: Resubmit Path
17: function REMOVE(lastNodeID, removedNodeIdx)
18:     idleList[removedNodeIdx] ← lastNodeID
19:     idleIndex[lastNodeID] ← removedNodeIdx
```

when all workers are in the same rack. Leaf schedulers maintain the same *idleList*, *loadList*, and *driftList* data structures as spine schedulers.

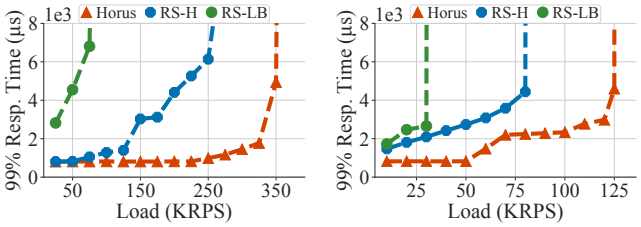
For illustration, consider an application with $W = 10,000$ workers uniformly distributed across $R = 10$ racks. A spine scheduler would need up to 120 bytes of memory, whereas a leaf scheduler would need approximately 1000 bytes. The recent Tofino switch has a few hundred Megabytes of memory.

In addition, Horus exchanges messages to update the state at schedulers. However, following our design principles, we keep the worker state localized within individual racks, and we only send aggregated updates to spine schedulers. In addition, Horus piggybacks the update messages with response packets of tasks.

A.6 Pseudo Code and P4 Implementation

Algorithm 1 and Algorithm 2 show the high-level pseudo code of Horus for handling idle nodes and busy nodes. Our implementation only uses the stages in the ingress pipeline of the switch. Upon receiving a task, Horus schedulers will access and read the number of available idle noads and only use the procedure in Algorithm 2 (line 15) if there are no available idle nodes.

Recall that removing an idle node after being selected by SCHEDULETASKIDLE depends on the scheduler type. A leaf scheduler removes the selected idle worker immediately after sending a task to it because the worker is no longer idle. A spine scheduler, however, only reads p and does not decrement it, since removing an idle node in a spine scheduler is triggered by an *idleRemove* packet sent by a leaf scheduler.



(a) Uniform (b) Skewed

Fig. 14: Testbed results for the RocksDB workload. Comparing Horus vs RackSched extensions in multi-rack settings: (a) 90%-GET, 10%-SCAN, (b) 50%-GET, 50%-SCAN distributions.

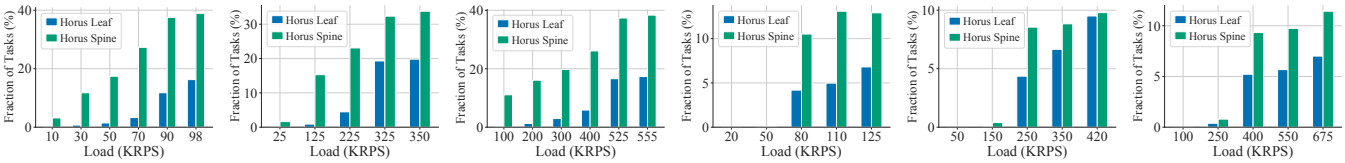
The UPDATE procedure (Algorithm 2, line 2) is triggered by receiving a reply packet from the worker at leaf schedulers. At the spine layer, updates are explicitly triggered by *loadUpdate* packets sent by leaf schedulers. Note that leaf schedulers increment the load or drift values after selecting the target worker by one, which reflects the selected worker queue length after the assignment. At the spine layer, the values are incremented by the additive factor for the selected rack ($1/\#workers$), based on the value stored in the tables as described in §A.2, which is not shown in the pseudo codes for simplicity.

Generating state update messages in the data plane can be challenging since packet generation is triggered based on the real-time state of the workers. Limiting the processing to the ingress pipeline of the switch enables us to realize this efficiently, without recirculating the packets. The leaf switch checks the trigger conditions when processing each arriving packet. If the condition to send an update is met, it duplicates the original packet via the traffic manager and sends the original copy to its destination. The switch then modifies the header fields of the other copy and sends it as an update message to the upper layer. As an example, *idleRemove* packet is generated when the switch receives a task and becomes aware that no more idle workers are available. The switch duplicates the packet, forwards the original packet to the worker, changes the *type* field to *idleRemove*, sets the *srcId* to the ID of the leaf switch, and sends the copied packet header to the spine.

Appendix B More Results from Testbed

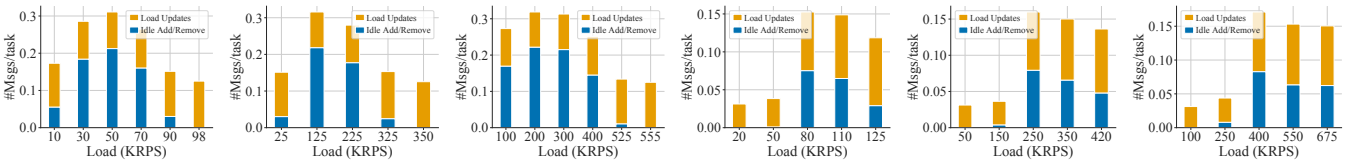
This appendix provides more evaluation results obtained from the testbed.

Response Time. We present additional results for comparing Horus against RackSched's extensions in the multi-rack set-



(a) RocksDB (50-50) (b) RocksDB (90-10) (c) TPC-C (d) RocksDB (50-50) (e) RocksDB (90-10) (f) TPC-C

Fig. 15: Fraction of resubmitted tasks for the Uniform worker placement in (a)-(c) and for the Skewed worker placement (d)-(f).



(a) RocksDB (50-50) (b) RocksDB (90-10) (c) TPC-C (d) RocksDB (50-50) (e) RocksDB (90-10) (f) TPC-C

Fig. 16: Rate of state update messages for the Uniform work placement in (a)-(c) and for the Skewed worker placement in (d)-(f).

tings in Figure 14. As shown in Figure 14a, for the workload with 90%-GET and 10%-SCAN requests, even using uniform worker placement across the racks, Horus achieves significantly lower tail latency at moderate and high loads. Similarly, Figure 14b shows that for the workload with 50%-GET and 50%-SCAN requests, under a skewed worker placement, the system maintains a lower latency and sustains up to 125 KRPS throughput, which is 60% higher than RS-H and 400% higher than RS-LB.

More Overhead Results. A Horus scheduler may selectively resubmit a fraction of task packets to the switch pipeline to update the scheduler view after making a scheduling decision. We measure the fraction of task packets that are processed twice by a spine or leaf switch in our experiments. Figure 15 shows that the maximum fractions of resubmitted tasks are 38% and 13% for the Uniform and Skewed placement setups, respectively. When the load is low, the scheduler does not need to resubmit tasks as most of the tasks are scheduled based on idle nodes, and the rate of reply packets is high enough to automatically update the scheduler state. We note that simple solutions to update the state would result in a 100% resubmission rate because they resubmit every packet after scheduling a task.

Further, we note that the rate of resubmitted packets is correlated with the placement of workers and the size of worker pool. Recall that resubmissions are triggered when the drift value is greater than the difference between the two sampled load values while scheduling a task. Therefore, the rate of resubmissions at the spine is impacted by the closeness of the average load values of the racks in the spine memory. In addition, the number of available racks impacts the rate of resubmissions at the spine layer. That is, having a small number of racks increases the probability of triggered resubmissions as this increases the likelihood of drawing the same two random samples for multiple arriving tasks before the state update arrives, which results in a resubmission.

Next, we analyze the different types of state update messages processed by the spine scheduler in our testbed. Since leaf schedulers passively track the load by processing tasks and reply packets passing through them, there are no extra overheads for state updates inside the racks. We measure and plot the rate of messages normalized by the task rate across different workloads and worker settings in Figure 16. There are two types of messages sent to the spine layer. Selective Load Updates are sent to update the average load of the rack, and Idle Add/Remove messages are sent when the state of the rack changes from busy to idle and vice versa.

As shown in Figure 16, the rate of the Load Update messages at the highest workload is less than 0.15. Horus significantly reduces the overheads compared to previous works (e.g., [48, 60, 78]), which require at least processing one message per task. The figure also shows a small rate for the Idle Add/Remove messages. Similar to the fraction of resubmitted packets, the rate of messages is impacted by the placement of workers and the size of the worker pool. The number of workers in each rack impacts the rate of oscillation between the idle states as well as the rate of required average load updates.

Appendix C More Results from Simulation

This appendix includes additional results from our simulation.

Analysis of Horus Components. We analyze the contributions of the two components of the proposed scheduling policy: (i) scheduling tasks to idle nodes using idleness information and (ii) scheduling tasks to busy nodes using the power-of-two policy. In this experiment, we focus on five sample worker pools with sizes ranging from 50 to 20,000 workers each, and we use the Bimodal task distribution.

Algorithm 2 Scheduling and state updates for busy nodes

- loadList: reg. array that holds the queue length of nodes (two identical copies maintained).
- driftList: reg. array holding the difference between values in loadList and actual load (two identical copies maintained).

```
// On a taskReply or loadUpdate pkt received
function UPDATE(pkt)
    loadList1[pkt.srcID] ← pkt.qlen
    loadList2[pkt.srcID] ← pkt.qlen
    driftList1[pkt.srcID] ← 0
    driftList2[pkt.srcID] ← 0
// Atomic, read-modify-write operation on the reg.
function CHECKDRIFT(selectedIdx, diffSamples)
    if driftList1[selectedIdx] ≤ diffSamples then
        Increment driftList1[selectedIdx]
        return NORESUB
    else
        return driftList1[selectedIdx]
(A) On an task pkt received: First Path
function SCHEDULETASKBUSY(pkt)
    randIdx1, randIdx2 ← genRandomSamples()
    sample1 ← loadList1[randIdx1]
    sample2 ← loadList2[randIdx2]
    selectedIdx, diffSamples ← CompareSamples()
    driftSelected ← CheckDrift()
    if driftSelected == NORESUB then
        Increment driftList2[selectedIdx]
        Update pkt headers and Forward
    else
        load1 ← driftSelected + loadSelected
        load2 ← driftList2[otherIdx] + loadOther
        resubmit(selectedIdx, load1, otherIdx, load2)
(B) On an task pkt received: Resubmit Path
function SCHEDULETASKBUSY(selectedIdx, load1, otherIdx, load2)
    selectedIdx ← CompareSamples()
    Increment loadList1[selectedIdx]
    Increment loadList2[selectedIdx]
    Update pkt headers and Forward
```

We simulate two variants of the power-of-two policy. The first relies only on reply packets from workers to update the state, as done in RackSched [78]. This is referred to as Pow-of-2 Delayed Updated (DU). The second variant, which is used in Horus, updates the state while scheduling tasks, and it may require resubmitting packets through the switch. We also simulate a scheduler that uses the idle node selection only: it schedules tasks to idle nodes, and if there are no idle nodes, it will assign tasks to nodes randomly.

The results are presented in §C, where we present the average response time versus the system load. The results show

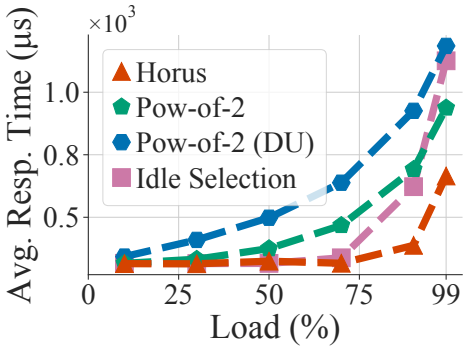


Fig. 17: Analysis of Horus components.

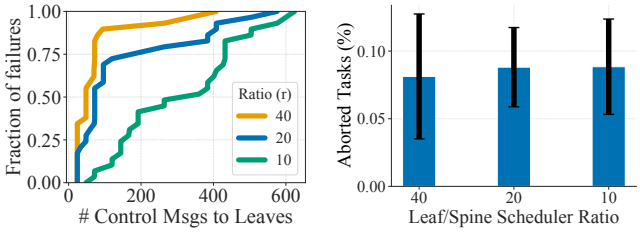
the impact and importance of the two components of Horus. That is, Horus achieves its performance by effectively tracking the load on nodes and assigning tasks to idle nodes whenever they are available. And when there are no idle nodes, Horus uses the power-of-2 policy with accurate load information to assign tasks to nodes with lower loads. Further, the randomness in the power-of-2 policy makes Horus robust against the task herding problem.

Impact of Scheduler Failures. We analyze the impact of spine scheduler failures while all of the 1K worker pools are running. Upon detecting a failure, the centralized controller notifies the leaf schedulers impacted by the failed spine scheduler. Similar to [38], we add latency to the control messages carrying the failure notices based on the number of hops between the failed spine and each impacted leaf. The experiment is repeated 30 times; each time, we fail a random spine scheduler. Notice that burst failures of spine switches are rare; the median time between failures of such switches is multiple hours [36]. Therefore, we only consider single spine failures.

We define r as the ratio of leaf to spine schedulers to control the number of spine schedulers per worker pool. For example, $r = 40$ indicates using four times fewer spine schedulers compared to $r = 10$, for the same number of leaf schedulers.

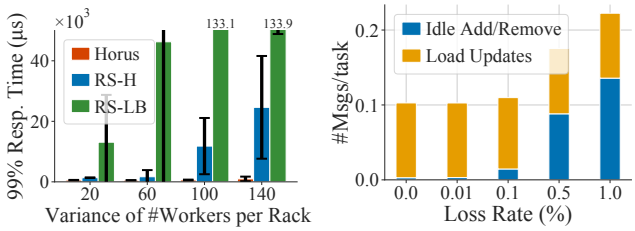
Figures 18a and 18b show the impact of spine scheduler failures and the trade-off for using different numbers of spine schedulers for worker pools. Figure 18a shows the number of messages sent from the centralized controller to the leaf switches as a result of the failure. When the state is distributed among more spine schedulers (i.e., small r), a failure results in more control messages sent. This is because the Horus controller sends a message to each leaf switch that needs to update its state. In the worst case ($r = 10$), an average of 306 (maximum 624) messages need to be sent for each spine failure event, which is a small message rate; centralized controllers in today's datacenters can send thousands of updates per second [59, 76].

Figure 18b shows the fraction of aborted scheduling tasks during spine switch failures. For all r values, less than 0.1% of the total submitted tasks are aborted on average during a



(a) Overhead during failure (b) Aborted tasks during failure

Fig. 18: Impact of spine scheduler failures.



(a) Worker placement (b) Packet losses

Fig. 19: Impact of worker placement and packet losses.

spine failure. This is because of the distributed scheduling in Horus, which enables multiple spine schedulers to handle the tasks submitted to a worker pool. Horus equally distributes the tasks belonging to a worker pool among spine schedulers. Therefore, before the failed switch is removed from the list of schedulers, only a small fraction of tasks that were sent to that switch may be affected. The aborted tasks can be re-launched by the application, which will be routed to the other active spine schedulers. Using more spine schedulers per worker pool provides better availability for the worker pools that were using the failed spine scheduler, because the scheduling requests of each worker pool are distributed uniformly among a larger number of spine schedulers.

Impact of Worker Placement. Datacenter operators may use different policies, e.g., [18], to allocate workers to worker pools, which can result in various worker distributions across racks. We analyze the impact of worker distribution on the performance of the task scheduler. Recall that we simulate 1K worker pools that have different numbers of workers (according to exponential distribution) randomly distributed across racks. We quantify the diversity in worker distribution by computing the variance in the number of workers per worker pool. A high variance indicates more scattered workers. We group worker pools that observed similar variance in the worker distributions together.

We plot, in Figure 19a, the average of the tail response time (at 90% load) observed by various worker pools. We also plot in the same figure, as error bars, the average plus/minus one standard deviation. The figure shows that as the variance in the worker distribution increases, RS-H and RS-R result in higher and more variable tail response times, whereas the response time of Horus remains stable across all worker distributions. For example, when the workers are highly scattered across racks (i.e., variance is 140), Horus reduces the tail response times by up to 94% and 99% compared to RS-R and RS-H, respectively.

The results in Figure 19a imply that the performance of Horus is robust against different worker distributions, which offers flexibility to datacenter operators to employ various worker allocation policies. The robustness of Horus is achieved by more accurately tracking the load on workers compared to RS-R and RS-H.

Impact of Packet Losses on Update Messages. Figure 19b shows the impact of packet loss on the rate of different types of update messages. Since Horus sends average load updates periodically based on the number of tasks that enter and exit the rack, the rate of load updates is not sensitive to packet losses. The *idleAdd* and *idleRemove* messages are, however, re-transmitted in case of loss. For example, when an *idleRemove* packet from a leaf to a spine is lost, it re-transmits the message until the leaf is removed from the idle list. This adds extra overhead on switches. As the figure shows, the rate of *idleAdd* and *idleRemove* messages increased by only 15% when the loss rate is 0.1%. Even with the much higher loss rates of 0.5 and 1%, Horus still functions properly, albeit at increased rates of *idleAdd* and *idleRemove* messages.