RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers

Hang Zhu, *Johns Hopkins University;* Kostis Kaffes, *Stanford University;*
Zixu Chen, *Johns Hopkins University;* Zhenming Liu, *College of William and Mary;*
Christos Kozyrakis, *Stanford University;* Ion Stoica, *UC Berkeley;* Xin Jin,
Johns Hopkins University

https://www.usenix.org/conference/osdi20/presentation/zhu

This paper is included in the Proceedings of the
14th USENIX Symposium on Operating Systems
Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the
14th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by USENIX

RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers

Hang Zhu
Johns Hopkins University

Kostis Kaffes
Stanford University

Zixu Chen
Johns Hopkins University

Zhenming Liu
College of William and Mary

Christos Kozyrakis
Stanford University

Ion Stoica
UC Berkeley

Xin Jin
Johns Hopkins University

Abstract

Low-latency online services have strict Service Level Objectives (SLOs) that require datacenter systems to support high throughput at microsecond-scale tail latency. Dataplane operating systems have been designed to *scale up* multi-core servers with minimal overhead for such SLOs. However, as application demands continue to increase, scaling up is not enough, and serving larger demands requires these systems to scale out to multiple servers in a rack.

We present RackSched, the first rack-level microsecond-scale scheduler that provides the abstraction of a rack-scale computer (i.e., a huge server with hundreds to thousands of cores) to an external service with network-system co-design. The core of RackSched is a two-layer scheduling framework that integrates inter-server scheduling in the top-of-rack (ToR) switch with intra-server scheduling in each server. We use a combination of analytical results and simulations to show that it provides near-optimal performance as centralized scheduling policies, and is robust for both low-dispersion and high-dispersion workloads. We design a custom switch data plane for the inter-server scheduler, which realizes power-of-k-choices, ensures request affinity, and tracks server loads accurately and efficiently. We implement a RackSched prototype on a cluster of commodity servers connected by a Barefoot Tofino switch. End-to-end experiments on a twelve-server testbed show that RackSched improves the throughput by up to 1.44×, and scales out the throughput near linearly, while maintaining the same tail latency as one server until the system is saturated.

1 Introduction

Online services such as search, social networking and e-commerce have strict end-to-end user-facing Service Level Objectives (SLOs) [12, 22]. To meet such SLOs, datacenter systems behind these services are expected to provide high throughput with low *tail latency* in the range of tens to hundreds of *microseconds* [12]. Example systems include key-value stores [6, 7, 60], transactional databases [9, 64], search ranking and sorting [13], microservices and function-as-a-service frameworks [17], and graph stores [43, 67].

With the end of Moore's law and Dennard's scaling, applications can no longer rely on single-threaded code to execute faster on newer processors with increased clock rates and instruction-level parallelism [31]. This leads to the rise of multi-core machines to scale up computation. Meeting microsecond-scale tail latency is challenging given that the request processing times with single-threaded code on bare metal hardware are already in the same time scale. This means that the operating system (OS) should impose minimal overhead when it manages resources and scales up these applications on multi-core machines.

This calls for new software architectures to efficiently utilize the resources of multi-core machines. One critical component of such architectures is scheduling. Dataplane operating systems have been designed to support low-latency applications with minimal overhead to meet strict SLOs [14, 39, 54, 58, 59]. For example, Shinjuku [39] uses efficient mechanisms to implement preemption and context switching, in order to realize *centralized* scheduling policies to avoid head-of-line blocking and address temporal load imbalance between multiple cores.

However, as application demands continue to increase, *scaling up* a single server is not enough. Serving larger demands requires these systems to *scale out* to multiple servers in a rack, which is termed as rack-scale computers, such as Berkeley Firebox [1], Intel Rack Scale Architecture [5], and HP "The Machine" [2]. Though previous efforts have not fully paned out yet, we believe it is inevitable, as evidenced by the emerging TPU Pods that pack high-density specialized hardware into a rack [8]. Even a traditional rack contains tens of servers and hundreds to thousands of cores, posing a challenge for scheduling microsecond-scale requests.

While dataplane operating systems address intra-server scheduling between multiple *cores*, head-of-line blocking and temporal load imbalance between multiple *servers* arise when the systems scale out. Using a single core for centralized scheduling and queue management is amenable for one server with a few to tens of cores [39]. But the core would quickly become the bottleneck if it were used to queue and schedule requests for a rack with hundreds to thousands of cores.

In this paper, we present RackSched, the first rack-level microsecond-scale scheduler that provides the abstraction of a rack-scale computer (i.e., a huge server with hundreds to thousands of cores) to an external service with network-system co-design. Serving microsecond-scale workloads is particularly challenging because the scheduler needs to *simultaneously* provide high scheduling speed (i.e., high throughput and low latency of the scheduler) and high scheduling quality (i.e., low tail latency to complete requests). Given the scheduling latency of modern OSes on a single server being a few microseconds [19, 39], our goal is to design a rack-level scheduler with comparable scheduling latency that can scale out to hundreds to thousands of cores in a rack. While there has been a long line of work on scheduling and load balancing, existing solutions are not designed for microsecond-scale workloads: software-based solutions [24, 27, 55, 56] suffer from low scheduling throughput and high scheduling latency (at least millisecond-scale); hardware-based ones [25, 51] are coarse-grained (based on five tuple) and server-agnostic, and thus suffer from long tail latency.

The core contribution of RackSched is a novel network-system co-design that *simultaneously* achieves high scheduling speed and high scheduling quality (§2). We propose a two-layer scheduling framework that integrates inter-server scheduling in the top-of-rack (ToR) switch with intra-server scheduling in each server to approximate centralized scheduling policies. This two-layer design, and the line-rate, on-path inter-server scheduling in the ToR switch, are critical for the scheduler to achieve high speed.

To provide high quality scheduling decisions, our key insight is that the two sources of long tail latency—load imbalance and head-of-line blocking—can be decoupled and handled by separate mechanisms. The ToR switch tracks real-time server loads, and schedules requests at *per-request* granularity to realize inter-server load balancing (LB). Each server keeps its own queue, and uses intra-server scheduling to preempt long requests that block pending short ones. It is known that centralized first-come-first-serve (cFCFS) is near-optimal for low-dispersion workloads, and processor sharing (PS) is near-optimal for heavy-tailed workloads or light-tailed workloads with high dispersion [39, 59, 69]. We use a combination of analytical results and simulations to show that our two-layer scheduling framework provides *near-optimal* performance as *centralized* policies, and is robust to different workloads (Figure 1).

Realizing the inter-server scheduler in the ToR switch requires the switch to schedule requests based on *server loads* on *per-request* granularity (§3). Today's stateful network load balancers map connections to servers based the hash of the five tuple [24, 25, 51, 53, 56], which is *static*, and cannot dynamically adapt the server selection under *microsecond-scale* load imbalance. There are three aspects of our approach to address this challenge. (i) We leverage the switch on-chip memory to store server loads, and use the multi-stage processing pipeline to implement power-of-k-choices and to support a variety of practical scheduling requirements, such as multi-queue policies. (ii) We design a *request state table* for *request affinity*, which guarantees the packets of the same request are sent to the same server. To maintain the dynamic mapping between requests and servers, the request state table supports *insert* (after scheduling the first packet), *read* (for sending following packets), and *remove* (when the request is completed) all in the *data plane*. (iii) We leverage *in-network telemetry* (INT) to monitor server loads with minimal overhead. Servers *piggyback* their load information in their *normal* traffic, and the switch tracks the latest reported load for each server.

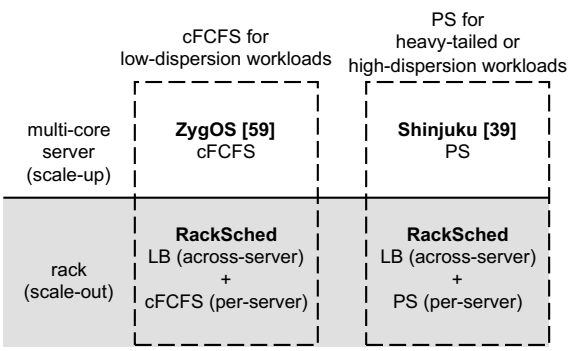


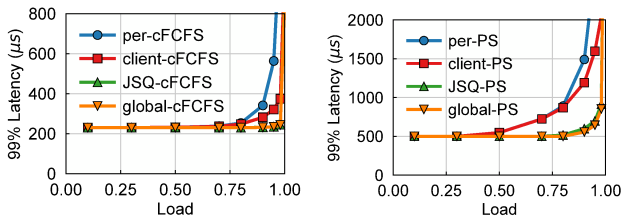
Figure 1: Key idea of RackSched.

Recent switch-based solution R2P2 [42] relies on expensive recirculation which does not scale for high request rate, and its scheduling policy has long tail latency under heavy-tailed or high-dispersion workloads due to head-of-line blocking (§4.5). In addition, R2P2 needs an extra round trip for multi-packet requests to ensure request affinity, while RackSched can finish in one round trip. RackSched is also more general in supporting many practical policies (§3.6).

In summary, we make the following contributions.

- We propose RackSched, the first rack-level microsecond-scale scheduler that provides the abstraction of a rack-scale computer to an external service.
- We design a two-layer scheduling framework that integrates inter-server scheduling in the ToR switch and intra-server scheduling in each server. We use a combination of analytical results and simulations to show that it provides near-optimal performance as centralized policies.
- We design a custom switch data plane for the inter-server scheduler, which realizes power-of-k-choices for near-optimal load balancing, ensures request affinity, and tracks server loads accurately and efficiently.
- We implement a RackSched prototype on a cluster of twelve commodity servers with a Barefoot Tofino switch. End-to-end experiments show that RackSched improves the throughput by up to 1.44×, and scales out the throughput near linearly, while maintaining the same tail latency as one server until the system is saturated.

The code of RackSched is open-source and available at `https://github.com/netx-repo/RackSched`.



(a) Low-dispersion workload. (b) High-dispersion workload.

Figure 2: Simulation results.

2 Design Decisions

In this section, we navigate through the design space of building a microsecond-scale scheduler for rack-scale computers, and highlight the design rationale of RackSched.

Scaling out to a rack. Supporting large application demands requires datacenter systems to scale out to multiple servers in a rack. While existing solutions like Shinjuku [39] solve the problem of scheduling requests to multiple cores (i.e., intra-server scheduling), they do not address the problem of scheduling requests to different servers (i.e., inter-server scheduling). When requests are simply scheduled to the servers randomly, the load imbalance and head-of-line blocking can happen at the server level, causing long tail latency for the entire system.

To motivate our work, we use simulations on representative workloads to show the impact of ineffective scheduling policies. We use the following two request service time distributions: (i) Exp(50) is an exponential distribution with mean = 50 μs, which is representative for low-dispersion workloads; (ii) Trimodal(33.3%-5, 33.3%-50, 33.3%-500) is a trimodal distribution with 33.3% of requests taking 5 μs, 33.3% taking 50 μs and 33.3% taking 500 μs, which is representative for high-dispersion workloads. The simulations assume eight servers and each server has eight workers (cores). The PS time slice used in the simulations is 25 μs.

We first compare the baseline policies that randomly send requests to servers and only use cFCFS or PS inside each server (per-cFCFS and per-PS) with the ideal centralized policies across all workers (global-cFCFS and global-PS). Figure 2 shows that the centralized policies perform better than the baseline policies. The tail latencies of per-cFCFS and per-PS quickly go up when the system load exceeds 0.75 and 0.5 respectively, while global-cFCFS and global-PS keep low tail latencies until the system is almost saturated.

Centralized scheduling cannot scale. The policy comparison in Figure 2 shows that there is a substantial gap between the tail latencies of the centralized policies (global-cFCFS and global-PS) and the baseline policies (per-cFCFS and per-PS). However, directly implementing the centralized policies is challenging because they would require a centralized scheduler for the entire rack. While a single core is capable of running a centralized scheduler to handle the requests for a multi-core server, it is unlikely to scale to a multi-server rack.

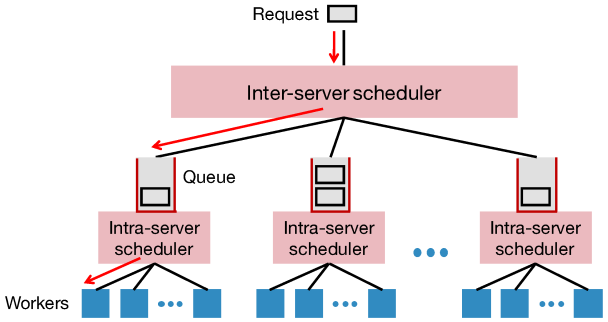


Figure 3: Two-layer scheduling framework.

Indeed, a single scheduler in Shinjuku [39] can scale to up to 11 cores, which falls well short of the demands of a rack with hundreds to thousands of cores.

Hierarchical scheduling. One natural solution to scale up the rack-scale scheduler is a two-layer hierarchical scheduler consisting of an inter-server scheduler at the high level, and per-server schedulers at the low level (Figure 3). This way, the inter-server scheduler only needs to schedule requests across tens of servers, instead of hundreds or thousands of cores. Each server employs its own intra-server scheduler to schedule requests across its cores.

Scaling the inter-server scheduler. While the inter-server scheduler only needs to schedule requests across the servers in the rack (instead of accross all cores), it still needs to handle every request. Assuming a rack with 1000 cores and 10 μs requests, the inter-server scheduler must handle up to 100 millions requests per second (MRPS) to saturate the rack! Unfortunately, such a scheduler would need to process a request every 10 ns, which exceeds the capability of a general-purpose computer.

To address this challenge, in this paper we propose to leverage emerging *programmable switches*, and have the ToR switch implement the inter-server scheduler. This design has the key benefit that the ToR switch is already on the path of the requests sent to the rack, and thus can readily process all these requests at line rate.

JSQ is near optimal and robust. A natural way to approximate a centralized scheduler with a two-layer scheduler is to implement the same scheduler at both layers. For example, if the global scheduler is cFCFS, in the corresponding hierarchical scheduler all the inter-server and intra-server schedulers will be cFCFS as well.

Unfortunately, the cFCFS scheduler needs to maintain a queue, and existing programmable switches have too limited memory to buffer requests, and are not well equipped to maintain dynamic data structures, such as queues. The join-the-shortest-queue (JSQ) scheduler can address the challenge because it is a bufferless scheduler. Upon a request arrival, it immediately forwards the request to the server with the shortest queue. This way, JSQ achieves fine-grained load balancing across the rack's servers. Figure 2 confirms that JSQ-cFCFS

and JSQ-PS can deliver nearly the same performance as the centralized policies (global-cFCFS and global-PS).

Theoretically, the two-layer scheduling framework implements the A/S/K/JSQ/P models in queueing theory, where A is the inter-arrival distribution, S is the service time distribution, K is the number of servers, JSQ is the join-the-shortest-queue policy implemented by the inter-server scheduler, and P is the intra-server scheduling policy which is either cFCFS or PS in this case. In particular, it is known that JSQ provides near-optimal load balancing, and importantly, is *robust* against request service time distributions. An expanded discussion is in the technical report [74].

Approximating JSQ. While conceptually simple, JSQ cannot be implemented in its definite form in practice, because it requires the switch to know the exact queue length of each server when scheduling a request. It takes a round trip time for the switch to ask each server, during which the queue lengths may have changed for microsecond-scale workloads. Furthermore, imperfect JSQ based on delayed server status is prone to *herding*, where several consecutive requests are sent to the same server before the server load is updated, and this can generate micro bursts and degrade system performance. Note that herding here is not caused by multiple asynchronous load balancers as there is only one load balancer (the inter-server scheduler), but from stale server load information in the load balancer. In addition, the switch can only do a limited number of operations for each request, and finding the shortest queue cannot be implemented for many tens of servers. Thus, we use power-of-k-choices to approximate JSQ, which samples k servers for each request and chooses the one with the minimum load. This approximation provides comparable performance as JSQ in theory [18] (see [74]), and handles these practical limitations well.

Why not a distributed, client-based solution? An alternative solution is to implement distributed scheduling at each client. The clients can use JSQ, power-of-k-choices or more complicated solutions like C3 [63] to pick workers for their requests. Such a client-based solution has two drawbacks. First, it needs client modification and increases system complexity. The clients need to probe server loads and make scheduling decisions. More importantly, the clients need to be notified for *every* system reconfiguration (e.g., adding or removing servers), because they have to know which set of servers a request can be sent to. Notifying a large number of clients of the latest system configuration imposes both a consistency challenge and high system overhead. Putting these functionalities in a scheduler, on the other hand, simplifies the clients and avoids these system complexities.

Second, a distributed, client-based solution provides a worse trade-off between performance and probing overhead than a centralized scheduler for microsecond-scale workloads. This is because microsecond-scale workloads are IO-intensive, and a probing request incurs comparable processing cost as a normal request at the servers. Thus, probing needs to be minimized to improve the throughput of processing the actual requests. No matter whether probing is done proactively or piggybacked in reply packets, given n clients with the same sending rate, a centralized scheduler can utilize n times as much probing data as that of one client in a client-based solution, resulting in better scheduling quality. Figure 2 confirms the benefit of the centralized scheduler over a client-based solution with a piggyback-based probing mechanism (client-cFCFS and client-PS). The simulation does not model the client software delay and the network delay to get the server loads, which favors the client-based solution. The client-based solution performs worse in real experiments (§4.5). In a multi-pipeline switch, though states are not shared across pipelines, RackSched can approximate JSQ within each pipeline. It works better than the client-based solution because the number of pipelines (e.g., 4) is far smaller than the number of clients (e.g., 1000 or more).

Putting it all together. We propose a two-layer scheduling framework that integrates inter-server scheduling in the ToR switch and intra-server scheduling in each server. The ToR switch uses power-of-k-choices to achieve inter-server load balancing, and each server uses cFCFS or PS to minimize head-of-line blocking. This solution approximates centralized scheduling for the entire rack, and provides the abstraction of a rack-scale computer: the capacity (throughput) of the rack-scale computer is the sum of that of its servers, and the tail latency is maintained as that of one server.

Challenges. Translating the two-layer scheduling framework to a working system implementation presents several technical challenges:

- What is the system architecture to realize this two-layer scheduling framework?
- How does the switch schedule requests based on the server loads, and handle practical scheduling requirements?
- How does the system ensure request affinity (i.e., the packets of the same request are sent to the same server), when the switch processes each packet independently?
- How do servers expose their states to the switch so that the switch can track the real-time loads on the servers efficiently and accurately?

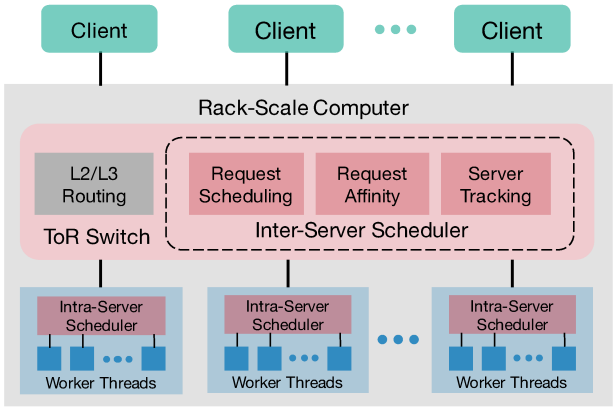
3 RackSched Design

In this section, we present the design of RackSched to address the challenges. We first give an overview of the system architecture, and then describe each component in detail.

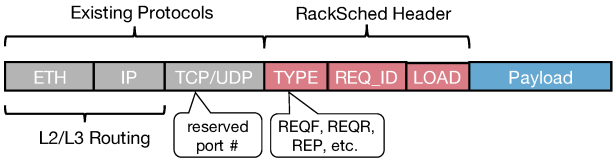
3.1 System Architecture

The core of RackSched is a two-layer scheduling framework that combines inter-server scheduling and intra-server scheduling. Figure 4(a) shows the RackSched architecture.

Inter-server scheduling. The ToR switch performs inter-server scheduling at per-request granularity via three modules:



(a) RackSched architecture.



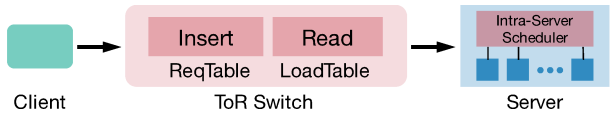
(b) RackSched packet format.

Figure 4: RackSched overview.

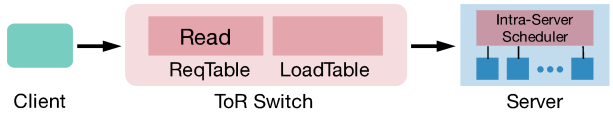
a request scheduling module that selects a server for a new incoming request based on server loads (§3.3) and scheduling requirements (§3.6), a request affinity module that forwards the packets of the same request to the same selected server (§3.4), and a server tracking module that tracks the real-time load on each server (§3.5). All three modules are implemented in the switch data plane that enables the inter-server scheduler to run at line rate.

Intra-server scheduling. Each multi-core server in the rack runs multiple worker threads to process requests. Each server has a centralized scheduler to queue and schedule requests to its own workers. The scheduler implements centralized scheduling policies for intra-server scheduling. Each server also piggybacks its load information in reply messages to report its load to the ToR switch.

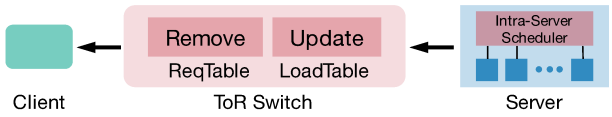
Deployment options. There are two deployment options for RackSched. (*i*) The first one is to integrate RackSched with the ToR switch of a rack-scale computer. This option adds additional functionalities to the ToR switch, but does not change any other part of the datacenter network. (*ii*) The second option is to treat the switch-based scheduler as a specialized server with a programmable switching ASIC. This server can be connected to the same ToR switch as worker servers. It owns the anycast IP address so all requests would be first sent to it for scheduling. By properly updating the addresses, it can also force the reply packets to pass through it before returning to the clients, in order to clear request states and update server loads. This option does not even modify the ToR switch, but has the latency cost of an extra hop by the detour to the switch-based scheduler.



(a) The first packet of a request.



(b) The following packets of the request.



(c) The reply packets.

Figure 5: Request processing in RackSched.

3.2 Request Processing

Network protocol. RackSched is designed for intra-datacenter scenarios. It uses an application-layer protocol which is embedded inside the L4 payload. We reserve an L4 port to distinguish RackSched packets from other packets. The network uses existing L2/L3 routing protocols to forward packets. There are no modifications to the switches in the network other than the ToR switch. The ToR switch uses the reserved L4 port to invoke the custom modules to process RackSched packets. Other switches do not need to understand the RackSched protocol, nor do they need to process RackSched packets. RackSched is orthogonal to and compatible with other network functionalities, such as flow/congestion control, which is typically implemented by the transport layer or the RPC layer (e.g., eRPC [41]). RackSched ensures request affinity under packet loss and retransmission by maintaining connection state (§3.4).

RackSched only requires the applications to add the RackSched header between the TCP/UDP header and the payload (Figure 4(b)), so that it can make scheduling decisions based on the RackSched header. Note that for TCP handshake packets that do not have any payload, the RackSched header should be appended after the TCP header to expose necessary information to the switch. We emphasize that RackSched focuses on microsecond-scale workloads. It is not intended to support long-lived TCP connections, which impose unnecessary system overhead to maintain connection states, especially under switch failures (§3.4), and restrict the scheduler from making per-request scheduling decisions to address temporal load imbalance. RackSched does support request dependency for tasks with multiple requests (§3.6).

The RackSched header contains three major fields, which are `TYPE`, `REQ_ID`, and `LOAD`. `TYPE` indicates the type of the packet, e.g., `REQF` (the first packet of a request), `REQR` (a remaining packet of a request), and `REP` (a reply packet).

Algorithm 1 ProcessPacket(pkt)

– *ReqTable*: on-chip memory for request-server mapping
– *LoadTable*: on-chip memory for server loads

// first packet of a request
1: **if** $pkt.type == REQF$ **then**
2: $server\_ip \leftarrow LoadTable.select\_server()$
3: $ReqTable.insert(pkt.req\_id, server\_ip)$
// remaining packets of a request
4: **else if** $pkt.type == REQR$ **then**
5: $server\_ip \leftarrow ReqTable.read(pkt.req\_id)$
// reply packets
6: **else if** $pkt.type == REP$ **then**
7: $ReqTable.remove(pkt.req\_id)$
8: $LoadTable.update(pkt.srcip, pkt.load)$
9: Update packet header and forward

REQ_ID is a unique ID for each request. All packets of the same request and the corresponding reply use the same REQ_ID. To ensure a REQ_ID is globally unique, each client appends its client ID in front of its locally generated unique request ID, i.e., a tuple of <client ID, local request ID>. LOAD indicates the load of the server. The server piggybacks its current queue length in the LOAD field in reply packets. LOAD is not used in request packets.

Processing request packets. Clients use an *anycast* IP address as the destination IP to send requests to the rack-scale computer, and are unaware of the number of servers behind the ToR switch. Figure 5 illustrates how RackSched processes packets, and Algorithm 1 shows the high-level pseudo code of the switch. The switch keeps two essential sets of state in the switch on-chip memory. One is *ReqTable* which stores the mapping from the request IDs to the servers, and the other is *LoadTable* which stores the queue lengths of the servers. As shown in Figure 5(a), when the first packet of a request arrives at the switch, the switch selects a server based on *LoadTable*, and remembers this selection by inserting an entry to *ReqTable* (line 1-3). Then in Figure 5(b), when remaining packets of the request arrive, the switch checks the *ReqTable* to get the selected server (line 4-5), which ensures request affinity. The switch uses the selected server IP to update the destination IP in the packet header and sends the packet to the corresponding server (line 9).

Processing reply packets. After a server receives a request, it uses its local scheduler to schedule and processes the request. Then the server generates a reply, and sets the LOAD field with its current queue length. As shown in Figure 5(c), the switch deletes the mapping from *ReqTable*, because the request has completed and the memory space can be freed for other requests (line 7). The switch also updates the server load in *LoadTable* based on the LOAD field (line 8). We do not distinguish the first and following packets for a reply even if the reply contains *multiple* packets (also equivalent to *multiple* replies). Because *ReqTable* checks *req_id* for deletion, if a slot is reused by another request, the following reply packets of the previous request would not be applied. The updates of *LoadTable* only affect server selection of new requests. Note that this is compatible with TCP even if the client initiates the termination of the connection, as the mapping of this request is removed from *ReqTable* when the server receives the request and sends the first reply packet back to the client. The switch control plane periodically checks the data plane to remove *stale* mappings from *ReqTable*, which can be caused by server failures or lost reply packets. In the end, the switch updates the source IP to the anycast IP in the packet header, and sends the packet back to the client (line 9).

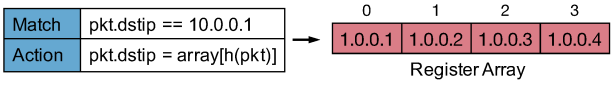


Figure 6: Traditional hash-based random dispatching.

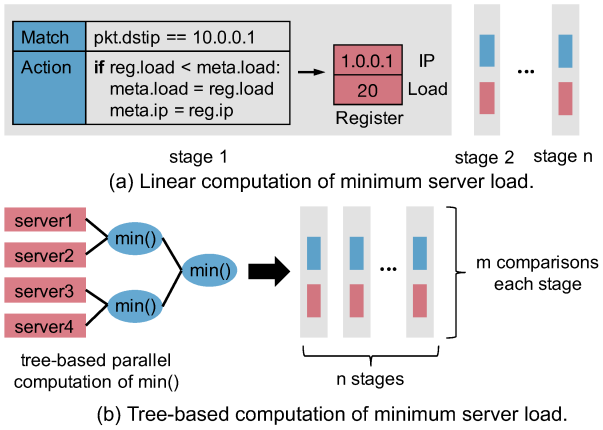


Figure 7: Optimal server selection.

3.3 Request Scheduling

The request scheduling module dynamically schedules requests based on server loads. Unfortunately, this is not supported in today's switches. Today's switch-based load balancers such as SilkRoad [51] only support ECMP-like random dispatching based on the five tuple. Figure 6 shows how hash-based random selection is implemented in the data plane. The register array stores a set of server IPs for the anycast IP 10.0.0.1. The rule in the match-action table matches packets with their destination IP being the anycast IP 10.0.0.1, and the action rewrites the destination IP to an IP in the register array, which is selected by computing a hash on the packet header (usually the five tuple). Because the selection is static, and is purely based on the hash, it can cause load imbalance and long tail latency as discussed in §2. We now describe how to realize dynamic request scheduling based on server loads. Handling practical scheduling requirements is in §3.6.

Optimal server selection. We leverage the register arrays to store the server loads together with the server IPs and use the multi-stage packet processing pipeline to compute the minimum. A naive way is to use multiple stages to scan the

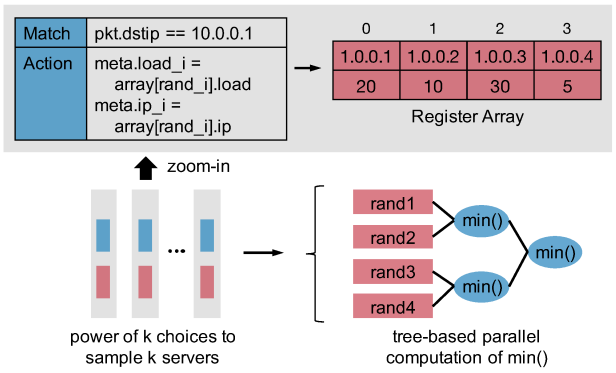


Figure 8: Approximate server selection.

server loads linearly, as shown in Figure 7(a). The number of servers this solution can support is limited to the number of stages. As a switch typically has 10–20 stages and many stages need to be used by other switch functionalities, this solution is not scalable. It can be optimized by computing the minimum in a tree structure as shown in Figure 7(b). The comparisons between two servers in each layer of the tree have no dependencies, and thus can be done in parallel in the same stage. In the ideal case, given n servers, this tree-based solution requires $log(n)$ stages, while the naive solution requires n stages. Let each stage support up to m comparisons. The comparisons in the first few layers need to be distributed to multiple stages, if they are larger than m.

Approximate server selection. As discussed in §2, always choosing the server with the shortest queue is prone to herding, and due to the limited stages and the need to support other functionalities, the tree-based approach cannot scale to many tens of servers. We design an approximate server selection mechanism based on power-of-k-choices [18], i.e., the switch samples k servers and chooses the one with the shortest queue from them. As shown in Figure 8, the sampling can be done via multiple stages if k is bigger than the number of register read operations supported by one stage. After the k servers are sampled, the tree-based mechanism can be applied to get the one with the shortest queue.

3.4 Request Affinity

Request affinity ensures all packets of the same request are sent to the same server. This is challenging because the switch processes each packet independently. In traditional network load balancers [24, 25, 51, 53, 56], the server selection is solely based on the hash of the packet header, and the switch does not need to keep any state for request affinity. But in RackSched, the selection is dynamic. If the switch performs a server selection for every packet, the packets of the same request might be sent to different servers.

Realizing request affinity requires the switch to keep states. Abstractly, the switch should maintain a request state table to store the mapping from request IDs to server IPs (i.e., *ReqTable* in Algorithm 1). One option is to use a match-action table, where the request IDs are stored in the match, and the servers IPs are stored in the action to update the destination IP of the packets (e.g., used by SilkRoad [51]). This option, however, does not work for microsecond-scale requests at million RPS throughput, because updating the match-action table (e.g., adding or removing a request) requires the control plane, which can only do about 10K updates per second [36, 48, 52]. To address this challenge, our design leverages register arrays to realize a multi-stage hash table that implements all necessary operations (i.e., *insert*, *read* and *remove*) for *ReqTable* in the data plane, as shown in Figure 9. Unlike match-action tables, register arrays can only be accessed via an index. We use the hash of the request ID to find the slot for a request, and the slot stores the request state, i.e., the request ID and server IP. To handle hash collisions and the limited array size in each stage, we leverage multiple stages to build a multi-stage hash table. Algorithm 2 shows the pseudo code to implement the three operations on *ReqTable* in Algorithm 1. The switch iterates over the stages to find an empty slot to insert a new request (line 1-5), and to find a matched slot to read the server IP (line 6-9) or remove a completed request (line 10-14). RackSched does not decrease the capability of the system to defend against DoS attacks. The switch has sufficient memory for *ReqTable* to support high throughput (§4.1), and a DoS attack that overwhelms *ReqTable* could have overwhelmed the servers first.

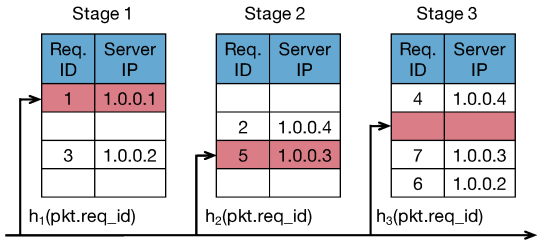


Figure 9: Multi-stage hash table for request affinity.

Handling switch failure. There is a relevant notion to request affinity called Per-Connection Consistency (PCC) for stateful layer-4 load balancers, which requires a TCP connection to be kept across load balancer failures and system reconfigurations [24, 25, 51, 53, 56]. We emphasize that RackSched focuses on microsecond-scale requests with strict deadlines (e.g., a couple of the request execution time). Rebooting a failed switch or replacing it with a backup switch takes a few minutes, by the time of which the requests have already *missed* their deadlines. Therefore, different from PCC, maintaining request affinity across switch failures is a non-goal for RackSched. Because of the *fate sharing* between the ToR switch and the rack, it is safe to disregard the *ReqTable* upon a switch failure, and the new switch starts with an empty *ReqTable*. Note that RackSched does not increase the chance of switch failures as a normal ToR switch without RackSched can still fail and make the rack disconnected.

Algorithm 2 Request affinity

– *ReqTable*[*n*][*m*]: register arrays to store request state, which spans *n* stages and has *m* slots in each stage

```
1: function INSERT(req_id, server_ip)
2:     for stage i in all stages do
3:         if ReqTable[i][h(req_id)] == None then
4:             ReqTable[i][h(req_id)] ← (req_id, server_ip)
5:             return
6: function READ(req_id)
7:     for stage i in all stages do
8:         if ReqTable[i][h(req_id)].req_id == req_id then
9:             return ReqTable[i][h(req_id)].server_ip
10: function REMOVE(req_id)
11:     for stage i in all stages do
12:         if ReqTable[i][h(req_id)].req_id == req_id then
13:             ReqTable[i][h(req_id)] ← None
14:     return
```

Handling system reconfiguration. Unlike switch failures, there is no fate sharing between each server and the rack, and RackSched does maintain request affinity across system reconfigurations such as adding or removing servers for an application. Because RackSched uses *ReqTable* to store the mapping, ongoing requests simply check *ReqTable* to go to the correct servers. Only the request scheduling module (§ 3.3) needs to be updated to have the right set of servers to choose from for new requests. We pre-allocate a large number of registers for *LoadTable* at compilation time, and use another register to indicate the number of *active* servers, which is dynamically updated for system reconfigurations. For an unplanned server removal (e.g., a server failure), RackSched uses the switch control plane to update the *ReqTable* and delete the stale entries related to the removed server.

3.5 Server Tracking

The server tracking module updates the server loads (i.e., *LoadTable* in Algorithm 1) for the switch to make scheduling decisions. The challenge is to accurately track the server loads at real-time with low overhead. A straightforward solution is to let the switch control plane periodically poll the queue lengths at each server and update the data plane. However, due to the millisecond-scale delay and the limited rate of control plane updates [37, 52], this solution does not apply to the microsecond-scale workloads targeted by RackSched. To do this in the data plane, a possible solution is to let the switch *proactively* track the server loads, i.e., incrementing and decrementing the counters for queue lengths when processing request and reply packets. This solution suffers from estimation errors due to packet loss and retransmissions, and fixes like decreasing the counters based on the server processing rate [47] cannot handle temporal load imbalance in high-dispersion microsecond-scale workloads.

RackSched leverages in-network telemetry to accurately track server loads with minimal overhead. In-network telemetry is widely used in network monitoring and diagnosis where switches put relevant measurement data into packet headers in the data plane. RackSched applies this mechanism to track server loads. Then the servers piggyback their loads in the reply packets to update the counters in the switch, which does not introduce new packets and thus minimizes system overhead. A potential problem is the feedback loop delay as stale information can cause herding, which can degrade the scheduling performance and make the system unstable (i.e., swing between overloading different servers). In RackSched, the switch and the servers are directly connected in the same rack, and the server-side data plane implementation bypasses traditional TCP/IP stack to report its queue length to the switch quickly, making the feedback loop delay minimal. And together with power-of-k-choices scheduling, RackSched can effectively avoid herding. An alternative solution that only keeps the server with the minimum load in the switch and updates it based on in-network telemetry cannot leverage power-of-k-choices to avoid herding. We show the impact of different ways to track and represent server loads in §4.6.

3.6 Handling Scheduling Requirements

Multi-queue support. By default, RackSched uses a single-queue policy, i.e., the system does not differentiate a priori between request types and aims to meet a single SLO for tail latency (e.g., a photo caching workload with only *get* requests). RackSched also supports multiple queues if the workload has multiple request types that have distinct service time distributions (e.g., a key-value store workload with both *get* and *range* requests). Applications indicate the request type in the `TYPE` field of the packet header. Each server maintains a separate queue for each type for intra-server scheduling. The switch maintains the counters for each type in *LoadTable*, and schedules requests based on the queue lengths of the request type. We remark that there is no fundamental limit on the number of queues on each server, since the queues are implemented in software and the switch only needs to keep a counter for each queue on each server.

Locality and placement constraints. RackSched handles two types of common locality and placement constraints, which are data locality and request dependency. (i) Data locality requires a request to be processed on a subset of servers that hold the input data. To support data locality, applications set different `LOCALITY` values to represent different locality requirements (i.e., different sets of servers that can process this request), and the switch maintains different mappings, which map a server ID to a server for different `LOCALITY` values. (ii) Request dependency requires multiple requests to be scheduled to the same server, e.g., a task consists of multiple requests and the input of one request is the output of one or more other tasks. Request dependency is supported using request affinity: relevant requests carry the same `REQ_ID` in their headers, so that they will be sent to the same server. Additional information is included in the RackSched header for the number of subsequent requests to expect. The server can send replies to each request separately and independently.

RackSched only requires the server to set `TYPE` to be reply in replies after it has received all requests in the set in order to safely delete the state in the switch. Note that applications can still use different request IDs for different requests and receive replies as soon as some requests are completed, by adding application-specific metadata in the payload.

Resource allocation policies. The scheduler is responsible for allocating resources when the demand exceeds the capacity of the rack-scale computer. RackSched supports two types of common resource allocation policies, which are strict priority and weighted fair sharing. (i) To support strict priority, each server maintains a separate queue for each priority. Similar to the multi-queue support, the switch tracks the queue lengths, and balances the server loads for each priority. Each server uses intra-server scheduling to preempt low-priority requests when high-priority requests arrive, which can be done in 5 μs in our implementation based on Shinjuku [39]. (ii) Supporting weighted fair sharing is similar. Each server maintains a separate queue for each client, and performs weighted fair queueing [23] for intra-server scheduling on the granularity of slice in PS. The switch tracks the queue lengths and balances the server loads for each client.

4 Evaluation

In this section, we evaluate RackSched with a variety of synthetic and real application workloads. We provide additional experiment results, including locality constraints, priority policies and multiple applications, in the technical report [74].

4.1 Methodology

Testbed. The experiments are conducted on a testbed of twelve server machines connected by a 6.5Tbps Barefoot Tofino switch. Each server has an 8-core CPU (Intel Xeon E5-2620 @ 2.1GHz), 64GB memory, and one 40G NIC (Intel XL710). Eight servers are used as workers to process requests, and they run Shinjuku [39] with our extension. Four servers are used as clients to generate requests. The bottleneck of the system is at the workers.

Implementation. We have implemented a RackSched prototype and integrated it with Shinjuku [39]. (i) The switch data plane is written in P4 [16] and compiled to Barefoot Tofino ASIC [11] with P4 Studio [10]. The request state table contains a hash table with 64K slots. The default implementation uses power-of-2-choices. (ii) The worker server is based on Shinjuku [39]. We have extended Shinjuku to support the RackSched packet header, maintain a counter and update the counter upon request arrival and reply departure to track the queue length and append the queue length in reply packets. Both RackSched and Shinjuku preempt requests that exceed 250 μs in our experiments. (iii) The client is open-loop and implemented in C using Intel DPDK 16.11.1 [4]. It can generate requests at high request rate based on synthetic workloads and the RocksDB application, and measure the throughput and latency of RackSched.

Resource consumption. Our prototype uses 13.12% SRAM, 9.96% Match Input Crossbar, 12.5% Hash Unit and 25% Stateful ALUs of the Tofino ASIC resources. We provide an analysis and a back-of-the-envelop calculation to show that RackSched consumes little switch memory. RackSched has two sets of state on the switch, i.e., *LoadTable* and *ReqTable*. (i) *LoadTable* maintains a counter for each queue of each server. Let the counter be 4 bytes, the number of queues in each server be 3 and the number of servers be 32. It only consumes 384 bytes. (ii) *ReqTable* maintains the selected server IPs for the *ongoing* requests, not *all* requests the system have received and processed. Each slot can be *reused* by many times each second because the requests are microsecond-scale. Given an average request processing latency of 50 μs, a slot can support 20 KRPS throughput, and a table with 64K slots can support 1.28 BRPS throughput. Let the request ID and server IP both be 4 bytes. A table with 64K slots (our implementation) consumes 256 KB, which is only a few percent of the on-chip memory (tens of MB). Overflowed requests can fall back to hash-based random dispatching which preserves request affinity.

Workloads. We use a combination of synthetic and application workloads. They include the following workloads. By default, the workloads use one-packet requests.

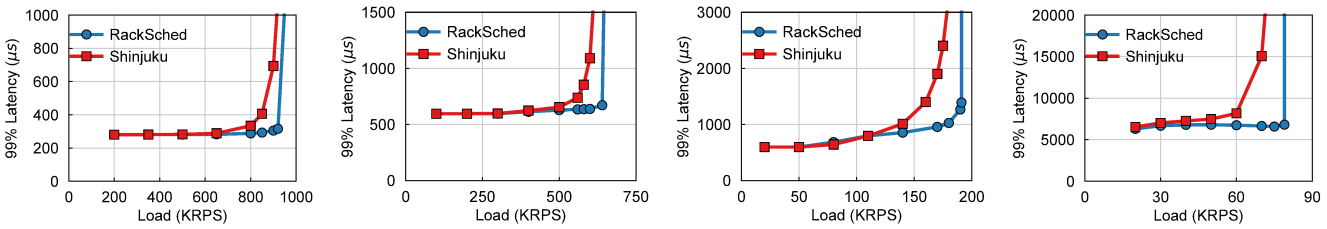
- Exp(50) is an exponential distribution with mean = 50 μs, which represents common query and storage workloads, such as *get* requests in photo caching.
- Bimodal(90%-50, 10%-500) is a bimodal distribution with 90% of requests taking 50 μs and 10% taking 500 μs, which represents workloads with a mix of simple requests and complex requests, such as *get* and *range* requests in key-value stores.
- Bimodal(50%-50, 50%-500) is a bimodal distribution with 50% of requests taking 50 μs and 50% taking 500 μs, which represents workloads with half simple requests and half complex requests.
- Trimodal(33.3%-50, 33.3%-500, 33.3%-5000) is a trimodal distribution with a third of requests taking 50 μs, 500 μs and 5000 μs, respectively, which represents workloads with more diverse request types, such as *point*, *range* and complex *join* requests in databases.

We also use RocksDB 5.13 [60], an open-source production-quality key-value store, as a real application workload to evaluate RackSched. RocksDB is configured to store data in DRAM to avoid blocking behavior and achieve low latency.

4.2 Synthetic Workloads

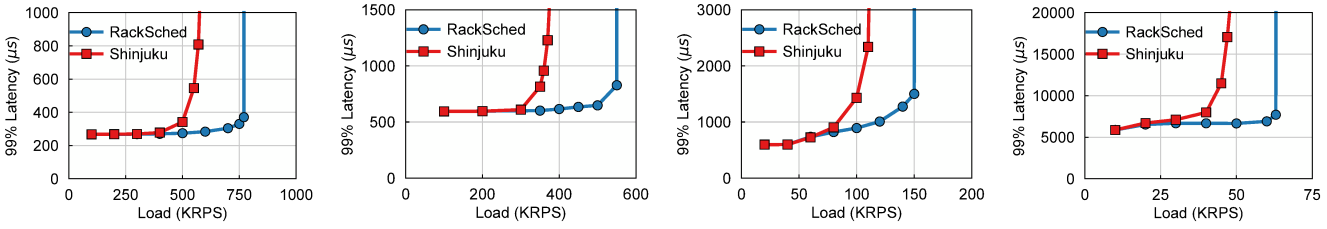
We evaluate the system on synthetic workloads that cover large application space. We compare RackSched with that directly runs Shinjuku in the cluster, i.e., the requests are randomly sent to the servers.

Figure 10(a) and Figure 10(b) compare RackSched and Shinjuku under Exp(50) and Bimodal(90%-50, 10%-500) workloads, respectively. In these two figures, both RackSched



(a) Exp(50). (b) Bimodal(90%-50, 10%-500). (c) Bimodal(50%-50, 50%-500). (d) Trimodal(33.3%-50, 33.3%-500, 33.3%-5000).

Figure 10: Experimental results for synthetic workloads with homogeneous servers.



(a) Exp(50). (b) Bimodal(90%-50, 10%-500). (c) Bimodal(50%-50, 50%-500). (d) Trimodal(33.3%-50, 33.3%-500, 33.3%-5000).

Figure 11: Experimental results for synthetic workloads with heterogeneous servers.

and Shinjuku use a single-queue policy. Under Exp(50), the 99% latencies of RackSched and Shinjuku are similar at low load. But the 99% latency of Shinjuku quickly goes up after 800 KRPS, while that of RackSched can support the system load up to 950 KRPS. Under Bimodal(90%-50, 10%-500), the 99% latency of Shinjuku quickly increases after 500 KRPS, while that of RackSched stays stable until 650 KRPS. In both workloads, RackSched supports larger request load with lower tail latency, because its inter-server scheduling addresses temporal load imbalance between servers, while Shinjuku experiences short bursts and long queues in individual servers under high request load.

Figure 10(c) and Figure 10(d) show the results for Bimodal(50%-50, 50%-500) and Trimodal(33.3%-50, 33.3%-500, 33.3%-5000) workloads, respectively. In these two figures, both RackSched and Shinjuku have a separate queue for each request type. Again, RackSched significantly outperforms Shinjuku. The improvement of RackSched is larger in Trimodal(33.3%-50, 33.3%-500, 33.3%-5000) than Bimodal(50%-50, 50%-500), because Trimodal(33.3%-50, 33.3%-500, 33.3%-5000) has more diverse service times and can benefit more from effective inter-server scheduling.

Figure 11 shows the results with heterogeneous servers. In this case, four servers have four workers and the other four servers have seven workers (one core used by the scheduler). This evaluates the cases when some servers are slower or some cores of these servers are grabbed for other purposes [15, 54]. We compare RackSched with Shinjuku under the same four distributions in Figure 10. RackSched is load-aware and tends to send requests to the servers with shorter queue lengths, while Shinjuku distributes the requests to the servers uniformly, disregarding the heterogeneity. RackSched can improve the performance further with heterogeneous servers.

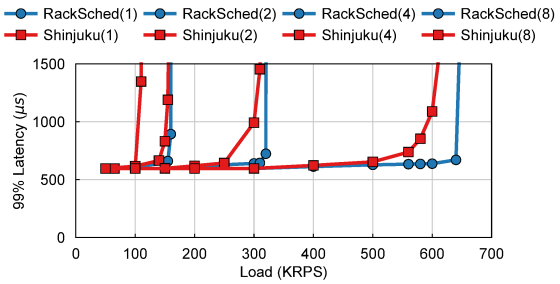
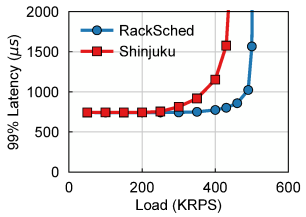


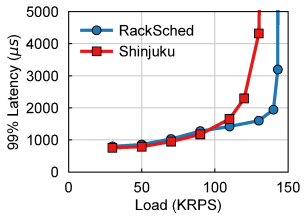
Figure 12: Scalability results.

4.3 Scalability

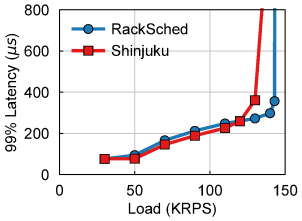
The key benefit of RackSched is that it enables the system to scale out by adding servers, while achieving low tail latency at high throughput. Figure 12 shows the 99% latency under different request load with one, two, four and eight servers, respectively. This figure uses Bimodal(90%-50, 10%-500) workload, and the results for other workloads are similar. With one server, the two systems, i.e., RackSched (1) and Shinjuku(1), have the same performance, as there is no need for inter-server scheduling. With two servers, load imbalance can happen, but the variability is small. With four servers, micro bursts can cause bigger temporal load imbalance, and the improvement of inter-server scheduling is also bigger. When there are eight servers, there is more variability between the loads on the servers, and inter-server scheduling has more opportunities to improve performance. Shinjuku(8) can only maintain low tail latency until 500 KRPS, while RackSched (8) can maintain low tail latency until 650 KRPS. We expect the improvement of RackSched over Shinjuku would be larger with more servers, because there would be more variabilities with more servers.



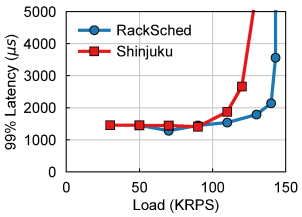
(a) 90%-GET, 10%-SCAN.



(b) 50%-GET, 50%-SCAN.



(c) GET in 50%-GET, 50%-SCAN.



(d) SCAN in 50%-GET, 50%-SCAN.

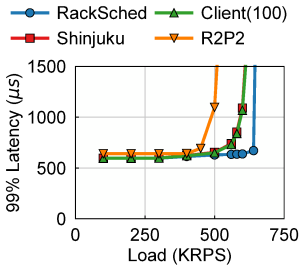
Figure 13: Experimental results for RocksDB.

Overall, RackSched scales out the total throughput of the system near linearly with the number of servers in the rack. And the throughput improvement is achieved without increasing the tail latency. Even with more servers, RackSched is still able to maintain the same tail latency as one server until the system is saturated.

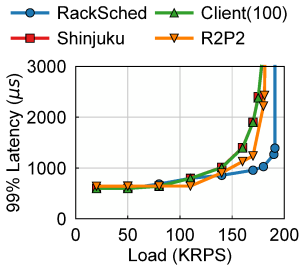
4.4 Application: RocksDB

We use RocksDB [60] to demonstrate the benefits of RackSched on real applications. RocksDB is an open-source production-quality storage system that is widely deployed to support many online services such as Facebook. In the experiments, RocksDB is configured with an in-memory file system (/tmpfs/) for microsecond-scale request processing. We use two request types. One is *GET* which gets 60 objects with a median request service time of 50 μs. The other is *SCAN* which scans 5000 objects with a median service processing time of 740 μs. Only 326 lines of code are needed to port RocksDB to RackSched and Shinjuku. Figure 13(a) shows the results for the workload that contains 90% *GET* requests and 10% *SCAN* requests. In this experiment, the system uses a single-queue policy. At low request load, RackSched and Shinjuku have comparable 99% latency. But Shinjuku can only maintain low tail latency until 300 KRPS, while RackSched is able to keep low tail latency until 500 KRPS.

Figure 13(b) shows the results for the workload that contains 50% *GET* requests and 50% *SCAN* requests. In this experiment, the system uses a multi-queue policy. RackSched is able to maintain low tail latency at a higher request load than Shinjuku. We further break down the results for each request type for this workload. Figure 13(c) and Figure 13(d) show the 99% latency for *GET* and *SCAN* under different total request load, respectively. Because RackSched uses the switch to balance the load of each request type between the



(a) Bimodal(90%-50,10%-500).



(b) Bimodal(50%-50,50%-500).

Figure 14: Comparison with other solutions.

servers, the improvement of RackSched over all requests does not come at the cost of sacrificing any individual request type. For both request types, RackSched is able to deliver comparable tail latency at low load, achieve significantly lower tail latency at high load, and support higher total request load.

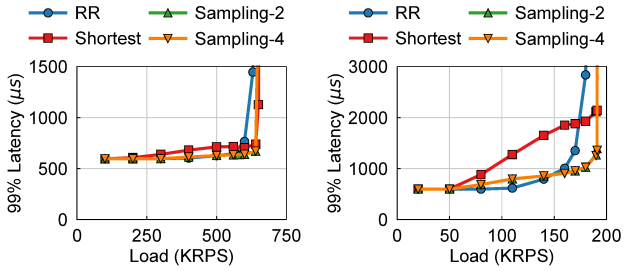
4.5 Comparison with Other Solutions

R2P2 [42] is a recent solution that proposes a join-bounded-shortest-queue (JBSQ) policy for request scheduling, and the solution can be implemented on programmable switches. R2P2 does not have preemptive intra-server scheduling and has head-of-line blocking. Thus, it suffers from long tail latency, especially under high-dispersion workloads. Client-based solutions are lack of global view and use power-of-k-choices scheduling based on stale server load information, and thus they suffer from inaccurate scheduling decisions. We emulate 100 clients that generate requests with the same rate in the machines. Each client performs the same policy as RackSched and tracks server queue lengths via piggybacking by its own. The performance of Client(10) (which emulates 10 clients) and Client(1000) (which emulates 1000 clients) are nearly the same as that of Client(100). Figure 14 shows the performance of RackSched, Shinjuku, the client-based solution and R2P2 under Bimodal(90%-50, 10%-500) and Bimodal(50%-50, 50%-500) workloads. In both workloads, RackSched outperforms others by maintaining low latency at higher request rate, and Client(100) has nearly the same performance as Shinjuku. More importantly, R2P2 is not robust to service time distributions. It is close to RackSched under Bimodal(50%-50, 50%-500), and the gap between R2P2 and RackSched is significantly larger under Bimodal(90%-50, 10%-500).

4.6 Analysis of RackSched

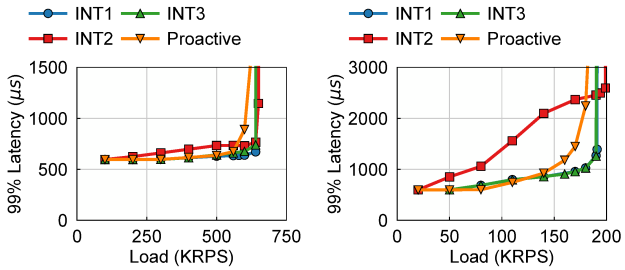
We analyze RackSched and show the impact of different design choices, including different scheduling policies of the switch-based inter-server scheduler and different mechanisms to track the server loads.

Impact of switch scheduling policies. Figure 15 evaluates the impact of different scheduling policies under Bimodal(90%-50, 10%-500) and Bimodal(50%-50, 50%-500) workloads. We compare four scheduling policies: *RR* (which



(a) Bimodal(90%-50,10%-500). (b) Bimodal(50%-50,50%-500).

Figure 15: Impact of switch scheduling policies.

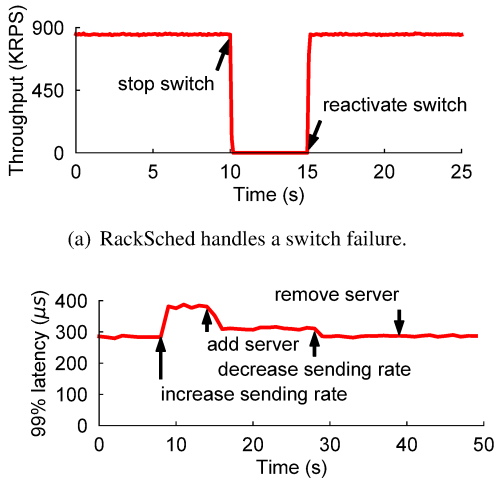


(a) Bimodal(90%-50,10%-500). (b) Bimodal(50%-50,50%-500).

Figure 16: Impact of server load tracking mechanisms.

schedules requests to server with round-robin), *Shortest* (which chooses the server with the smallest queue length), *Sampling*-2 (which samples two servers and chooses the one with the smaller queue length), and *Sampling*-4 (which samples four servers and chooses the one with the smallest queue length). RR sends an even number of requests to each server, without considering the variability of request service times. Thus, it suffers from long tail latency at high request load. Theoretically, *Shortest* can provide effective load balancing, but it incurs high tail latency in practice, even at low request load. As discussed in §2, the reason is that there is a delay to update the queue lengths in the switch from the servers. When a server becomes the one with the smallest queue length, multiple consecutive requests would all choose this server, causing a micro *herding* behavior. And the queue length of this server has to wait to be updated until the new queue length is piggybacked in the first reply packet to update in the switch. As discussed in §2, this herding behavior can be handled by adding randomization to the scheduling process. The results in the figure confirm the effectiveness of sampling. For the scale of the evaluated scenario, sampling two and four servers have similar performance, because sampling two servers already provides enough choices to avoid hotspots and enough randomization to avoid herding.

Impact of server load tracking mechanisms. Figure 16 evaluates the impact of different mechanisms to track server loads, under both Bimodal(90%-50, 10%-500) and Bimodal(50%-50, 50%-500) workloads. We compare three tracking mechanisms discussed in §3.5: *INT*1 (which tracks the number of outstanding requests for each server and computes the minimum), *INT*2 (which only tracks the minimum number of outstanding requests and updates on reply packets), *INT*3 (which tracks the total service time of outstanding requests for each server) and *Proactive* (which increments and decrements the counters by the switch). *Proactive* cannot precisely maintain the queue length for each server as packet loss and retransmissions can introduce errors on the counters, and as a result, it does not work well as others. *INT*2 performs worse than *INT*1 because it only keeps one server with the minimum load, resulting in herding. *INT*3 is comparable to *INT*1. However, it presumes that the service times are known as a priori, which is normally not the case in practice. *INT*1 works the best because it accurately tracks server loads, enables randomization to avoid herding for effective load balancing, and does not require any priori knowledge.



(a) RackSched handles a switch failure.

(b) RackSched handles server reconfigurations.

Figure 17: Handling failures and reconfigurations.

4.7 Request Affinity

Handling switch failures. To simulate a switch failure, we first stop the switch manually, then reactivate the switch after several seconds. Figure 17(a) shows the total throughput during this period under Exp(50) workload. At 10 s, the switch is stopped and the total throughput drops to 0. We reactivate the switch after 5 seconds and the total throughput recovers to the initial level. The microsecond-scale requests have already timed out after 5 seconds. So it is safe to start with an empty *ReqTable* after the reactivation, as discussed in §3.4.

Handling system reconfigurations. RackSched maintains request affinity during system reconfigurations. Figure 17(b) shows the 99% latency under system reconfigurations. We use two-packet requests under Exp(50) workload, and start with 500 KRPS load and seven machines as the servers. At time 8 s, we increase the request sending rate, and the 99% latency goes up to around 380 μs. At time 14 s, we add another server to serve the requests, and the 99% latency drops to 310 μs. At

time 28 s, we set the request sending rate back to 500 KRPS. And the 99% latency drops to 280 μs further. At time 39 s, we remove a server from the rack. Since seven servers are enough for such workload, the 99% latency remains the same. As discussed in § 3.4, the request affinity is maintained by the *ReqTable* in the above process.

5 Discussion

Target workloads. RackSched supports both stateless and replicated stateful services. Examples include microservices, function-as-a-service, stream processing, replicated caches and storage, and replicated machine learning models for high-throughput inference. It is unlikely for a stateful service to be replicated to all servers in a rack. We expect a more practical scenario is that a rack would run multiple such services, where each service is provided by a subset of (overlapping) servers, i.e., locality constraints. The evaluation shows that RackSched can provide significant improvements for a service hosted on just 8 servers (§4). And we provide additional results to show the benefits for multiple services with locality constraints in the technical report [74]. These results demonstrate that RackSched provides significant benefits even when the service is replicated on just a couple of servers.

Going beyond a rack. We focus on a single rack in this paper. A modern rack can already pack hundreds of cores, and a future rack is expected to pack thousands of cores [1, 2, 5], which is sufficient for many services. For planetary-scale services, a single microservice may span multiple racks. In this scenario, there is no central place like the ToR switch in a rack that can see and process all traffic. Yet, the abstraction of a rack-scale computer provided by RackSched provides a useful building block for distributed inter-rack scheduling. This would be an interesting direction for future work.

6 Related Work

Dataplane designs for low latency. Conventional networking stacks and operating systems usually sacrifice low latency for generality. To address the need for low latency, various dataplane designs have been proposed, including optimized networking stacks [4, 21, 34] and dataplane operating systems [14, 20, 32, 39, 54, 58, 59]. RackSched leverages such dataplane designs and enhances them with an inter-server scheduler, realizing low latency in a rack-scale computer.

Scheduling and resource management. There is a long line of research on job scheduling and resource management [26, 28, 29, 32, 38, 40, 42, 55, 57, 65, 66, 71]. Many systems focus on large jobs that can run from seconds to hours, and they can afford running sophisticated scheduling algorithms to make effective decisions. RackSched works at microsecond scale and optimizes the tail latency with network-system co-design.

Programmable switches. Programmable switches bring new opportunities to improve datacenter networks and systems, such as key-value stores [36, 48, 49, 50], coordination and consensus [35, 45, 46, 70, 73], network telemetry [3, 33], machine learning acceleration [61, 62] and query processing offload [44]. There are also proposals for managing systems built with programmable switches [30, 68, 72]. RackSched is a new solution that leverages the programmable switch as an inter-rack scheduler to optimize microsecond-scale tail latency for rack-scale computers.

7 Conclusion

We present RackSched, a rack-level microsecond-scale scheduler that provides the abstraction of a rack-scale computer to an external service. RackSched leverages a two-layer scheduling framework to achieve scalability and low tail latency. We hope that with the end of Moore's law and Dennard's scaling, RackSched will inspire a new generation of datacenter systems enabled by domain-specific hardware and hardware-software co-design.

Acknowledgments We thank our shepherd Ryan Stutsman and the anonymous reviewers for their valuable feedback. We thank Jack Humphries for helping debug Shinjuku issues. This work is supported in part by NSF grants CNS-1813487, CCF-1918757 and CNS-1955487, a Facebook Communications & Networking Research Award, and a Google Faculty Research Award.

References

[1] FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. https://www.usenix.org/node/179918.

[2] HP The Machine. https://www.labs.hpe.com/the-machine.

[3] In-band Network Telemetry (INT) Dataplane Specification. https://p4.org/specs/.

[4] Intel Data Plane Development Kit (DPDK). http://dpdk.org/.

[5] Intel Rack Scale Design. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[6] Memcached key-value store. https://memcached.org/.

[7] Redis data structure store. https://redis.io/.

[8] TPU Pods. https://cloud.google.com/tpu/.

[9] Voltdb in-memory database. https://www.voltdb.com.

[10] Barefoot P4 Studio. https://www.barefootnetworks.com/products/brief-p4-studio/.

[11] Barefoot Tofino. https://www.barefootnetworks.com/technology/.

[12] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 2017.

[13] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 2003.

[14] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX OSDI*, 2014.

[15] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *USENIX OSDI*, 2018.

[16] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, July 2014.

[17] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Putting the "micro" back in microservice. In *USENIX ATC*, 2018.

[18] M. Bramson, Y. Lu, and B. Prabhakar. Randomized load balancing with general service time distributions. *ACM SIGMETRICS performance evaluation review*, 38(1):275–286, 2010.

[19] F. Cerqueira and B. Brandenburg. A comparison of scheduling latency in linux, preempt-rt, and litmus rt. In *Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2013.

[20] A. Daglis, M. Sutherland, and B. Falsafi. RPCValet: NI-driven tail-aware balancing of μs-scale RPCs. In *ACM ASPLOS*, 2019.

[21] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *USENIX NSDI*, 2018.

[22] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, February 2013.

[23] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM*, 1989.

[24] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX NSDI*, 2016.

[25] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *ACM SIGCOMM*, August 2015.

[26] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX NSDI*, 2011.

[27] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *USENIX OSDI*, 2016.

[28] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *USENIX OSDI*, 2016.

[29] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *USENIX OSDI*, 2016.

[30] D. Hancock and J. Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *ACM CoNEXT*, 2016.

[31] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 2019.

[32] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *ACM SIGCOMM HotNets Workshop*, 2019.

[33] N. Ivkin, Z. Yu, V. Braverman, and X. Jin. QPipe: Quantiles sketch fully in the data plane. In *ACM CoNEXT*, December 2019.

[34] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *USENIX NSDI*, 2014.

[35] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *USENIX NSDI*, 2018.

[36] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*, 2017.

[37] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, 2014.

[38] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *USENIX OSDI*, 2016.

[39] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX NSDI*, 2019.

[40] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis. Centralized core-granular scheduling for serverless functions. In *ACM Symposium on Cloud Computing*, 2019.

[41] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter rpcs can be general and fast. In *USENIX NSDI*, 2019.

[42] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *USENIX ATC*, 2019.

[43] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *USENIX OSDI*, 2018.

[44] A. Lerner, R. Hussein, P. Cudre-Mauroux, and U. eXascale Infolab. The case for network accelerated query processing. In *CIDR*, 2019.

[45] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM SOSP*, October 2017.

[46] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*, November 2016.

[47] J. Li, J. Nelson, X. Jin, and D. R. Ports. Pegasus: Load-aware selective replication with an in-network coherence directory. *Technical Report UW-CSE-18-12-01*, 2018.

[48] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *USENIX NSDI*, March 2016.

[49] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward in-network computation with an in-network cache. In *ACM ASPLOS*, April 2017.

[50] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *USENIX FAST*, 2019.

[51] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[52] NoviSwitch. `http://noviflow.com/products/noviswitch/`.

[53] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless datacenter load-balancing with Beamer. In *USENIX NSDI*, 2018.

[54] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX NSDI*, 2019.

[55] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *ACM SOSP*, 2013.

[56] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud scale load balancing. In *SIGCOMM CCR*, 2013.

[57] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.

[58] S. Peter, J. Li, I. Zhang, D. R. Ports, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *USENIX OSDI*, 2013.

[59] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *ACM SOSP*, 2017.

[60] RocksDB. RocksDB. `https://rocksdb.org/`.

[61] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *ACM SIGCOMM HotNets Workshop*, November 2017.

[62] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv*, 2019.

[63] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *USENIX NSDI*, 2015.

[64] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *ACM SOSP*, 2013.

[65] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, 2016.

[66] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *ACM Symposium on Cloud Computing*, 2013.

[67] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. TAO: How Facebook serves the social graph. In *ACM SIGMOD*, May 2012.

[68] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. Ports, and A. Panda. Multitenancy for fast and programmable networks in the cloud. In *USENIX HotCloud Workshop*, July 2020.

[69] A. Wierman and B. Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[70] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin. Netlock: Fast, centralized lock management using programmable switches. In *ACM SIGCOMM*, 2020.

[71] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *USENIX OSDI*, 2008.

[72] P. Zheng, T. Benson, and C. Hu. P4Visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *ACM CoNEXT*, 2018.

[73] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proceedings of the VLDB Endowment*, 2019.

[74] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. RackSched: A microsecond-scale scheduler for rack-scale computers (technical report). In *arXiv*, 2020.