When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone

Henri Maxime Demoulin
University of Pennsylvania, USA

Joshua Fried
MIT CSAIL, USA

Isaac Pedisich
Grammatech*, USA

Marios Kogias
Microsoft Research, United Kingdom

Boon Thau Loo
University of Pennsylvania, USA

Linh Thi Xuan Phan
University of Pennsylvania, USA

Irene Zhang
Microsoft Research, USA

Abstract

This paper introduces Perséphone, a kernel-bypass OS scheduler designed to minimize tail latency for applications executing at microsecond-scale and exhibiting wide service time distributions. Perséphone integrates a new scheduling policy, Dynamic Application-aware Reserved Cores (DARC), that reserves cores for requests with short processing times. Unlike existing kernel-bypass schedulers, DARC is not work conserving. DARC profiles application requests and leaves a small number of cores idle when no short requests are in the queue, so when short requests do arrive, they are not blocked by longer-running ones. Counter-intuitively, leaving cores idle lets DARC maintain lower tail latencies at higher utilization, reducing the *overall* number of cores needed to serve the same workloads and consequently better utilizing the datacenter resources.

CCS Concepts: • **Computer systems organization**;

Keywords: Kernel-bypass; Scheduling

ACM Reference Format:
Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–28, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3477132.3483571

*Work done while at Penn.

1 Introduction

Datacenter networks and in-memory systems increasingly have (single) microsecond [10] latencies. These latencies are critical for today's complex cloud applications to meet SLOs while fanning out to hundreds of datacenter backend servers [21, 72]. At microsecond-scale, the distribution of request processing times can be especially extreme; for example, Redis can process GET/PUT requests in 2μs [74] but more complex SCAN and EVAL requests can take hundreds of microseconds or milliseconds to complete. As a result, a single long-running request can block hundreds or thousands of shorter requests.

To bound tail latency, especially for short requests, modern datacenter servers run at low utilization to keep queues short and reduce the likelihood that a short request will block behind long requests. For instance, Google reports that machines spend most of their time in the 10-50% utilization range [11]. Unfortunately, this approach wastes precious CPU cycles and does not guarantee that microsecond datacenter systems will always meet SLOs for short requests.

Recent kernel-bypass schedulers have improved utilization with shared queues [62] and work-stealing [73, 75] but these techniques only work for uniform and lightly-tailed workloads. For workloads with a wide distribution of response times, Shinjuku [48] leverages interrupts for processor sharing; however, Shinjuku's interrupts impose non-negligible delays for single digit microsecond requests and are too expensive to run frequently (our experiments saw ≈2us per interrupt and preempting as often as every 5μs had a high penalty on sustainable load). Furthermore, Shinjuku's non-standard use of hardware virtualization features makes it difficult to use in the datacenter [62] and public clouds, *e.g.*, Google Cloud, Microsoft Azure, AWS, *etc.*

Recent congestion control schemes [3, 68], similarly, optimize network utilization and reduce flow completion times by approximating *Shortest-Remaining-Processing-Time (SRPT)*, which is optimal for minimizing the average waiting time [79]. Unlike CPU scheduling, though, switch packet schedulers have a physical 'preemption' unit, which is the MTU in the worst case; they process packet headers that include the actual message size; and leverage traffic classes that can prioritize packets based on the size of the flow they belong to, which makes scheduling decisions and policy enforcement easier. A CPU scheduler cannot know in advance for how long each request will occupy the CPU and there is no upper limit on execution time, which makes the implementation of *SRPT-like* policies, or generally policies that prioritize short requests, hard to implement at the microsecond scale.

The unifying factor between congestion control schemes, such as Homa [68] and CPU schedulers, such as Shinjuku, that deal with heavy-tail flow and request distributions, respectively, is that they both temporarily multiplex the shared resource. This paper takes a different approach to CPU scheduling for heavy-tailed service time distributions by taking advantage of parallelism and the abundance of cores on a modern multicore server through application-aware [52] spatial isolation of CPU resources.

First, we observe that a kernel-bypass scheduler can, with a little help from programmers, identify the type of incoming requests. For many cloud applications, the messaging protocol exposes the required mechanisms to declare request types: Memcached request types are part of the protocol's header [63]; Redis uses a serialization protocol specifying commands [86]; Protobuf defines Message Types [35]; Next, we observe that requests of the same type often have similar processing types, so, given the ability to identify types, we can track past per-type process times to predict future processing times. Finally, we carefully leave cores idle to prevent short requests from queuing behind arbitrarily longer ones.

Inspired by prior research in networking [2], our approach goes against the grain for OS schedulers, which commonly prioritize work conservation. We show that by making a minor sacrifice in the maximum achievable throughput, we can increase the achievable throughput under an aggressive latency SLO and as a result, increase the overall CPU utilization of the datacenter.

To implement this approach, we need to tackle two challenges: (1) predict how long each request type will occupy a CPU and (2) efficiently partition CPU resources among types while retaining the ability to handle bursts of arrivals and minimizing CPU waste. To this end, we introduce Perséphone, an application-aware kernel-bypass scheduler. Perséphone lets applications define *request classifiers* and uses these classifiers to dynamically profile the workload. Using these profiles, Perséphone implements a new scheduling policy, *Dynamic, Application-aware Reserved Cores* (DARC) that leverages work conservation for short requests only and is not work conserving for long requests. DARC prioritizes short requests at a small cost in throughput – 5% in our experiments – and is best suited for applications that value microsecond responses. For other applications, existing kernel-bypass scheduler work well, though we believe there is a large set of datacenter workloads that can benefit from DARC.

We prototype Perséphone using DPDK and compare it to two state-of-the-art kernel-bypass schedulers: Shinjuku [48] and Shenango [73]. Using a diverse set of workloads, we show that Perséphone with DARC can drastically improve requests tail latency and sustain up to 2.3x and 1.3x more load than Shenango and Shinjuku, respectively, at a target SLO. In addition, these improvements come at a lower cost to long requests than Shinjuku's preemption technique, highlighting the challenges of traditional OS scheduling techniques at microsecond scale.

2 The Case for Idling

For workloads with wide service time distribution, long requests can block short requests even when queues are short because long requests can easily occupy all workers for a long time. We refer to this effect as *dispersion-based head-of-line blocking*. To better understand how dispersion-based blocking affects short requests, we look beyond request latency and study *slowdown*: the ratio of total time spent at the server over the time spent doing pure application processing [40].

Slowdown better reflects the impact of long requests on short requests. For heavy-tailed workloads, short requests experience a slowdown proportional to the length of the tail. More concretely, consider the following workload, similar to Zygos' "bimodal-2" [75], a mix of 99.5% short requests running for 0.5μs and 0.5% long requests executing in 500μs. A short request blocked behind a long one can experience a slowdown of up to 1001, while a long request blocked behind a short request will see a slowdown of 1.001. As a result, a few short requests blocked by long requests will drive the slowdown distribution and increase tail latency.

Using this workload, we simulate four scheduling policies, including DARC, listed in Table 1. *Decentralized first come, first served* (d-FCFS) models Receive Side Scaling, widely used in the datacenter today [27, 60] and by IX [14] and Arrakis [74]. With d-FCFS, each worker has a local queue and receives an even share of all incoming traffic. *Centralized first come, first served* (c-FCFS) uses a single queue to receive all requests and send them to idle workers. c-FCFS is usually used at the application level — for example, web servers (*e.g.*, NGINX) often use a single dispatch thread — and captures recent research on kernel-bypass systems [73, 75], which simulate c-FCFS with per-worker queues and work stealing. *Time Sharing* (TS) is used in the Shinjuku system [48], with multiple queues for different request types and interrupts at the microsecond scale using Dune [13]. We simulate TS with a 5μs preemption frequency and 1μs overhead per preemption, matching Shinjuku's reported ≈ 2000 cycles overhead on a 2GHz machine.

Figure 1 shows our simulation results assuming, an ideal system with no network overheads. We use 16 workers, simulate 1 second of requests distributed under Poisson, and report the observed slowdown for the 99.9th percentile of

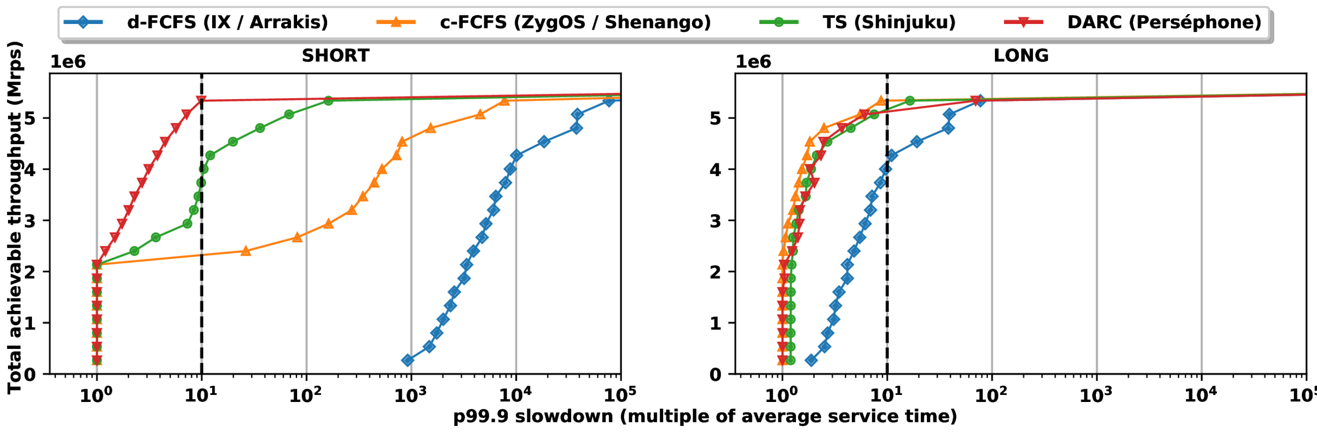


Figure 1. Simulated achievable throughput as a function of 99.9th percentile slowdown for the policies listed in Table 1 on a 16 cores system and a workload composed of 99.5% short requests (0.5μs) and 0.5% long requests (500μs). For a target SLO of 10 times the average service time *for each request type*, c-FCFS and TS can only handle 2.1 and 3.7 Millions requests per second (Mrps), respectively. DARC can sustain 5.1 Mrps for the same objective. The Y axis represents the total achievable throughput for the entire workload.

Table 1. Unlike most existing kernel-bypass OS schedulers, DARC is not work conserving. It extracts request types from incoming requests, estimates how long a request will occupy a CPU before scheduling it and reserves workers for short requests to minimize dispersion-based head-of-line blocking.

Policy	Exploit typed queues	Non Work conserving	Non preemptive	Example System
d-FCFS	✗	✓	✓	IX [14] Arrakis [74]
c-FCFS	✗	✗	✓	ZygOS [75] Shenango [73]
TS	✓	✗	✗	Shinjuku [48]
DARC	✓	✓	✓	Perséphone

each type of requests — to capture the impact of the 0.5% long requests on the tail — at varying utilizations, up to a maximum of 5.3 million requests per second (Mrps).

d-FCFS performs poorly; it offers an uncontrolled form of non work conservation where workers sit idle while requests wait in other queues. Additionally, d-FCFS has no sense of request types: workers might process a long request ahead of a short one if it arrived first. c-FCFS performs better because it is work conserving but short requests will block when all workers are busy processing long requests. To meet a target SLO of 10x slowdown *for each type of requests*, c-FCFS must run the server at 2.1 Mrps, 40% of the peak load. Shinjuku's TS policy fares better than c-FCFS and d-FCFS, being both work conserving and able to preempt long requests: it maintains slowdown below 10 up to 3.7 Mrps, 70% of the peak load. However, this simulation accounts for an optimistic 1μs preemption overhead and overlooks the practicality of supporting preemption at the microsecond scale (*c.f.*, Sec. 6).

The DARC way: Our key insight is that prioritizing short requests is critical to protect their service time, an observation the networking community has already made when designing datacenter congestion control schemes [2, 3, 68]. However, using traffic classes and bounded buffers do not work for CPU scheduling since schedulers do not know how long a request may occupy a CPU and preemption is unaffordable at single-digit microsecond scales. We observe that *leaving certain cores idle for readily handling potential future (bursts of) short requests is highly beneficial at microsecond scale.* For a request that takes 1 μs or less, even preempting as frequently as every 5μs introduces a 6x slowdown. Instead, given an understanding of each request's potential processing time, an application aware, not work conserving policy can reduce slowdown for short requests by estimating their CPU demand and dedicating workers to them. These workers will be idle in the absence of short requests, but when they do, they are guaranteed to not be blocked behind long requests.

As seen in Figure 1, DARC can meet the 10x slowdown SLO target for both type of requests at 5.1 Mrps. This represents *2.5x and 1.4x more sustainable throughput* than c-FCFS and an optimistically cheap time sharing policy. At this load, short requests experience 9.87μs p99.9th tail latency, *3 and 1 orders of magnitude* smaller than c-FCFS and TS with 7738μs and 161μs, respectively. To achieve this, DARC asks programmers for a *request classifier* to identify types and uses this classifier to dynamically estimate requests' CPU demand. In this example, DARC reserves 1 worker for short requests at a small penalty of 5% achievable throughput.

Counter-intuitively, although DARC wastes cycles idling, it *reduces the overall number of machines needed to serve this workload* because servers can run at much higher utilization while retaining good tail latencies for short and long requests.

3 DARC Scheduling

The objective of DARC is to improve tail latency for single-digit microsecond requests in cloud workloads without preemption. Like recent networking techniques that co-design the network protocol and the management of network switches' priority queue [2, 3, 68] to favor small messages, we protect short requests at backend servers by extracting their type, understanding their CPU demand, and dedicating enough resources to satisfy their demand.

In this section, we describe the challenges associated with implementing these techniques as a CPU scheduling policy, then present the DARC scheduling model, how to compute reservations and when to update them. Table 2 describes the notation used throughout this section.

Challenges. Protecting short requests in a dynamic way through priority queues and non work conservation is difficult because we need to (1) predict how long each request type will occupy a CPU and (2) partition CPU resources among types while retaining the ability to handle bursts of arrivals and minimizing CPU waste.

The first challenge stems from the granularity of operation DARC is targeting, microsecond scales, and from the need to react to changes in workload. We tackle this challenge with a combination of low-overhead workload profiling and queuing delay monitoring, using the former to build a fingerprint of requests' CPU demand and the latter as a signal that this fingerprint might have significantly changed. This section describes the technique and Sec. 4 its implementation.

The second challenge can be detailed in two parts: burst-tolerance and CPU waste. First, though reducing the number of cores available to a given request type forbids it from negatively impacting other types, it also reduces its ability to absorb bursts of arrivals [57]. We solve this tension by enabling *cycles stealing* from shorter types to longer ones, a mechanism in which short requests can execute on cores otherwise reserved for longer types — but not the opposite. The rationale for stealing is that shorter requests comparatively cause less slowdown to long requests. Note that cycle stealing is a similar concept to work stealing [73, 75] but is different in practice, as it is performed from the DARC dispatcher rather than from application workers (thus does not require expensive cross-worker coordination).

Second, and similarly to message types and priority queues in network devices, the number of request types can be different than the number of CPU cores on the machine, so very likely the demand for each request type will be fractional — *i.e.*, a request type could require 2.5 workers on average. As a result, we need to determine a strategy for sharing — or not — CPU cores between certain request types. Sharing cores leads to a tension: regrouping types onto the same cores risks dispersion-based blocking, but always giving entire cores to types with fractional demand can lead to over-provisioning and starving other types. We handle this tension with two mechanisms: grouping types together and providing spillway cores. Grouping lets all request types fit onto a limited number of cores and reduces the number of fractional ties while retaining the ability to separate types based on processing time. Spillway cores allows DARC to always provide service to types with little average CPU demand (typically much less than an entire core) as well as undeclared, unknown requests.

Table 2. Notation used to define DARC

Symbol	Description
N	Number of request types
S	Average service time
τ	A request type
$\tau.S$	Type's average service time
$\tau.R$	Type's occurrence
δ	Service time similarity factor for two types

Scheduling model. DARC presents a single queue abstraction to application workers: it iterates over typed queues sorted by average service time and dequeues them in a first come, first served fashion. Requests of a given type can be scheduled not only on their reserved cores but also steal cycles from cores allocated to longer types — a concept used in Cycle Stealing with Central Queue (CSCQ), a job dispatching policy for compute clusters [42]. Algorithm 1 describes the process of worker selection. For each request type registered in the system, if there is a pending request in that type's queue, DARC greedily searches the list of reserved workers for an idle worker. If none is found, DARC searches for a stealable worker. If a free worker is found, the head of the typed queue is dispatched to this worker. When a worker completes a request, it signals completion to the DARC dispatcher.

DARC reservation mechanism. The number of workers to dedicate to a given request type is based on the average CPU demand of the type at peak load. We use average demand because it is a provable indicator of stability [40] for the system. In addition, workloads can have performance outliers that should not necessarily drive SLOs [22]. We compute average CPU demand using the workload's composition, normalizing the contribution of each request type's average service time to the entire workload's average service time. The contribution of a given request type is its average service time multiplied by its occurrence ratio as a percentage of the

Algorithm 1 Request dispatching algorithm

```
procedure Dispatch(Types)
    w ← None
    for $\tau$ ∈ Types.sort() do
        if $\tau$.queue == ∅ then
            continue
        else
            workers ← $\tau$.reserved ∪ $\tau$.stealable
            for worker ∈ workers do
                if worker.is_free() then
                    w ← worker
                    break
            if w ≠ None then
                r ← $\tau$.queue.pop()
                schedule(r, w)
```

Algorithm 2 Worker reservation algorithm

```
procedure Reserve(Types, $\delta$)
    // First group together similar request types
    groups = group_types(Types, $\delta$).sort()
    // Then attribute workers
    $S \leftarrow \sum_{j=0}^{N} S_j * R_j$
    n_reserved = 0
    for g ∈ groups do
        g.$S = \sum \tau.S * \tau.R \ \forall \ \tau \in$ g
        d = $\frac{g.S}{S}$
        P ← round(d)
        if P == 0 then P ← 1
        for $i$ ← 0; $i$ < P; $i$++ do
            g.reserved[$i$] ← next_free_worker()
            n_reserved++;
        // Set stealable workers
        n_stealable ← num_workers - n_reserved;
        for $i$ ← 0; $i$ < n_stealable; $i$++ do
            g.stealable[$i$] ← next_free_worker()
```

entire workload. Specifically, given a set of N request types $\{\tau_i \ ; i = 0 \ldots N\}$, the average CPU time demand Δ_i of τ_i with service time S_i and occurrence ratio R_i is:

$$0 \leq \frac{S_i * R_i}{\sum_j^N S_j * R_j}, \leq 1 \tag{1}$$

Given a system with W workers, this means that we should attribute $\Delta_i * W$ workers to τ_i.

Because CPU demand can be fractional and given the non-preemptive requirement we set for the system, we need a strategy to attribute fractions of CPUs to request types. For each such "fractional tie", we have to make a choice: either `ceil` fractions and always grant entire cores or `floor` fractions and consolidate fractional CPU demands on *shared cores*. The former risks over-provisioning certain types, at the cost of others, while the latter risks creating dispersion-based blocking by mixing long and short requests onto the same core(s).

Our approach combines the two: first we decrease the number of "fractional ties" by grouping request types of similar processing times and computing a CPU demand for the entire group; second we round this demand. As a result, for G groups, if f_i is the fractional demand of group i, the average CPU waste for DARC across all G groups is:

$$\sum_{i, f_i \geq .5}^{G} 1 - f_i \tag{2}$$

across all f_i that are greater or equal to 0.5 — otherwise it is 0 for group i. In practice, during bursts, because we selectively enable work conservation through work stealing for shorter requests, CPU waste is smaller.

Algorithm 2 describes the reservation process. First, we identify similar types whose average service time falls within a factor δ of each other. Next, we compute the demand for each group and accordingly attribute workers to meet it, rounding fractional demands in the process. We always assign at least one worker to a group. DARC grouping strategy can still result in earlier groups — of shorter requests — consuming all CPU cores. For example, a group of long requests with a CPU demand smaller than 0.5 will not find any free CPU core. To provide service to these groups, we set aside "spillway" cores. If there are no more free workers, `next_free_worker()` returns a spillway core. In our experiments (Sec. 5), we use a single spillway core.

Finally, we selectively enable work conservation for shorter requests and let each group steal from workers not yet reserved, *i.e.*, workers that are to be dedicated to longer request types. This lets DARC better tolerate bursts of shorter requests with little impact on the overall tail latency of the workload.

As we process groups in order of ascending service time, we favor shorter requests, and it is possible for our algorithm to under-provision long requests — but never deny them service thanks to spillway cores. Operators can tune the δ grouping factor to adjust non work conservation to their desired SLOs. Grouping lets DARC handle workloads where the number of distinct types is higher than the number of workers.

Profiling the workload and updating reservations. At runtime, the DARC dispatcher uses *profiling windows* to maintain two pieces of information about each request type: a moving average of service time and an occurrence ratio. These are the S_i and R_i of equation 1. The dispatcher gathers them when application workers signal work completions. The dispatcher uses queuing delay and variation in CPU demand as performance signals. If the former goes beyond a target slowdown SLO and the latter deviates significantly from the current demand, the dispatcher proceeds to update reservations and transition to the next windows. During the first windows, at startup, the system starts using c-FCFS, gathers samples, then transitions to DARC. This technique lets DARC cope with changing workloads where a type's profile changes (effectively, *misclassification*). During a profiling window, unknown or unexpected requests can use the spillway core(s) to execute. We discuss the sensitivity of this mechanism in Sec. 4.3.3.

4 Perséphone

We implement DARC within a kernel-bypass scheduler called Perséphone. Though DARC requires no special hardware or major application modifications, Perséphone must meet the following requirements to support microsecond-scale kernel-bypass applications: (1) the networking stack must be able to efficiently sort requests by type in the data path, (2) the scheduler must be able to quickly make per-request scheduling decisions, and (3) workload profiling for updating DARC reservation must present low overheads.

Perséphone meets the first requirement with a *request classifiers* API for capturing request types. Using request classifiers, programmers provide a way for the system to classify requests based on types as they enter the system. Perséphone meets the remaining two requirements with a carefully architected networking stack, profiler, and scheduler packaged in a user-level library.

4.1 System Model

Perséphone is designed for datacenter services that must handle large volumes of traffic at microsecond latencies. Examples include key-value stores, fast inference engines [54], web search engines and RESTful micro-services. We assume the application uses kernel-bypass for low latency I/O (e.g., with DPDK [26] or RDMA [80]) and performs all application and network processing through Perséphone. Our current prototype is designed for UDP networking, but our technique also works for a stateful dispatcher (*c.f.*, Sec. 6 for a more elaborate discussion).

4.2 Request classifiers

Perséphone relies on user-defined functions, *i.e.*, "request classifiers" to group incoming requests. A request classifier accepts a pointer to an application payload (Layer 4 and above) and returns a request type. If the classifier cannot recognize a request, Perséphone categorizes it as `UNKNOWN` and places it in a low priority queue. There is at most one classifier active at a time in our current design. Though most of our target applications use optimized protocols such as Redis' RESP [86] that allow a classifier to look-up for a header field to parse the request type, we opted for generality and allowing users to write arbitrarily complex classifiers. There is, of course, a performance trade-off: a non-optimized request classifier will impact the dispatcher's performance because

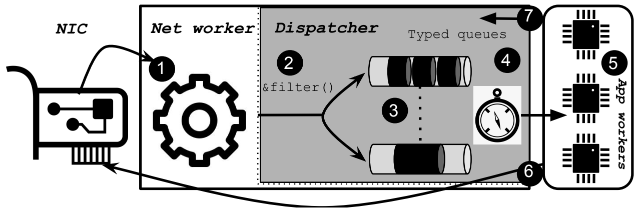


Figure 2. Perséphone architecture. After the net worker processes incoming packets, the dispatcher classifies requests using a user-defined classifier. Requests wait in typed queues for DARC to push them to workers. (*c.f.*, Sec. 4.3 for pipeline details.)

request classifiers are "bumps-in-the-wire" on the dispatching critical path. We leave it to users to quantify this trade off based on the performance they wish to obtain from the dispatcher (*i.e.*, how many requests per second it should be able to sustain). While a complete study of classifier performance is out of scope for this paper, we found that for standard protocols where the request type's position is known in the header, our dispatcher can process up to 7 millions packets per second on our testbed, a number competitive with existing kernel-bypass schedulers.

4.3 Perséphone Architecture

Perséphone consists of three components, shown in Figure 2: one or many *net workers* dequeueing packets from the network card, a *dispatcher* applying request classifiers and performing DARC scheduling, and *application workers* performing application processing (*e.g.*, fetching the value from the key-value store). These components operate as an event-driven pipeline and process packets as follows. ❶ On the ingress path, the net worker takes packets from the network card and pushes them to the dispatcher, which ❷ classifies incoming requests using a user-defined *request classifier* and ❸ stores them in *typed queues*, *i.e.*, buffers specialized for a single request type. ❹ The dispatcher, running DARC, selects a request from a typed queue and pushes it to a free *application worker*. ❺ The worker processes the request, formats a response buffer, and ❻ pushes a pointer to that buffer to the NIC. In addition, ❼ the application worker notifies the dispatcher that it has completed the request.

4.3.1 Networking Both the net worker and application workers receive a *network context* at initialization. This context gives them unique access to receive and transmit queues in the NIC. Perséphone registers a statically allocated memory pool with the NIC for contexts to quickly allocate new buffers when receiving packets. This memory pool is backed by a multi-producer, single-consumer ring so workers can release buffers after transmission. Both the net and application workers use a thread-local buffer cache to decrease interactions with the main memory pool. For requests contained in a single application-level buffer, we perform zero-copy and pass along to workers a pointer to the network buffer. To issue a response on the transmit path — and if the request holds in a single packet — the worker reuses the ingress network buffer to host the egress packet, reducing the number of distinct network buffers in use (with the goal of allowing all buffers to fit in the Last Level Cache space used by DDIO [25] — usually 10% of the LLC). Our current implementation requires copy if the request spans multiple packets.

4.3.2 Component Communication

The dispatcher uses single-producer, single-consumer circular buffers to share requests and commands with application workers in a lock-less interaction pattern. We use a lightweight RPC design inspired by Barrelfish [12], where both sender and receiver synchronize their send/read heads using a shared variable. To reduce cache coherence traffic between cores, the sender synchronizes with the receiver — to update the read head and avoid overflows — only when its local state shows the buffer to be full. In our prototype, operations on that channel take 88 cycles on average.

4.3.3 Dispatcher

The dispatcher maintains three main data structures: a list of `RequestType` objects, which contains type information such as the type ID and instrumentation data; typed request queues; and a list of free workers. In addition, the dispatcher holds a pointer to a user-defined request classifier. The list of free workers is updated whenever a request is dispatched and each time application workers notify the dispatcher about work completion; this is done using a specific control message on the memory channel shared between dispatcher and each worker. Finally, the dispatcher maintains profiling windows, during which it computes a moving average of service times by request type and increment a counter for each type seen so far. DARC uses these profiling windows to compute resource allocation (Sec. 3) and adjust to changes in the workload's composition. In our prototype, at the median, updating the profile of a request takes 75 cycles, checking whether an update is required takes about 300 cycles, and performing a reservation update takes about 1000 cycles.

To control the sensitivity of the update mechanism in face of bursty arrivals, we set a lower bound on the number of samples required to transition — 50000 in our experiments — and the minimum deviation in CPU demand from the current allocation — 10% in our experiments. As a measure of flow control, when the system is under pressure and workers cannot process requests as fast as they arrive, the dispatcher drops requests from typed queues that are full. This allows to shed load only for overloaded types without impacting the rest of the workload.

Table 3. Workloads exhibiting 100x and 1000x dispersion.

Workload	*Short* Runtime (μs)	*Short* Ratio	*Long* Runtime (μs)	*Long* Ratio
High Bimodal	1	50%	100	50%
Extreme Bimodal	0.5	99.5%	500	0.5%

4.3.4 Application Workers

Upon receiving a pointer to a request, application workers dereference it to access the payload. As an optimization, they can access the request type directly from the `RequestType` object rather than duplicating work to identify needed application logic (e.g., to differentiate between a SET or GET request). Once they finish processing the request, they reuse the payload buffer to format a response and push it to the NIC hardware queue using their local network context. Finally, they signal work completion to the dispatcher.

5 Evaluation

We built a prototype of Perséphone, in 3700 lines of C++ code[1], to evaluate DARC scheduling against policies provided by Shenango [73] and Shinjuku [48]:

- For a workload with 100x dispersion between short and long requests, Perséphone can sustain 2.35x and 1.3x more throughput compared to Shenango and Shinjuku, respectively (Sec. 5.4.1)
- For a workload with 1000x dispersion, Perséphone can sustain 1.4x more throughput than Shenango and improve slowdown by up to 1.4x over Shinjuku for short requests. (Sec. 5.4.2)
- For a workload modeled on the TPC-C benchmark, Perséphone reduces slowdown by up to 4.6x over Shenango and up to 3.1x over Shinjuku. (Sec. 5.4.3)
- For a RocksDB application, DARC can sustain 2.3x and 1.3x higher throughput than Shenango and Shinjuku, respectively (Sec. 5.4.4)

In addition, we demonstrate that Perséphone can handle adversarial situations where workloads changes swiftly and where programmers provide an incorrect request classifier.

5.1 Experimental Setup

Workloads. We model workloads exhibiting different service time dispersion after examples found in academic and industry references. Often such workloads exhibit n-modal distributions with either an equal amount of short and long requests (*e.g.*, workload A in the YCSB benchmark [20]) or a majority of short requests with a small amount of very long requests (*e.g.*, Facebook's USR workload [7]). Dispersion between shorter and longer requests is commonly found to be two orders of magnitude or more [5, 18, 66]. We evaluate *High Bimodal* and *Extreme Bimodal* (Table 3), two workloads that exhibit large service time dispersion, and *TPC-C* (Table 4), which models requests in the eponymous benchmarking suite [84], a standardized OLTP model for e-commerce.

[1]https://github.com/maxdml/psp

Table 4. The TPC-C benchmark models operations of an online store. Payment and NewOrder transactions are most frequent.

Transaction name	Runtime (μs)	Ratio	Dispersion
Payment	5.7	44%	1x
OrderStatus	6	4%	1.05x
NewOrder	20	44%	3.3x
Delivery	88	4%	15.4x
StockLevel	100	4%	17.5x

Finally, we evaluate DARC with an in-memory store built over RocksDB, a database engine used at Facebook [32].

With *High Bimodal*, long requests represent 50% of the workload but "only" exhibit 100x dispersion. With *Extreme Bimodal*, long requests are much slower — 1000x slower — but very infrequent (0.5% of the mix). We profile *TPC-C* transactions with an in-memory database [85] and run it as a synthetic workload. Our goal with *TPC-C* is to evaluate how Perséphone performs with an n-modal request distribution. The workload consists of five request types with moderate service time dispersion — at most 17.5x between infrequent `StockLevel` requests and frequent `Payment` requests. We assume that requests are not dependent on each other. Finally, the RocksDB workload is made of 50% GETs and 50% SCANs requests, executing for 1.5μs and 635μs, respectively, and exhibiting a 420x dispersion factor. This workload strikes a balance between *High Bimodal* and *Extreme Bimodal*.

Performance metrics. We present two performance views: (i) the slowdown at the tail taken across all requests in the experiment, and (ii) the *typed* tail latency, *i.e.*, a selected percentile over *only* the type's response times' distribution. These views help us to understand the various trade-offs offered by the systems and policies under evaluation. For both metrics, we use the 99.9th percentile and plot them as a function of the total load on the system.

Client. The client is a C++ open loop load generator that models the behavior of bursty production traffic. It generates requests under a Poisson process centered at the workloads' average service time. Each experiment runs for 20 seconds and we discard the first 10% of samples to remove warm-up effects. We ran our experiments for several minutes and found the results similar. To interact with the server, we use a simple protocol where *TPC-C* transaction ID, RocksDB query ID, and synthetic workload request types are located in the requests' header. We accordingly register a request classifier on the server to map these IDs to request types. This request classifier adds a one-time ≈ 100 nanoseconds overhead to each request.

Systems. In addition to Perséphone, we compare two state-of-the-art systems: Shenango and Shinjuku. Shenango's *IOKernel* uses RSS hashes to steer packets to application cores, which perform work stealing to balance load and avoid dispersion-based blocking, in a fashion similar to ZygOS [75].

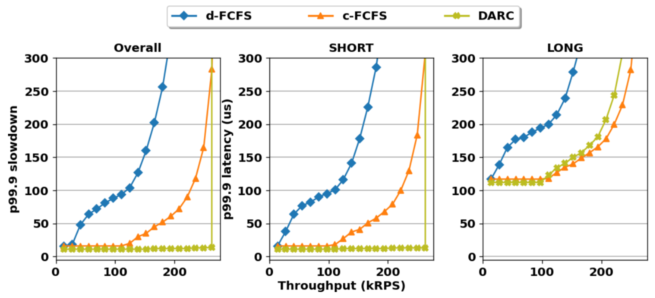


Figure 3. Evaluating DARC on *High Bimodal* (50.0:1.0 – 50.0:100.0) within Perséphone. The first column is p99.9 overall slowdown, the second and third p99.9 latency for short and long requests, respectively. For all columns, the X axis is the total load on the system. DARC improves slowdown over c-FCFS by up to 15.7x, at a cost of up to 4.2x increased latency for long requests.

We also compare to a version of Shenango with work stealing disabled, to evaluate d-FCFS. We choose Shenango over ZygOS due to its more recent implementation and its support for UDP. Shinjuku implements microsecond-scale, user-level preemption by exploiting Dune's virtualization features [13] at up to 5μs frequency. Leveraging this ability to preempt, Shinjuku implements a single queue policy, where preempted requests are enqueued at the tail of the queue, and a multi-queue policy with a queue per request type and where preempted requests are enqueued at the head of their respective queue. The multi-queue policy selects the next queue to dequeue using a variant of Borrowed Virtual Time [29]. Across experiments, DARC updates reservations whenever a request experiences queuing delays of ten times its average profiled service time. Lastly, all systems use UDP networking.

Testbed. We use 7 Cloudlab [30] c6420 nodes (6 clients, 1 server), each equipped with a 16-core (32-thread) Intel Xeon Gold 6142 CPU running at 2.60GHz, 376GB of RAM, and an Intel X710 10 Gigabit NIC. The average network round trip time between machines is 10μs. We disabled TurboBoost and set `isolcpu`. Shinjuku and Perséphone run on Ubuntu 16.04 with Linux kernel version 4.4.0. Shenango runs on Ubuntu 18.04 with Linux kernel version 5.0. Shinjuku uses one hyperthread for the net worker and another for the dispatcher, collocated on the same physical core. Shenango runs its IOKernel on a single core, and Perséphone runs both its net worker and dispatcher on the same hardware thread. All systems use 14 worker threads running on dedicated physical cores. For Shenango, we provision all cores at startup and disable dynamic core allocation since we want to evaluate performance for a single application and Shenango otherwise re-assign cores to multiple applications running on the same machine.

5.2 DARC versus existing policies

To validate that DARC improves performance of short requests compared to c-FCFS and d-FCFS, we run these policies

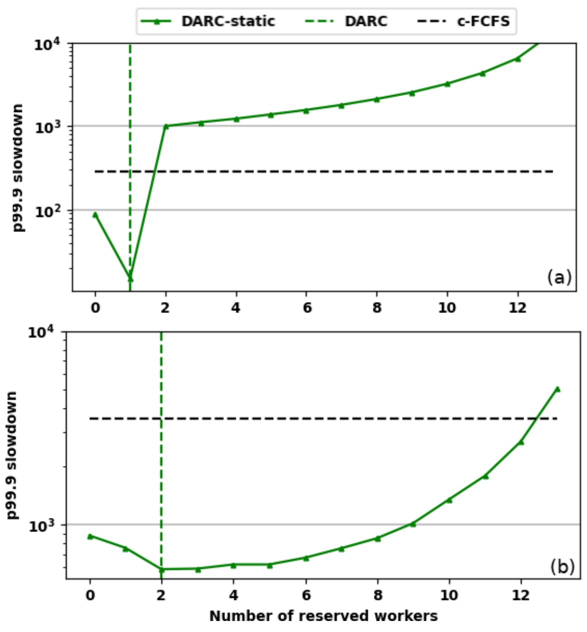


Figure 4. Gradually adjusting the degree of work conservation (" DARC-static") with *High Bimodal* and *Extreme Bimodal* at 95% load. Reserving 1 (a) and 2 (b) cores decreases slowdown by 4.4x and 1.5x, respectively.

on *High Bimodal* in Perséphone. Figure 3 presents our results. c-FCFS improves the tail latency of short requests over d-FCFS by eliminating local hotspots at workers, a result consistent with previous work [75]. However, because c-FCFS does not protect the service time distribution of short requests, they experience dispersion-based blocking from long requests. With c-FCFS, short requests experience 309μs p99.9 latency at 260kRPS, driving slowdown for the entire workload to 283x. In contrast, DARC reserves 1 core for short requests and schedules them first, reducing slowdown upon c-FCFS **by a factor of 15.7**[2] and can sustain **2.3x higher throughput** for a SLO of 20μs for short requests. This comes at the cost of up to a 4.2x increase in tail latency for long requests. The average CPU waste occasioned by DARC is 0.86 core. Because slowdown is driven by short requests and the two graphs are very similar, we omit short requests in the next sections and focus on overall slowdown and tail latency for long requests.

5.3 How much non work-conservation is useful?

We empirically validate DARC's reservation mechanism (Sec. 3) by manually configuring the number of workers dedicated to short requests from 0 to 14. We call this version "DARC-static". It schedules short requests first and let them execute on all the cores. When the number of reserved workers is 0, DARC-static is equivalent to a simple Fixed Priority policy favoring short requests. Figure 4 presents the overall slowdown experienced by *High Bimodal* (a) and *Extreme Bimodal* (b) at 95% load. We observe that for the former, the best slowdown — a 4.4x improvement — is achieved with 1 core, and for the latter with 2 cores — a 1.5x improvement. Those settings validate DARC's selection, as described in Sec. 5.2 and Sec. 5.4.

For comparison, we draw the slowdown line offered by c-FCFS on Perséphone. Reserving too many workers results in long requests being starved. Simple Fixed Priority scheduling results in dispersion-based blocking for short requests.

5.4 Comparison with Shinjuku and Shenango

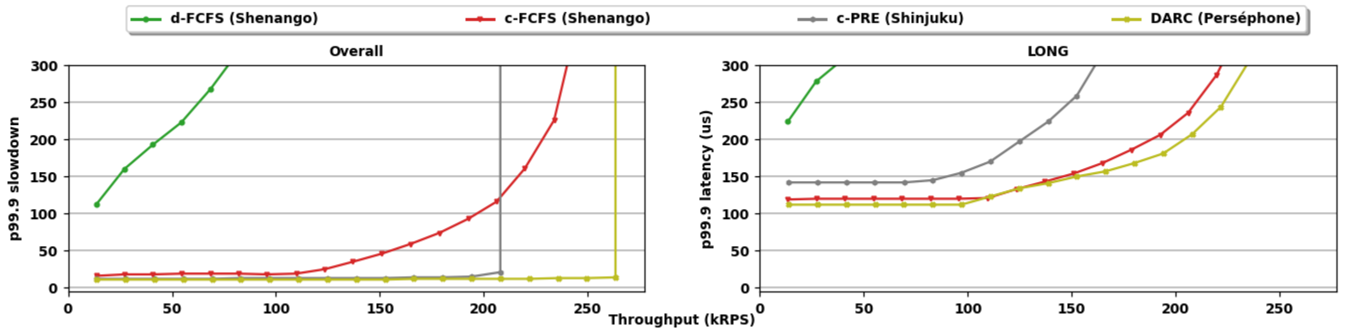
Figures 6a and 6b show the performance experienced by *High Bimodal* and *Extreme Bimodal* in all three systems. Figure 6 presents *TPC-C* performance, and Figure 8 RocksDB performance. Shenango implements d-FCFS and c-FCFS. Shinjuku uses its multi-queue policy for *High Bimodal*, *TPC-C*, and RocksDB; and its single queue policy for *Extreme Bimodal* (per the Shinjuku paper [48]). We invested significant efforts in tuning Shinjuku for short requests performance and preempting as frequently as possible. We could only sustain 75% for *High Bimodal* (5μs interrupts) and RocksDB (15μs interrupts), and 55% load for *Extreme Bimodal* (5μs interrupts), after which the system starts dropping packets and eventually crashes (despite sustaining close to 4.5 million 1μs RPS without preemption on our testbed). We found that reducing the frequency of preemption helped sustain higher loads at the expense of shorter requests. *TPC-C* is most favorable to Shinjuku because the services times are higher and dispersion smaller. Shinjuku can handle 85% of this load when preempting every 10μs.

5.4.1 ***High Bimodal.*** Shinjuku improves the tail latency of short requests over Shenango's c-FCFS by preempting long requests. However, Shinjuku aggressively preempts every 5μs to maintain good latency for short requests and adds a constant overhead — at least 20% in this experiment — to preempted requests. As a result, it can sustain only 75% of the load before dropping requests. In comparison, DARC reserves 1 core for short requests and can sustain **2.35x and 1.3x more load** than Shenango and Shinjuku, respectively, for a target slowdown of 20x. At 75% load, DARC **reduces slowdown by 10.2x and 1.75x over Shenango and Shinjuku**, respectively. Perhaps more importantly, compared to Shinjuku's preemption system, DARC consistently provides better tail latency for long requests. We also observe that Perséphone's centralized scheduling offers better performance for long requests than Shenango — compared to the c-FCFS line in figure 3 — because Perséphone does not have to approximate centralization with work stealing.

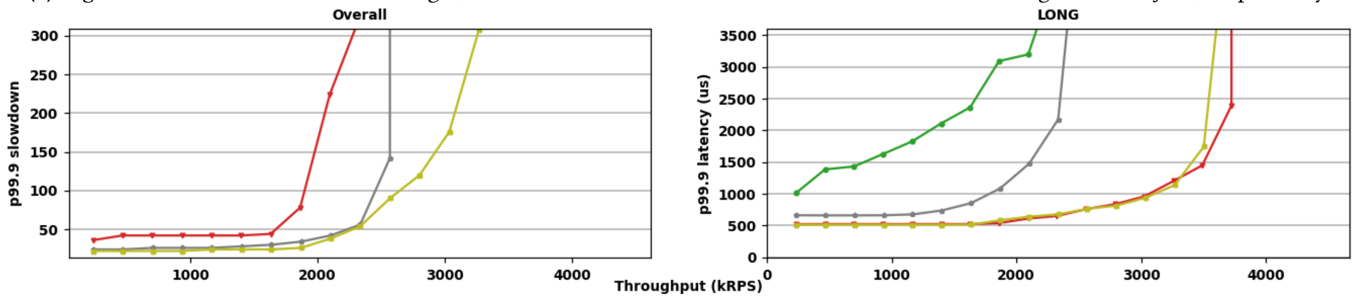
5.4.2 ***Extreme Bimodal.*** We observe similar trends for this workload. For a target 50x slowdown, both Shinjuku and Perséphone can sustain **1.4x higher throughput** than

[2]The network contributes 10μs to response time. At 260kRPS, short requests experience 309μs end-to-end latency with c-FCFS and 18μs with DARC. This means that **server-side slowdown is 37x better with DARC**.

Figure 5



(a) *High Bimodal* For a 20x slowdown target, DARC can sustain 2.35x and 1.3x more traffic than Shenango and Shinjuku, respectively.



(b) *Extreme Bimodal* For a 50x slowdown target, DARC and Shinjuku can sustain 1.4x more load than Shenango. In addition, DARC reduces short requests' slowdown up to 1.4x compared to Shinjuku. Note the different Y axis for slowdown and long requests tail latency.

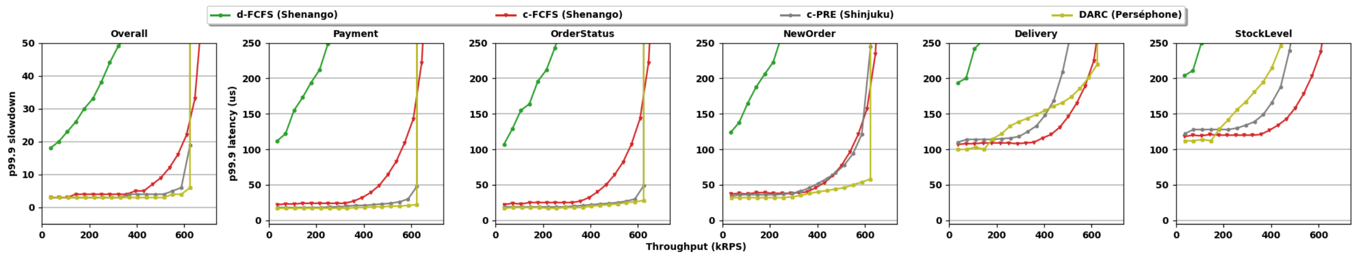


Figure 6. *TPC-C* with Shenango, Shinjuku and Perséphone. The first column is the p99.9 slowdown across all transactions. Each subsequent column is the p99.9 latency for a given transaction. Transactions are presented in ascending average service time. Note the different Y axis for slowdown and latency. At 85% load, Perséphone offers 9.2x, 7x, and 3.6x improved p99.9 latency to `Payment` (b), `OrderStatus` (c) and `NewOrder` (d) transactions, compared to Shenango's c-FCFS, reducing overall slowdown by up to 4.6x (a). For a slowdown target of 10x, Perséphone can sustain 1.2x and 1.05x more throughput than Shenango and Shinjuku, respectively.

Shenango. However, past 55% load, the overheads of aggressively preempting every 5μs is too expensive and Shinjuku starts dropping packets. For long requests, preemption overheads are always at least 24% (620μs for 500μs requests). In contrast, Perséphone reserves 2 cores to maintain good tail latency for short requests and can sustain **1.25x more load** while **reducing slowdown up to 1.4x** over Shinjuku. All the while, Perséphone provides tail latency for long requests competitive with Shenango. For this workload the CPU waste occasioned by DARC is, on average, 0.67 core.

5.4.3 *TPC-C.*

For this workload, DARC groups `Payment` and `OrderStatus` transactions (group A), lets `NewOrder` transactions run in their own group (B), and groups `Delivery` and `StockLevel` transactions (group C). DARC attributes workers 1 and 2 to group A, 3 − 8 to group B, and 9 − 14 to group C. Group A can steal from workers 3−14, group B from workers 9−14, and group C cannot steal. There is no average CPU waste with this allocation because groups A and B are slightly under-provisioned and can steal from C. Figure 6 presents our findings. DARC strongly favors shorter transactions from groups A and B. Compared to Shenango's c-FCFS, DARC provides up to **9.2x, 7x and 3.6x better tail latency** to `Payment`, `OrderStatus` and `NewOrder` transactions, respectively. These transactions represent 92% of the workload, resulting in up to **4.6x slowdown reduction** at the cost of

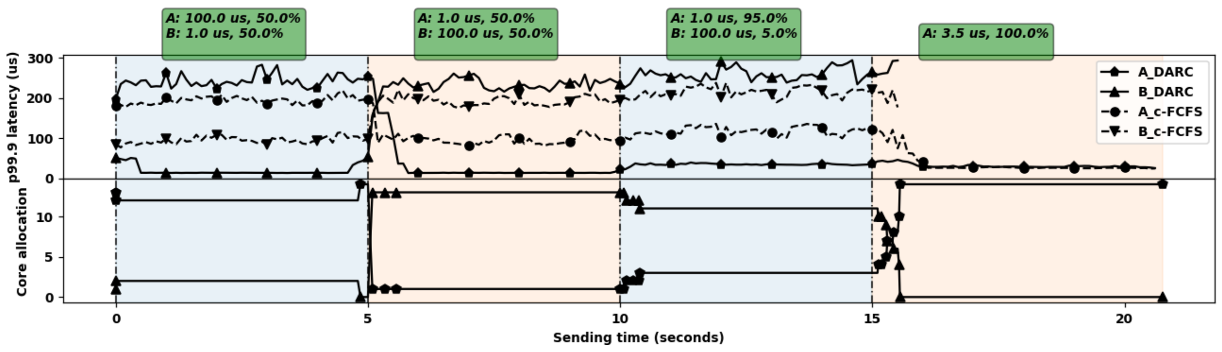


Figure 7. p99.9 latency and guaranteed cores for two request types A and B during 4 phases, under c-FCFS and DARC. Top boxes describe phases (service times and ratios). During transitions, Perséphone's profiler picks on the new service time and ratio for each type and accordingly adjusts core allocation. Markers for the core allocation row indicate reservation update events.

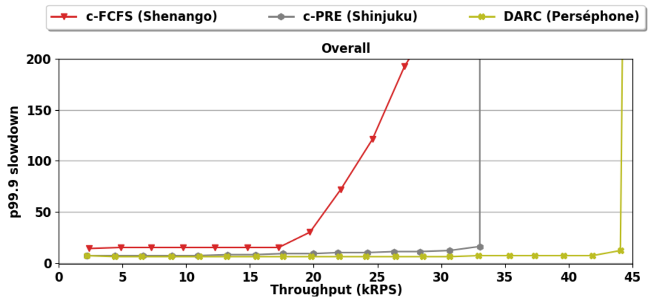


Figure 8. RocksDB slowdown with 50% GETs (1.5μs), 50% SCANs (635μs). For a 20x slowdown target, DARC can sustain 2.3x and 1.3x higher throughput than Shenango and Shinjuku, respectively.

5% throughput from the longer StockLevel transactions. Because DARC excludes the longer Delivery and StockLevel transactions from 8 out of 14 workers, those transactions suffer higher tail latency compared to Shenango's c-FCFS. Interestingly, however, due to DARC's priority-based scheduling, Delivery transactions experience tail latency competitive with c-FCFS at high load. In addition, though benefiting Payment and OrderStatus requests, Shinjuku's offers performance similar to c-FCFS for the moderately slow NewOrder requests, because it preempts them halfway to protect the shorter requests. Likewise, Delivery and StockLevel requests suffer from repetitive preemption. In contrast, DARC is able to maintain good tail latency for NewOrder requests, offers a better trade-off for Delivery and Stocklevel at high load (not show in the graph for the latter), and **reduces slowdown up to 3.1x** compared to Shinjuku.

Given a 10x overall slowdown target, **Perséphone can sustain 1.2x and 1.05x higher throughput** than Shenango and Shinjuku, respectively.

5.4.4 RocksDB.

We use Perséphone to build a service running RocksDB and create a Shenango application running a similar RocksDB service. Shinjuku already implements a RocksDB service. The database is backed by a file pinned in memory. We use the same workload as Shinjuku's: 50% GET requests and 50% SCAN requests over 5000 keys. On our testbed, GETs execute in 1.5μs and SCANs in 635μs. Consistently with previous experiments, we were able to sustain only about 75% of the theoretical peak load with Shinjuku using a 15μs preemption timer and its multi-queue policy. We omit d-FCFS because it offers poor performances. DARC reserves 1 core for GET requests, idling 0.96 core on average. Figure 8 presents slowdown for this experiment: **for a 20x slowdown QoS objective, DARC can sustain 2.3x and 1.3x higher throughput than Shenango and Shinjuku, respectively**.

5.5 Handling workload changes

In this section, we demonstrate Perséphone capacity to react to sudden changes in workload composition. For comparison with a baseline, we include c-FCFS performance. The experiment studies three phase changes: (1) fast requests suddenly become slow and *vice-versa* (2) the ratio of each type change and (3) the workload becomes only fast requests. Across this experiment, we keep the server at 80% utilization. Each phase lasts for 5 seconds. Figure 7 presents the results. Green boxes describe phases. The first row is the 99.9th percentile latency and the second row the number of cores guaranteed to each type (not including stealable cores).

At first, B requests can run on all 14 cores — 1 dedicated core and 13 stealable cores — and A requests are allowed to run on 13 cores. Latency is slightly higher for B requests and slightly slower for A requests at the beginning of the experiment because the system starts in c-FCFS before proceeding to the first reservation. In the second phase, we inverse the service time of A and B to evaluate how DARC can handle miss-classification. During the transition, which takes about 500ms, "B-fast" requests observe increased latency — up to 50μs— as "A-slow" requests are allowed to run on all cores before the reservation update. The graph shows latency increase before the transition because these "B-fast" requests were already in the system and the X axis is the sending time.

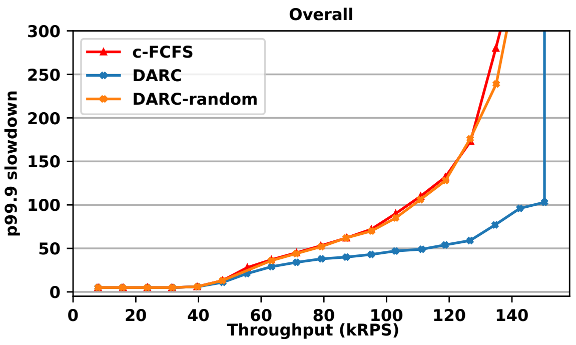


Figure 9. *High-bimodal* performance when DARC is set with a broken, random request classifier. DARC-random's behavior converges to c-FCFS.

During the second transition, we change the ratio of each type: A requests now make up 99.5% of the mix. As a result, their CPU demand increases and DARC reserves them 2 cores. For this new composition, 80% utilization on the server results in increased throughput, and latency becomes slightly higher for both types of requests as all queues grow larger.

Finally, we change the workload to be only made of A requests. Despite A requests being able to run on all 14 cores, pending B requests can be serviced on the spillway core.

5.6 Random classifier

Finally, we evaluate DARC's behavior when users fail to provide a correct request classifier. We modify the dispatcher to push incoming requests to a random typed queue. We expect each typed queue to hold an equal share of each request type, thus converging to c-FCFS. We run High-bimodal on a two nodes setup (one server with 8 worker threads and one client, both running on Silver 4114 Xeon CPUs and using Mellanox Connectx-5 cards). Figure 9 presents the results. DARC-random uses the random classifier. As expected, DARC-random and c-FCFS exhibit similar behaviors.

6 Discussion

Networking model. In the current implementation, the net worker is a layer 2 forwarder and performs simple checks on Ethernet and IP headers. Application workers handle layers 4 and above and directly perform TX. This design intends to maximize our dispatcher's performance — the main bottleneck in Perséphone— and make it competitive with existing systems. Shenango and Shinjuku separate roles in a similar way. There is no fundamental reason, though, for not having the net worker handle more of the network stack Using a stateful network stack would preclude offloading TX to the workers since shared state between the net worker and application workers would hinder performance. For TCP, this problem is partly addressed by TAS [53] and Snap [62].

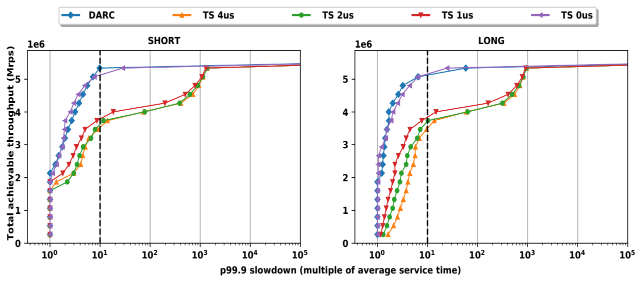


Figure 10. Simulating single queue preemptive systems with various overheads. At the microsecond scale, implementing and orchestrating preemptive systems remains challenging.

Interrupts at μs scale. Though desirable in theory because it enables a better approximation to SRPT, interrupts at the microsecond scale are challenging to implement. In addition to operating overheads, adequate timing is also critical for protecting short requests. Consider the simulation from section 2. We extend this analysis with a set of single queue preemptive systems with different interrupt overheads and delay. We assume a preemption event can be triggered as soon as a short request is blocked in the queue by long requests running on workers. The first system, "TS 4μs" takes 2μs to propagate a preemption event to a worker, and 2 additional μs to preempt the running request. "TS 2μs" and "TS 1μs" operate similarly. "TS 0μs" represents an ideal system with instant preemption and no overheads. Figure 10 shows their impact on requests slowdown, compared to DARC. Unsurprisingly, the ideal system performs similarly or better than DARC. As preemption speed increases, short requests are better protected and long requests suffer less in the process. However, at the microsecond scale, even 1μs overheads result in 30% less sustainable load, for a slowdown target of 10 for the short requests. In response to implementation and tuning challenges for preemption at the μs scale, DARC proposes a complementary resource partitioning technique that alleviates the need for interrupts. Finally, preemption comes with a second class of challenges related to practicality. One has to carefully re-work existing applications to ensure preemption cannot happen during critical sections — memory management, interaction with thread local storage, etc. — or non re-entrant functions. This represents considerable efforts and encourages other designs trade-off such as semi-cooperative scheduling [17].

DARC in the datacenter ecosystem. Though not a focus of this paper, DARC can cooperate with an allocator to obtain and release cores, adapting to load changes and updating reservations during such events. In the event of a system overload, DARC will keep prioritizing short requests as far as possible, triggering flow control for longer requests first.

7 Related work

Kernel-bypass. Bypassing a general-purpose kernel to provide application-tailored services has been revisited regularly over the past fifty years. Some notable examples include the RC 4000 multi-programming system [37], Hydra [88], Mach [1], Chorus [77], SPIN [16], and Exokernel [31].

More recently, faster networks and stagnating CPU speeds have led researchers to look more closely at user-level network stacks [47, 53], to provide high-performance storage systems [19, 55, 58, 71], access to disaggregated memory [4], user-level network services [53] such as eRPC [49], and fast I/O processing (*e.g.*, IX [14] and Chronos [50]). Similarly, user-level scheduling has been explored with ZygOS [75] and Shinjuku [48], which focus on improving tail latency by implementing centralized dispatch policies and user-level preemption, both of which outperform current decentralized offerings, as is well understood by theory [57, 87, 90]. DARC builds on this recent line of work with a more application-customized solution, motivated by recent insights when observing performance gain from sharing application-level information with the dataplane [23, 52, 61], and "common case service acceleration", which can improve tail latency for important requests [64]. DARC multi-queue policy could be integrated with existing kernel-bypass schedulers [48, 73, 75], with the merit that programmers could decide whether providing request classifiers is worthwhile for their workload. In fact, we originally designed Perséphone for the Demikernel Library OS Architecture [89], with the goal of integrating user-level insights to a wide range of datapaths.

Scheduling Policies: Recent works on kernel-bypass and microsecond-scale applications have revived research interest in scheduling policies, specifically for tail-tolerance. We compare DARC with existing policies proposed for process or packet scheduling, and identified the best fit for each. Table 5 summarizes our findings. DARC shares ideas with Fixed Priority (FP) scheduling without suffering from head-of-line blocking and with Cycle Stealing with Central Queue (CSCQ [42]), but does not impose limits on stealing for shorter requests. It also shares ideas with Static Partitioning (SP) without being as work conservation avoidant, thus being able to absorb bursts. DARC targets applications with high service time dispersion similarly to Processor Sharing policies, implemented as the Completely Fair Scheduler [67], Borrowed Virtual Time [29], and Multi-Level Feedback Queue [6] in commodity operating systems and variants of these on datacenter operating systems [48]. Processor sharing policies, despite being application agnostic, are expensive or impossible to implement in many environments, *e.g.*, the public cloud. DARC is, to our best knowledge, the first application-aware and non-preemptive policy that classifies requests to improve RPC tail latency and can be implemented on a kernel-bypassed system serving microsecond-scale requests. We note that existing work has specifically made use of non work conservation to reduce resource contention in SMT architectures [33, 76, 82], though with a focus on instruction throughput rather than tail latency for datacenter workloads.

In-network end-host scheduling. R2P2 [56] and Metron [51] propose to integrate core scheduling in the network. Loom [83] proposes a novel NIC design and OS/NIC abstraction to express rich hierarchies of network scheduling and traffic shaping policies across tenants. Our work is orthogonal since request classifiers can be offloaded to the network. eRSS [78] scaling groups offer the possibility to schedule request groups, which works only on network headers and requires a specific programming model from the NIC. RSS++ [9] addresses RSS vulnerability to traffic imbalance but cannot handle variability in application-request processing times. Intel recently introduced Application Device Queues (ADQ) [44], a feature for applications to reserve NIC hardware queues. ADQ requires specific network interfaces (currently Intel's Ethernet 800 Series) and does not allow applications to further partition reserved queues by request type. Finally, recent progress in network hardware could enable instantiating DARC either in a programmable switch using a PIFO scheduling transaction [81], or directly in the end-host hardware using the NanoPU hardware thread scheduler [43] or a SmartNIC.

Network scheduling for tail latency. Prioritizing packets to improve tail latency has been extensively studied in the networking literature [3, 8, 34, 36, 59, 68, 69]. As analyzed by Mushtaq et al. [70], this line of work uses priority queues in switches to approximate Shortest Remaining Processing Time (SRPT) scheduling and avoid head-of-line blocking caused by FIFO policies. Dedicating more CPU resources to short requests is similar to prioritizing packets belonging to short flows, but whereas network devices schedule at the granularity of packets — bounded by MTU sizes — and preempt long flows by not sending their packets, there is no affordable way to preempt a long request once dispatched at a CPU core within microseconds. DARC efficiently partitions CPU resources among requests by profiling their CPU demand and enabling work conservation only for short requests, capping resources allocated to long requests and resulting in a similar trade off than Homa [68], pFabric [3], or HULL [2].

Other efforts to improve tail latency. Haque et al. [39] exploit DVFS and heterogeneous CPUs to speed up long requests in latency sensitive workloads at the expense of short requests, with the goal of improving overall tail latency. Our technique is orthogonal to such optimization, since DARC defines a clear target to configure power and core settings for given request types. Another line of work adapts the degree of parallelism of long requests and improves overall tail latency [38, 46], but this comes at the cost of shorter requests

Table 5. Summary of different scheduling policies as comparison points to DARC

Policy	App Aware	Non preemptive	Non Work Conserving	Prevent HOL	Ideal Workload	Comments
d-FCFS	✗	✓	✓	✗	Light-tailed	Easy to implement Load Imbalance
c-FCFS	✗	✓	✗	✗	Light-tailed	Ideal policy for the workload
Processor Sharing (Linux CFS, BVT [28], MLFQ [6])	✗	✗	✗	✓	Heavy-tailed without priorities	Hard to implement
(Deficit) (Weighted) Round Robin	✗	✓	✗	✗	Request flows with fairness requirements	No latency guarantees
Static Partitioning	✓	✓	✓	✗	Different request types with different SLOs	Inflexible with rapid workload changes
Fixed Priority	✓	✓	✗	✗	Request priority indendent of service time	Can lead to priority inversion
Earliest Deadline First	✓	✓	✗	✗	Request priority indendent of service time	Requires clock sync
Shortest Job First	✓	✓	✗	✗	Custom	Can starve long RPCs
SRPT	✓	✗	✗	✓	Heavy-tailed	Optimal for average latency Hard to implement
Cycle Stealing with Central Queue	✓	✓	✓	✗	Mix of short and long requests with the same priority	Can absorb short RPCs bursts
DARC	✓	✓	✓	✓	Heavy-tailed with high priority short requests	Favor short RPCs over longs

from which more resources are taken away. Mirhosseini et al. modify another of Mor Harchol-Balter's typed-queue policy, SITA [41], to prevent dispersion-based blocking in hardware queues [65]. RobinHood [15] improves tail latency by provisioning more cache to backends that affect such latency. Minos [24] shards data based on size to reduce GETs variability across shards. Finally, isolation techniques such as PerfIso [45] also eschew work conservation to protect latency critical tasks, but are strictly less efficient than work conserving, preemption-based techniques. In contrast, at microsecond latencies, the trade-off between preemption and idling changes, making our non-work-conserving kernel-bypass scheduler a better optimized solution.

8 Conclusion

This paper proposes Perséphone, a new kernel-bypass OS scheduler implementing DARC, an application aware, non work conserving policy. DARC maintains good tail latency for shorter requests in heavy-tailed workloads that cannot afford the overheads of existing techniques such as work stealing and preemption. DARC profiles requests and dedicates cores to shorter requests, guaranteeing they will not be blocked behind long requests. Our prototype of Perséphone maintains good tail latency for shorter requests and can handle higher loads with the same amount of cores than state-of-the-art kernel-bypass schedulers, overall better utilizing datacenter resources.

9 Acknowledgment

We thank our Shepherd Jonathan Mace as well as the OSDI and SOSP anonymous reviewers for their valuable inputs. We also extend thanks to the anonymous artifact evaluation committee for their efforts in reproducing our results. Daniel Berger pointed us to CSCQ and Mor Harchol-Balter's work in general. Various friends helped proof read the paper, including Tom Anderson, Adam Belay, and Philip Levis. This work was partially supported by NSF CNS-1750158, CNS-1513679 and CNS-1563873.

References

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for unix development.

[2] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, Apr. 2012), USENIX Association, pp. 253–266.

[3] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. Pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, Association for Computing Machinery, p. 435–446.

[4] Amaro, E., Branner-Augmon, C., Luo, Z., Ousterhout, A., Aguilera, M. K., Panda, A., Ratnasamy, S., and Shenker, S. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[5] Amvrosiadis, G., Park, J. W., Ganger, G. R., Gibson, G. A., Baseman, E., and DeBardeleben, N. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 533–546.

[6] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces*, 1.00 ed. Arpaci-Dusseau Books, August 2018.

[7] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference*

on Measurement and Modeling of Computer Systems (New York, NY, USA, 2012), SIGMETRICS '12, Association for Computing Machinery, p. 53–64.

[8] Bai, W., Chen, L., Chen, K., Han, D., Tian, C., and Wang, H. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, May 2015), USENIX Association, pp. 455–468.

[9] Barbette, T., Katsikas, G. P., Maguire, G. Q., and Kostić, D. Rss++: Load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (New York, NY, USA, 2019), CoNEXT '19, Association for Computing Machinery, p. 318–333.

[10] Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. Attack of the killer microseconds. *Commun. ACM 60*, 4 (Mar. 2017), 48–54.

[11] Barroso, L. A., Hölzle, U., and Ranganathan, P. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture 13*, 3 (2018), i–189.

[12] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, Association for Computing Machinery, p. 29–44.

[13] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (USA, 2012), OSDI'12, USENIX Association, p. 335–348.

[14] Belay, A., Prekas, G., Primorac, M., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst. 34*, 4 (Dec. 2016), 11:1–11:39.

[15] Berger, D. S., Berg, B., Zhu, T., Sen, S., and Harchol-Balter, M. Robinhood: Tail latency aware caching – dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 195–212.

[16] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, Association for Computing Machinery, p. 267–283.

[17] Boucher, S., Kalia, A., Andersen, D. G., and Kaminsky, M. Lightweight preemptible functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 465–477.

[18] Cao, Z., Tarasov, V., Raman, H. P., Hildebrand, D., and Zadok, E. On the performance variation in modern storage stacks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, Feb. 2017), USENIX Association, pp. 329–344.

[19] Chen, Y., Lu, Y., and Shu, J. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[20] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, Association for Computing Machinery, p. 143–154.

[21] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, Association for Computing Machinery, p. 205–220.

[22] Demoulin, H. M., Pedisich, I., Vasilakis, N., Liu, V., Loo, B. T., and Phan, L. T. X. Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 693–708.

[23] Demoulin, H. M., Vasilakis, N., Sonchack, J., Pedisich, I., Liu, V., Loo, B. T., Phan, L. T. X., Smith, J. M., and Zhang, I. TMC: Pay-as-You-Go Distributed Communication. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019* (New York, NY, USA, 2019), APNet '19, Association for Computing Machinery, p. 15–21.

[24] Didona, D., and Zwaenepoel, W. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (USA, 2019), NSDI'19, USENIX Association, p. 79–93.

[25] Direct, I. D. I/o technology (intel ddio) a primer, 2012.

[26] DPDK. Data plane development kit. https://www.dpdk.org/. Accessed: 2020-10-27.

[27] Drew Gallatin, Netflix Technology Blog. Serving 100 Gbps from an Open Connect Appliance. https://netflixtechblog.com/serving-100-gbps-from-an-open-connect-appliance-cdb51dda3b99. Accessed: 2020-12-04.

[28] Duda, K. J., and Cheriton, D. R. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose schedular. In *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999* (1999), D. Kotz and J. Wilkes, Eds., ACM, pp. 261–276.

[29] Duda, K. J., and Cheriton, D. R. Borrowed-virtual-time (bvt) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1999), SOSP '99, Association for Computing Machinery, p. 261–276.

[30] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., and Mishra, P. The design and operation of cloudlab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 1–14.

[31] Engler, D. R., Kaashoek, M. F., and O'Toole, J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, Association for Computing Machinery, p. 251–266.

[32] Facebook. Rocksdb. https://rocksdb.org/. Accessed: 2020-11-17.

[33] Fedorova, A., Seltzer, M., and Smith, M. D. A non-work-conserving operating system scheduler for smt processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA* (2006), vol. 33, pp. 10–17.

[34] Gao, P. X., Narayan, A., Kumar, G., Agarwal, R., Ratnasamy, S., and Shenker, S. Phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2015), CoNEXT '15, Association for Computing Machinery.

[35] Google. Protocol Buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf. Accessed: 2020-12-09.

[36] Grosvenor, M. P., Schwarzkopf, M., Gog, I., Watson, R. N. M., Moore, A. W., Hand, S., and Crowcroft, J. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, May 2015), USENIX Association, pp. 1–14.

[37] Hansen, P. B. The nucleus of a multiprogramming system. *Commun. ACM 13*, 4 (Apr. 1970), 238–241.

[38] Haque, M. E., Eom, Y. h., He, Y., Elnikety, S., Bianchini, R., and

McKinley, K. S. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, Association for Computing Machinery, p. 161–175.

[39] Haque, M. E., He, Y., Elnikety, S., Nguyen, T. D., Bianchini, R., and McKinley, K. S. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2017), MICRO-50 '17, Association for Computing Machinery, p. 625–638.

[40] Harchol-Balter, M. *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press, 2013.

[41] Harchol-Balter, M., Crovella, M. E., and Murta, C. D. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing 59*, 2 (1999), 204–228.

[42] Harchol-Balter, M., Li, C., Osogami, T., Scheller-Wolf, A., and Squillante, M. S. Cycle stealing under immediate dispatch task assignment. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 2003), SPAA '03, Association for Computing Machinery, p. 274–285.

[43] Ibanez, S., Mallery, A., Arslan, S., Jepsen, T., Shahbaz, M., Kim, C., and McKeown, N. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021), USENIX Association, pp. 239–256.

[44] Intel. Adq. https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/application-device-queues-technology-brief.html. Accessed: 2020-11-10.

[45] Iorgulescu, C., Azimi, R., Kwon, Y., Elnikety, S., Syamala, M., Narasayya, V., Herodotou, H., Tomita, P., Chen, A., Zhang, J., and Wang, J. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 519–532.

[46] Jeon, M., He, Y., Kim, H., Elnikety, S., Rixner, S., and Cox, A. L. Tpc: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. *SIGPLAN Not. 51*, 4 (Mar. 2016), 129–141.

[47] Jeong, E. Y., Woo, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. Mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (USA, 2014), NSDI'14, USENIX Association, p. 489–502.

[48] Kaffes, K., Chong, T., Humphries, J. T., Belay, A., Mazières, D., and Kozyrakis, C. Shinjuku: Preemptive scheduling for msecond-scale tail latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (USA, 2019), NSDI'19, USENIX Association, p. 345–359.

[49] Kalia, A., Kaminsky, M., and Andersen, D. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (2019), USENIX Association, pp. 1–16.

[50] Kapoor, R., Porter, G., Tewari, M., Voelker, G. M., and Vahdat, A. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, Association for Computing Machinery.

[51] Katsikas, G. P., Barbette, T., Kostić, D., Steinert, R., and Jr., G. Q. M. Metron: NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, Apr. 2018), USENIX Association, pp. 171–186.

[52] Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., and Krishnamurthy, A. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, Association for Computing Machinery, p. 67–81.

[53] Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A., and Anderson, T. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[54] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 3146–3154.

[55] Klimovic, A., Litz, H., and Kozyrakis, C. Reflex: Remote flash ≈ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, Association for Computing Machinery, p. 345–359.

[56] Kogias, M., Prekas, G., Ghosn, A., Fietz, J., and Bugnion, E. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 863–880.

[57] Li, J., Sharma, N. K., Ports, D. R. K., and Gribble, S. D. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, Association for Computing Machinery, p. 1–14.

[58] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (USA, 2014), NSDI'14, USENIX Association, p. 429–444.

[59] Lu, Y., Chen, G., Luo, L., Tan, K., Xiong, Y., Wang, X., and Chen, E. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications* (2017), IEEE, pp. 1–9.

[60] Marek Majkowski, The Cloudflare Blog. How to achieve low latency with 10Gbps Ethernet. https://blog.cloudflare.com/how-to-achieve-low-latency/. Accessed: 2020-12-04.

[61] Marinos, I., Watson, R. N., and Handley, M. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, Association for Computing Machinery, p. 175–186.

[62] Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S., and et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 399–413.

[63] memcached. Memcached protocol. https://github.com/memcached/memcached/blob/master/doc/protocol.txt. Accessed: 2021-03-24.

[64] Mirhosseini, A., and Wenisch, T. F. The queuing-first approach for tail management of interactive services. *IEEE Micro 39*, 4 (2019), 55–64.

[65] Mirhosseini, A., West, B. L., Blake, G. W., and Wenisch, T. F. Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), IEEE, pp. 207–219.

[66] Misra, P. A., Borge, M. F., Goiri, I. n., Lebeck, A. R., Zwaenepoel, W., and Bianchini, R. Managing tail latency in datacenter-scale file systems under production constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[67] Molnar, I. [patch] modular scheduler core and completely fair scheduler. https://lwn.net/Articles/230501/. Accessed: 2020-12-01.

[68] Montazeri, B., Li, Y., Alizadeh, M., and Ousterhout, J. Homa: A

receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 221–235.

[69] Munir, A., Baig, G., Irteza, S. M., Qazi, I. A., Liu, A. X., and Dogar, F. R. Friends, not foes: Synthesizing existing transport strategies for data center networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, Association for Computing Machinery, p. 491–502.

[70] Mushtaq, A., Mittal, R., McCauley, J., Alizadeh, M., Ratnasamy, S., and Shenker, S. Datacenter congestion control: Identifying what is essential and making it practical. *SIGCOMM Comput. Commun. Rev. 49*, 3 (Nov. 2019), 32–38.

[71] Nanavati, M., Wires, J., and Warfield, A. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (2017), USENIX Association, pp. 17–33.

[72] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al. Scaling memcache at facebook. In *Proc. of NSDI* (2013), pp. 385–398.

[73] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2019), NSDI'19, USENIX Association, pp. 361–377.

[74] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (USA, 2014), OSDI'14, USENIX Association, p. 1–16.

[75] Prekas, G., Kogias, M., and Bugnion, E. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 325–341.

[76] Rosti, E., Smirni, E., Serazzi, G., and Dowdy, L. W. Analysis of non-work-conserving processor partitioning policies. In *Workshop on Job Scheduling Strategies for Parallel Processing* (1995), Springer, pp. 165–181.

[77] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Léonard, P., et al. Overview of the chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures* (1992), pp. 39–70.

[78] Rucker, A., Shahbaz, M., Swamy, T., and Olukotun, K. Elastic rss: Co-scheduling packets and cores using programmable nics. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019* (New York, NY, USA, 2019), APNet '19, Association for Computing Machinery, p. 71–77.

[79] Schrage, L. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research 16*, 3 (1968), 687–690.

[80] Shanley, T. *InfiniBand network architecture.* Addison-Wesley Professional, 2003.

[81] Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S.-T., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., and McKeown, N. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, Association for Computing Machinery, p. 44–57.

[82] Smirni, E., Rosti, E., Serazzi, G., Dowdy, L. W., and Sevcik, K. C. Performance gains from leaving idle processors in multiprocessor systems. In *Proceedings of the 1995 International Conference on Parallel Processing, Urbana-Champain, Illinois, USA, August 14-18, 1995. Volume III: Algorithms & Applications* (1995), K. A. Gallivan, Ed., CRC Press, pp. 203–210.

[83] Stephens, B., Akella, A., and Swift, M. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 33–46.

[84] TPC. Tpc-c. http://www.tpc.org/tpcc/. Accessed: 2020-10-20.

[85] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 18–32.

[86] Wei Dai. Redis Protocol specification. https://redis.io/topics/protocol. Accessed: 2021-05-05.

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

[88] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM 17*, 6 (June 1974), 337–345.

[89] Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O., Martinez, A., Liu, J., Simpson, A., Jayakar, S., Penna, P., Demoulin, M., Choudhury, P., and Badam, A. The Demikernel Library OS Architecture for Microsecond, Kernel-Bypass Datacenter Systems Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the 27th Symposium on Operating Systems Principles*, SOSP '21, Association for Computing Machinery.

[90] Zhang, Y., Meisner, D., Mars, J., and Tang, L. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *Proceedings of the 43rd International Symposium on Computer Architecture* (2016), ISCA '16, IEEE Press, p. 456–468.