TailClipper: Reducing Tail Response Time of Distributed Services Through System-Wide Scheduling

Nathan Ng
University of Massachusetts Amherst
kwanhong@cs.umass.edu

Abel Souza
University of California Santa Cruz
absouza@ucsc.edu

Ahmed Ali-Eldin
Chalmers University of Technology
ahmed.hassan@chalmers.se

David Irwin
University of Massachusetts Amherst
irwin@ecs.umass.edu

Don Towsley
University of Massachusetts Amherst
towsley@cs.umass.edu

Prashant Shenoy
University of Massachusetts Amherst
shenoy@cs.umass.edu

ABSTRACT

Reducing tail latency has become a crucial issue for optimizing the performance of online cloud services and distributed applications. In distributed applications, there are many causes of high end-to-end tail latency, including operating system delays, request re-ordering due to fan-out/fan-in, and network congestion. Although recent research has focused on reducing tail latency for individual application components, such as by replicating requests and scheduling, in this paper, we argue for a holistic approach for reducing the end-to-end tail latency across application components. We propose TailClipper, a distributed scheduler that tags each arriving request with an arrival timestamp, and propagates it across the microservices' call chain. TailClipper then uses arrival timestamps to implement an oldest request first scheduler that combines global first-come first serve with a limited form of processor sharing to reduce end-to-end tail latency. In doing so, TailClipper can counter the performance degradation caused by request reordering in multi-tiered and microservices-based applications. We implement TailClipper as a userspace Linux scheduler and evaluate it using cloud workload traces and a real-world microservices application. Compared to state-of-the-art schedulers, our experiments reveal that TailClipper improves the 99^{th} percentile response time by up to 81%, while also improving the mean response time and the system throughput by up to 54% and 29% respectively under high loads.

SoCC '24, November 20–22, 2024, Redmond, WA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698554

CCS CONCEPTS

• **Software and its engineering → Scheduling**; • **Computer systems organization → Cloud computing**.

KEYWORDS

Cloud computing, scheduling, tail latency reduction

ACM Reference Format:
Nathan Ng, Abel Souza, Ahmed Ali-Eldin, David Irwin, Don Towsley, and Prashant Shenoy. 2024. TailClipper: Reducing Tail Response Time of Distributed Services Through System-Wide Scheduling. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3698038.3698554

1 INTRODUCTION

Today's cloud platforms run on a plethora of distributed web services in domains such as finance, news, and entertainment. Many modern distributed services employ a containerized microservices architecture, where application functionality is partitioned into independent containerized services that interact with each other via well-defined interfaces (e.g., REST or gRPC). In contrast to traditional multi-tiered web applications that consist of a few tiers, microservices-based applications can consist of tens or hundreds of modular components [52], and requests need to undergo processing at numerous stages to complete execution. This modular design enables the independent development and deployment of each microservice component, allowing them to scale individually based on workload demands.

Optimizing the tail latency of requests is one of the key issues in enhancing the performance of distributed web services. Studies have highlighted that high tail latency can significantly increase customer abandonment rate [2, 11, 29, 33]. For example, one study found that even small increases in response times can cause a one percent reduction in e-commerce sales [2]. One of the major causes of high tail latency is the complexity of modern software and hardware stacks in distributed applications. Scheduling delays [31], garbage collection [47], energy optimizations [45], and the execution of background tasks [11] can all cause significant and random delays in the execution of a request, leading to requests being served orders of magnitude slower than the average [31]. Distributed applications exacerbate the problem since requests need to go through multiple microservices (or tiers) to complete execution, increasing the likelihood of encountering the above delays. Additionally, straggler problems can occur because request processing times can vary across different microservices.

Various approaches have been proposed to minimize the high tail latencies seen during request processing. One approach involves dedicating a CPU core to handle network interrupts or core re-allocations [31, 36]. Another approach is a modern version of the Borrowed Virtual Time (BVT) scheduler, which reduces tail latency by incorporating real-time priorities to prioritize requests that have been in the system for an extended period of time [30]. Furthermore, hardware virtualization techniques have been introduced to minimize context switch overhead, enabling efficient round-robin scheduling to prevent starvation of short requests [25]. In addition, prior work has proposed replicating requests to mitigate the high tail latency caused by stragglers [19, 46].

Notably, the extensive prior work on this topic has largely focused on optimizing the tail latency of individual application components and has not addressed the end-to-end tail latency problem seen in distributed applications, where requests require processing by multiple components. However, techniques that independently optimize tail latency for each application component do not fully address distributed request processing effects across components that can result in high end-to-end tail latencies. Specifically, local tail latency reduction techniques at a component do not have visibility into how much time each request has spent in the system since its arrival. Consequently, schedulers at later components are unable to compensate for longer delays or greater processing overheads incurred at earlier components and thus are likely to incur long end-to-end tail latencies. Moreover, requests may be reordered during end-to-end processing, with recent requests arriving at a particular component before older requests—an effect caused by variability in request processing times at early components and non-determinism in OS schedulers. Such request reordering can cause old requests that incur long processing times to fall behind in their overall progress, potentially increasing their overall tail latencies. Addressing both of these factors requires knowledge of the system-level arrival time of each request and techniques to prioritize the scheduling of older requests over more recent ones at each component. While such end-to-end system-wide scheduling has been studied for batch processing of DAG computations [4, 48], it remains relatively unexplored for latency-sensitive online services.

Motivated by these observations, this paper presents TailClipper, a distributed scheduler that implements an end-to-end approach for minimizing the tail latency of distributed cloud applications. In designing, implementing, and evaluating TailClipper, our paper makes the following contributions.

- TailClipper provides visibility into the total time spent by each request since its arrival by timestamping each incoming request with its system arrival time. This timestamp, which we refer to as the global arrival time (GAT), is then propagated horizontally—along the microservices call chain—and vertically—from microservice requests to the thread serving the request and to the OS scheduler.
- TailClipper employs a new OS scheduling approach to minimize the end-to-end tail latency that is inspired by queueing theory. Specifically, TailClipper employs Oldest Request First (ORF) scheduling that schedules requests based on their GAT tags and prioritizes older requests in the scheduler run queue over more recent ones. Since ORF is a type of global FCFS policy (gFCFS) and FCFS policies cause starvation of short requests, especially when the requests have heavy-tailed distributions, we combine ORF's global FCFS with a limited processor sharing (LPS) policy that uses fine-grain time slices to avoid starvation. TailClipper's hybrid gFCFS-LPS ORF scheduler is inspired by queueing theory results that show that global FCFS is optimal for light-tailed workloads, while processor sharing is preferred for heavy-tailed workloads.
- We implement a prototype of TailClipper using ghOSt, a userspace Linux scheduling framework that enables delegation of kernel scheduling policies to userspace [23]. We open source our implementation and evaluate it against the ghOSt implementation of Shinjuku (Shinjuku-gh) [25] and Linux CFS (CFS-gh) [37], two commonly deployed scheduling policies that only consider local information. Our experiments use a mix of synthetic workloads, cluster workload traces [32], and a real-world image classification application. Our results show that TailClipper improves upon Shinjuku-gh by up to 81% in tail latency, 54% in mean latency, and 29% in throughput under high loads.

2 BACKGROUND

This section presents background on distributed web applications, tail latency reduction techniques, and request reordering problems that arise during distributed request processing.

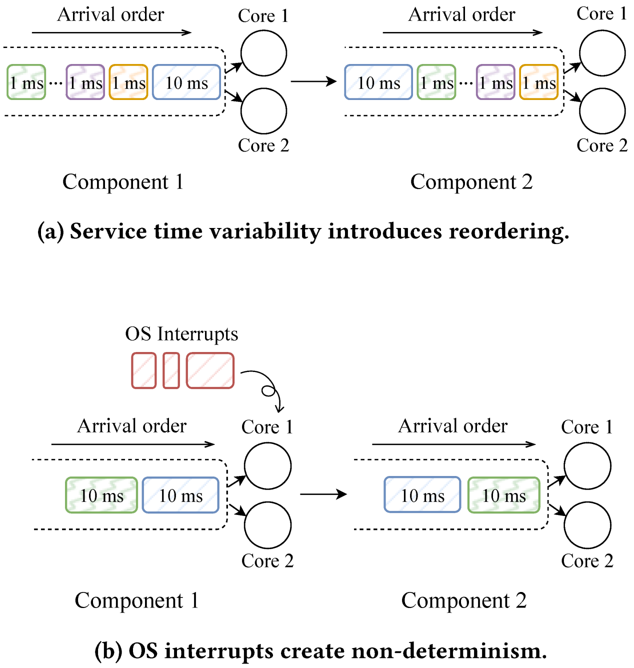
2.1 Distributed Web Applications

Modern web services that run on cloud platforms employ a distributed architecture comprising multiple tiers or components. A traditional multi-tiered application consists of a front-end HTTP tier, a middle tier comprising the business logic, and a back-end database tier [44, 52]. To further improve flexibility and scalability, web applications now increasingly adopt a microservices approach, where the business logic and data tiers are split into a number of smaller, interacting components. In both cases, incoming requests undergo processing by some subset of the application components – requests are partially processed by each component in the processing path and forwarded to the next tier. Request processing may also involve fan-outs and fork-joins across application components [2, 15]. The total end-to-end response time seen by a request is the sum of all processing and queuing delays across all components along a call chain path.

2.2 Tail Latency Reduction

There has been a wealth of research on optimizing the average response time of web requests using methods such as horizontal and vertical scaling, request scheduling, and the use of caches such as memcached [17, 31]. More recent work has emphasized tail latency reduction since it has been shown to strongly correlate with user satisfaction [11, 29]. For this reason, typical service level objectives (SLO) for web applications are often specified in terms of a bound on the tail of the response time distribution, e.g., a threshold bound on the 99^{th} percentile (P99) of response times. In the case of microservices or multi-tiered web applications, the chance of a request incurring a high latency increases due to the presence of multiple components—a performance bottleneck at any component can cause high end-to-end tail latencies. Typical performance pitfalls in microservices [11] include transient workload spikes, network bottlenecks caused by workload spikes, queuing delays in the OS, and application-level delays due to, e.g., garbage collection [47].

As noted in §1, prior work has proposed many techniques for reducing the tail latency of web services. OS-level techniques include the use of a dedicated core to handle network interrupts [36], core reallocations [31], and the use of real-time priorities to implement fair scheduling [30]. Additionally, cluster scheduling techniques such as the use of redundant requests for straggler mitigation [19, 46] have been proposed. Recent work has also incorporated higher-level application context into scheduling decisions for tail latency [3, 18, 36]. One recent example is Shinjuku [3], which is designed to reduce tail latencies through a custom scheduling policy. Shinjuku achieves up to 6.6× higher throughput and 88% lower tail latency, leading to significant performance improvements. Shenango [36] and Caladan [18] both take a systems approach and allocate a set of cores to applications, and as workload changes, core reallocations are triggered across applications. As noted earlier, these approaches focus on tail latency reduction at a single node or at a single application component.



(a) Service time variability introduces reordering.

(b) OS interrupts create non-determinism.

Figure 1: Two reasons that can cause request reordering in distributed services.

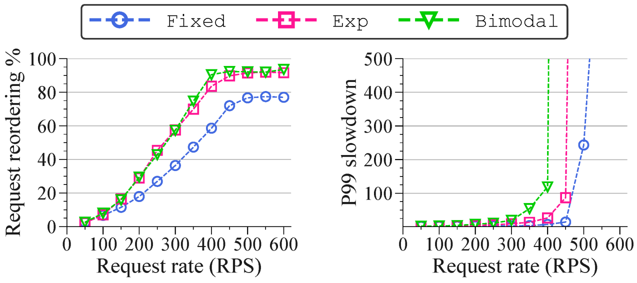
2.3 Request Reordering

Consider a distributed application that consists of multiple components such as tiers or microservices. Request processing in such applications involves a sequence of k components that execute each request, with each component performing some partial processing before handing the request to the next component in the path. Different requests may incur different processing times at each component or take different processing paths.

As such, the nature of distributed processing makes requests more susceptible to tail latency issues. One key source of increased tail latency is *request reordering*—a phenomenon where later arriving requests jump ahead of requests that arrived before them at some stage in the request processing path, which increases the queuing delay and the overall latency incurred by such requests [31]. Request reordering across components occurs for two reasons. First, when the request processing demand varies across requests, requests with shorter processing times will rapidly complete execution at one component and proceed to the next one, while those with longer processing times will spend more time at that component and fall behind. A natural consequence of requests with greater processing demands is that they take longer to complete; if they also incur longer queuing delays during their execution, this can also impact their tail latencies. This is depicted in Figure 1a, which shows a two-component application serving a mix of long and short requests. The example shows an arrival order at the first component consisting of a long request with a 10 ms service time, followed by ten short requests each with a 1 ms service time. In a two-core system with FCFS scheduling, the ten short requests will finish before the long request, resulting in a different arrival order at the second component as shown. The reordering can potentially increase the queuing time seen by the long request at the second component, impacting its end-to-end response time. Importantly, such reordering occurs even when the scheduler uses time slicing, as is the case in modern operating systems. For example, if the system in Figure 1a uses round-robin time slicing with 1 ms time slices, the arrival order at the second component will be similar to the FCFS case.

Second, OS scheduler non-determinism can also introduce reordering effects. As another example, if requests have equal processing demands and arrive in a deterministic fashion, OS scheduler non-determinism can introduce small reordering effects at a downstream component. This is depicted in Figure 1b where two requests with identical 10 ms service times arrive at the same instant and are serviced using the two cores using 1 ms time slices. However, scheduler non-determinism, due to factors such as the need to process OS interrupts caused by I/O, network and memory operations, cache misses, and system calls can cause the requests to arrive out of order at the next component. Our empirical results discussed next reveal that request reordering arises even with simple deterministic workloads, and is exacerbated in more complex distributions prevalent in real-world applications.

To demonstrate request reordering in practice, we conducted an experiment with three microservices arranged in a sequential chain. In this experiment, all requests arrive at the first microservice and are processed by each of the three components before departing from the third microservice. We assume that each service runs on a separate 3-core server and that requests are processed in First-Come First-Served (FCFS) order by executing each request on the next available core on each machine. Figure 2 then shows the percentage of requests that are reordered, i.e., requests with completion orders that deviate from the ideal completion order where requests can execute upon arrival without queueing delay, as well as the 99^{th} percentile of request slowdown, defined as the ratio of a request's end-to-end latency, including the processing delay on each microservice and network delay between microservices, to its total service time among all tiers.



(a)

(b)

Figure 2: (a) Request reordering percentage and (b) P99 slowdown with local-FCFS scheduling. All service time distributions have a mean of 5 ms.

In Figure 2, requests are processed following three different service time distributions: fixed, exponential, and bimodal, all with a mean of 5 ms. Even when all requests impose identical processing demands of 5 ms at each tier, FCFS processing results in 40% reordering at a rate of 300 requests per second. This is primarily due to scheduler non-determinism from processing background kernel tasks (e.g., network and memory operations) on each machine. Request reordering worsens when requests at each tier are of unequal lengths, as shown by the exponential and bimodal distributions, where the exponential distribution has a mean of 5 ms, and the bimodal distribution has 99% of requests with exponential service times of 4 ms and the remaining 1% have a mean of 100 ms. As shown, the percentage of request reordering increases compared to fixed-length requests, and so does the P99 slowdown for all requests due to request reordering. Although not shown here, similar reordering effects were seen in processor sharing (i.e., time-sliced) systems.

2.4 Queuing Theory Foundations

The design of TailClipper is inspired by theoretical results from queueing theory. Queueing theory research has mathematically analyzed systems where requests undergo processing by multiple components using a network of queues approach [5]. While many closed-form results exist for mean response times seen in these systems under different workload distributions, tail response times remain more challenging to analyze. An early result from the early 1990s [43] showed that a global FCFS policy, where each component schedules requests in FCFS order based on their system-wide arrival time (rather than arrivals at the current component), minimizes variance in response time over all other policies.

Since minimizing variance in response time also reduces tail latency, this result inspires the use of global arrival timestamps and the oldest request first scheduler presented in Section 3. We note, however, that this classical querying theory result only holds for the case that non-preemptive schedulers are used at each component. When workloads are heavy-tailed, further improvements can be obtained through time slicing, a class of preemptive policy. Intuitively, FCFS-based schedulers can increase waiting times in the presence of heavy-tailed requests, since executing long requests causes short requests to wait, increasing their wait times. For long, OS schedulers have used time slicing, also known as processor sharing, to avoid such issues. Recent queueing theory results confirm this practice and show that processor sharing is an optimal approach for heavy-tailed workloads [41], with the caveat that the theoretical result was shown for a single queue (a single component) and has not been generalized to a network of queues scenario. Nevertheless, our empirical results in Section 5.1 show the benefits of processor sharing for handling heavy-tailed requests in distributed applications with multiple components. Since real-world systems can experience a range of workloads—light, medium, and heavy-tailed—these theory results inspire our overall systems approach of combining global FCFS with a form of processor sharing to achieve good tail latency behavior under a range of workload scenarios.

3 TAILCLIPPER DESIGN

This section presents the design of our TailClipper system. TailClipper's design consists of two key components. First, it provides visibility into how much time requests have spent in the system since their arrival. This is done by attaching an arrival timestamp to each request and propagating this timestamp across components traversed by the request. Second, TailClipper employs a queueing theory-inspired Oldest Request First (ORF) scheduler that combines arrival-based FCFS with limited processor sharing to reduce end-to-end tail latencies. We describe our design of these TailClipper components below.

3.1 Global Arrival Time Timestamping

To provide visibility into request arrival times, TailClipper timestamps each incoming request with a global arrival time (GAT) tag. These timestamps are propagated horizontally and vertically as follows. First, distributed applications in TailClipper are assumed to consist of numerous microservices, and overall request processing involves each component along the call chain making (one or more) independent requests to the next downstream component [29]. Hence, the arrival timestamp is propagated to all downstream components during request processing. Second, each microservice is assumed to dispatch requests to threads in its own thread pool, which are then scheduled by the OS CPU scheduler. The arrival timestamp information should also be propagated down to the OS thread servicing each request in order to enforce the ORF policy.

Suppose that g denotes the time at which a request enters the system (global time, GAT). The local arrival time (LAT) at component i, denoted by l_i, is the time at which the request arrives at that component. The age of the request – the total time spent by the request since its arrival – is $(t - g)$, where t is the current time. Thus, when the request arrives at component i, its age is $(l_i - g)$. Older requests are those with greater age values of $(t - g)$ and need to be prioritized over newer ones. Finally, if d is the departure time of the request – the time when it completes execution – the end-to-end response time is $(d - g)$.

TailClipper attaches the GAT tag g to the request at the system entry point (e.g., the load balancer or the frontend microservice). This is done by including g in the request headers and is transparent to the application. TailClipper is designed to support many request types such as HTTP and RPCs. The GAT tag is propagated along the call chain sequence by copying it from the current local request to the next one as it completes processing at a component. Importantly, TailClipper makes the GAT tag visible to the thread servicing a request, which enables the underlying CPU scheduler to employ these tags when making scheduling decisions. Once a request completes execution through the call chain, a response is sent to the client after removing the GAT tag from the reply.

Conceptually, TailClipper divides the end-to-end system into two domains – a GAT domain and a non-GAT domain. Horizontally, the non-GAT domain consists of all nodes (clients, switches) that handle the request until it reaches the server running the load balancer or the first component where the GAT is generated. Nodes beyond this point are part of the GAT domain. On each server node, only threads that have access to GAT tags are part of the GAT domain. Other applications, background processes, and kernel threads are part of the non-GAT domains and are handled by the default OS scheduler as usual. We next describe TailClipper's ORF scheduler designed for scheduling threads using GAT tags.

3.2 Oldest Request First (ORF) Scheduling

TailClipper's Oldest Request First (ORF) scheduler is designed to reduce tail latencies experienced by requests in the system across its entire call chain. The novelty of its ORF scheduler lies in three aspects. First, ORF is designed to minimize end-to-end tail latency rather than latency at the local component level. Second, ORF combines global FCFS with a

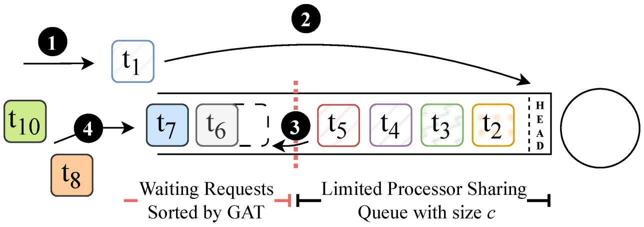

1
2
3
4
t1
t10
t8
t7
t6
t5
t4
t3
t2
HEAD
Waiting Requests Sorted by GAT
Limited Processor Sharing Queue with size c

Figure 3: TailClipper Design – Despite arriving later than all other requests ❶, TailClipper prioritizes t_1 by placing it at the head of the queue due to its oldest GAT ❷, resulting in the rearrangement of the previously c-th oldest requests t_5 ❸ and in the limited processor sharing ($c = 4$). Younger requests are subsequently sorted at the end of the queue ❹.

limited processor sharing (LPS) policy to provide robust latency performance across a wide range of workloads. Third, ORF uses a tunable parameter that enables performance fine-tuning in different settings.

At its core, ORF is a global FCFS scheduler (gFCFS) that prioritizes requests based on their age (i.e., lower GATs). TailClipper requires two input parameters: 1. the GAT of each request, and 2. a configurable parameter c, which controls the concurrency degree. TailClipper propagates a request's GAT tag to the thread servicing that request, ensuring that threads are added to the scheduler run queue in GAT order. This approach allows the OS scheduler of each component to service requests in their arrival order to the system, effectively implementing global FCFS. This is depicted in Figure 3, showing threads in the run queue ordered by their age (i.e., GAT tags). However, as is well known, any FCFS scheduler is vulnerable to starvation issues since a thread serving long requests can delay or starve other queued threads [13]. Consequently, ORF combines global FCFS with time slicing, where the first c threads in the queue are serviced using round-robin processor sharing. We refer to this approach as *limited processor sharing* (LPS) and, as noted in §2.4, such a policy works well for heavy-tailed workloads with a mix of long and short requests [34]. In TailClipper's LPS, c is a configurable system parameter that provides a balance between gFCFS and processor sharing. If c is set to 1, ORF degenerates to pure gFCFS which is optimal for light-tailed workloads, while $c = \infty$ turns ORF into a pure PS (round-robin) scheduler. Typically, TailClipper uses small values of c (e.g., $c = 5$) to prevent starvation while ensuring priority for older requests at the head of the run queue. In Section 5.5, we will evaluate how different settings of c affect the system performance. This hybrid gFCFS-LPS policy provides a good balance between reducing tail latencies and avoiding starvation [13, 34, 43].

4 TAILCLIPPER IMPLEMENTATION

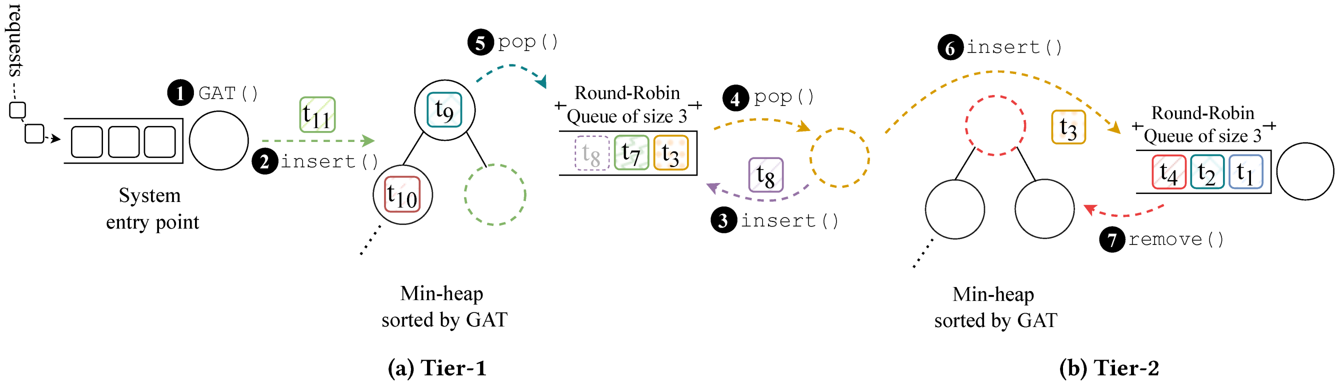
We now discuss the implementation of TailClipper and its scheduling logic, which combines gFCFS and LPS to enforce the ORF policy (§3.2). We have implemented a prototype of TailClipper in C++ using 1200 SLOC, and the source code is publicly available[1].

Scheduling Framework TailClipper's implementation is tightly integrated with the Operating System (OS) scheduler, using thread-level information to make decisions according to the ORF policy. Our TailClipper prototype is implemented using ghOSt, a Linux framework from Google that enables the delegation of kernel scheduling decisions to userspace applications [23]. We utilize ghOSt due to its comprehensive and lightweight API that provides efficient mechanisms to directly adjust and control the kernel's scheduling policy. In addition, ghOSt ensures a fair comparison of different policies, including baselines, since all strategies are subject to the same environmental overheads, e.g., context switch, kernel optimizations, etc. Applications process requests using ghOSt threads, a wrapper around native Linux threads. The scheduling of ghOSt threads is controlled by a user-defined scheduler, i.e., TailClipper.

TailClipper Applications To implement ORF scheduling, TailClipper requires visibility into the system arrival times of requests. The system entry point (e.g., load balancer) encodes the request arrival time into the request header (e.g., HTTP header), and this information is propagated along the microservice call chain. If multiple entry points exist, they need to be time-synchronized to maintain a consistent global arrival order. Another option is to use a logical clock to maintain the request arrival order across multiple entry points. Importantly, other than the entry points, servers hosting microservices do not need to be time-synchronized, as the system arrival times in the request headers are sufficient to locally order requests in system arrival order.

GAT Tags When a request arrives at a microservice, TailClipper extracts the GAT from its header and processes it using a ghOSt thread selected from a pool (thread pool model) or created on the fly (thread-per-request model). TailClipper utilizes ghOSt APIs to propagate its GAT to the ORF scheduler through a small memory buffer. After completing local processing, the request may still need to undergo processing at several other tiers to complete execution. To propagate the GAT along the call chain sequence, all child requests to downstream components inherit the GAT of the parent request. The same GAT is encoded in the request header of child requests so that the following tiers can adhere to the ORF policy.

[1] https://github.com/umassos/TailClipper



(a) Tier-1 **(b) Tier-2**

Figure 4: Example of the scheduling logic of a two-tier distributed application using TailClipper. For simplicity, each tier is deployed on a one-core server. TailClipper employs a min-heap and a round-robin queue to collectively implement the ORF policy.

Algorithm 1 TailClipper's ORF Scheduling Logic

```
Require: $Q_{rr}, Q_{heap}$    // Round-robin queue and min-heap
procedure ScheduleRound()
    for $p \in \{CPU_{idle}\}$ do
        $p_i \leftarrow Q_{rr}$.pop()  // Assign threads to idle CPUs
    for Thread $t, p \in \{CPU_{list}\}$ do    // Thread per CPU
        if time($t_i$) $> CPU_{slice}$ then
            // $t_i$ exceeds time slice, reset its time and swap
               with a new thread from the round-robin
               queue
            $t_i \leftarrow$ reset($t_i$)
            $Q_{rr} \leftarrow Q_{rr} \cup t_i$
            $p_i \leftarrow Q_{rr}$.pop()
    if length($Q_{rr}$) $< c$ then
        // Insert oldest thread from heap into round-robin
           queue
        $t \leftarrow Q_{heap}$.pop()
        $Q_{rr} \leftarrow Q_{rr} \cup t$
    else if $length(Q_{rr}) > c$ then
        // Insert $(c+1)$-th oldest thread from the round-
           robin queue back into the heap
        $t \leftarrow Q_{rr}$.pop()
        $Q_{heap} \leftarrow Q_{heap} \cup t$
```

TailClipper ORF Scheduling To implement the Oldest Request First scheduler using ghOSt, TailClipper employs two primary data structures that collectively implement LPS while also minimizing request reordering with global FCFS. These structures consist of a binary min-heap and a round-robin queue for efficient request scheduling. The round-robin queue holds a subset of the c-th oldest requests that share processing time, while the min-heap sorts all other requests based on their GAT. TailClipper dynamically determines which requests from the min-heap enter or exit the round-robin queue, prioritizing older requests to prevent prolonged queuing time while enforcing that only c requests multiplex the processor. Additionally, c can be dynamically adjusted by modifying the size of the round-robin queue, allowing for flexible control over the degree of concurrency. In the following, we outline the corresponding actions performed by the scheduler when the kernel notifies it with a thread update message. Here, references to threads denote ghOSt threads, which are managed by TailClipper or other ghOSt schedulers.

When TailClipper is notified of the creation of a new thread, it decides whether to insert it into the round-robin queue or the min-heap. TailClipper first compares the new thread's GAT with the youngest thread's GAT in the round-robin queue. If the new thread has a younger GAT, TailClipper inserts it into the min-heap since the new thread has a lower priority. Otherwise, TailClipper inserts the new thread into the round-robin queue to prioritize it, as it has one of the c youngest GATs. After adding the new thread to the round-robin queue, the scheduler may need to remove one thread from the CPU-sharing pool to ensure only c threads are allocated CPU time. If the pool size is larger than c, TailClipper checks whether the youngest thread in the pool is in the round-robin queue or currently running. If it is in the queue, the scheduler removes it from the queue and inserts it back into the min-heap. On the other hand, if the thread is running, the scheduler temporarily allows a violation of the c constraint. It waits for the youngest thread to be preempted in the next scheduling round and then inserts it back into the min-heap.

As discussed in previous sections, a running thread may be preempted by threads in a higher-priority scheduling class or during I/O operations. When TailClipper is notified by the kernel about a thread preemption, it resets the time slice of the thread and adds the preempted thread back to the round-robin queue since the request has the c-th oldest GAT. In the case that a thread completes processing, TailClipper pops the next thread in the round-robin queue and assigns it to the available CPU. Subsequently, the scheduler retrieves the request with the oldest GAT from the min-heap and inserts it into the round-robin queue to maintain the c constraint.

Scheduling round Algorithm 1 outlines TailClipper's scheduling logic. In each scheduling round, TailClipper first assigns threads from the round-robin queue to idle CPUs (Lines 2–3). Then, it implements limited processor sharing by checking whether the running threads' time slices exceed the preemption interval (Line 4). If a thread's slice expires, TailClipper preempts it, resets the slice, and moves it to the back of the queue (Lines 5–8). Subsequently, it schedules a thread from the queue to that CPU (Line 9). After evaluating the time slices of all running threads, TailClipper ensures the size of the round-robin queue remains within c. If the pool size exceeds c, it removes the youngest thread from the queue and inserts it into the min-heap (Lines 10-13). Conversely, if the pool size is below c, TailClipper inserts the thread with the smallest GAT from the min-heap into the round-robin queue (Lines 14-17).

Workflow Figure 4 provides a walk-through example of request scheduling in a two-tier application, with the workflow stages denoted by numbered bullet points. For simplicity, suppose each tier has one worker CPU. In tier-1 (Figure 4a), the application first decodes the GAT from the request header, creates a ghOSt thread to process the request, and propagates the GAT of the thread down to TailClipper ❶. Upon receiving the notification of a new thread creation from the kernel, TailClipper compares the new thread's GAT to the largest GAT in the round-robin queue to check if it was reordered. If it has a larger GAT, indicating that it's younger than all threads in the round-robin queue, TailClipper inserts it into the min-heap ❷. Meanwhile, TailClipper performs limited processor sharing by sharing the CPU time slice among threads in the round-robin queue. When a thread's time slice ends, TailClipper preempts it, resets the slice, and moves it to the back of the queue ❸. Then, it assigns the next thread in the queue to the CPU for processing ❹. Once a thread (e.g., t_3) completes its processing at tier-1, the application sends the subsequent request together with its GAT is passed to tier-2. Concurrently, TailClipper removes the thread with the smallest GAT from the min-heap and adds it to the round-robin queue ❺ to maintain a queue size of c. Once the request arrives at tier-2, the scheduler performs a similar procedure to check the new thread's GAT. If the thread has been reordered, TailClipper directly inserts the new thread into the round-robin queue ❻. It then removes the youngest thread from the queue and inserts it into the min-heap if the updated queue size exceeds c ❼.

5 EXPERIMENTAL EVALUATION

This section presents the results of our experimental evaluation. We compare TailClipper to two state-of-the-art scheduling policies – Shinjuku and Linux's Completely Fair Scheduler – using a real application, a production cluster workload, and a parameterized synthetic workload. In all workloads, we use P99 latency as one of the primary metrics to evaluate the performance of these scheduling policies.

5.1 Experimental Setup

Scheduling policies In addition to TailClipper, we employ two scheduling policies provided by ghOSt [20] for our experiments. Specifically, we use ghOSt-provided implementations of Shinjuku and CFS to ensure all scheduling policies share the same system and hardware optimization features provided by the framework (§4). We denote these policies as *Shinjuku-gh* and *CFS-gh* to distinguish them from their native kernel implementations. *Shinjuku-gh* [25] employs a round-robin scheduler with a centralized queue for all worker cores. Both TailClipper and Shinjuku-gh use a centralized model where a dedicated CPU is responsible for communicating with the kernel and making scheduling decisions. *CFS-gh* is based on Linux's default scheduler [37]. It guarantees a minimum CPU time for each request before potential preemption within a predefined interval. Different from TailClipper and Shinjuku-gh, CFS-gh uses a decentralized model in which each CPU is responsible for the scheduling decisions of its own thread pool. For TailClipper and Shinjuku-gh, the scheduler's preemption interval is set to 30 μsec, consistent with previous work [23, 25]. For CFS-gh, we use the default 1 ms minimum run time and 10 ms guarantee interval provided in the original ghOSt implementation. We configure TailClipper to allow 6 requests to share the CPUs (i.e., $c = 6$) by default.

Infrastructure Our setup comprises five machines: one client machine that hosts the workload generator, one server that hosts the distributed system entry point, and three servers each hosting one microservice tier. Each machine has two 16-core Intel Xeon 2.1GHz processors, 64GB of DRAM, and runs Linux kernel version 5.11 with ghOSt kernel (v70) patches applied [20]. We use an open loop workload generator that supports both trace replay as well as synthetic request generation. The entry point maintains an "infinite" admission queue and limits the number of requests within the distributed application to 200 to prevent system saturation [9]. Upon receiving a request, the entry point uses the current timestamp as the request's GAT and encodes this arrival information into the request header. Subsequently, the entry point forwards the request to the first microservice tier. Each microservice tier decodes the GAT from the request header and processes the request using one thread, which is managed by the userspace scheduler designated at runtime. After that, a component of TailClipper on behalf of the application sets the GAT for the corresponding thread in a shared memory region accessible to the kernel to propagate this information down to the scheduler. Upon completion, the microservice forwards the request together with its GAT in the request header to the subsequent microservice tier over TCP. Finally, the request is returned along the original path to the workload generator (client), where its latency is measured. Each microservice is allocated three worker cores by default.

Workloads We use a parameterized synthetic workload, a production cluster workload, and a real application to compare TailClipper against state-of-the-art scheduling policies.

The synthetic workload generates requests that require processing by three microservices sequentially, a typical depth in real-world applications [32]. At the time of request generation, the workload generator sets the request's processing demand (service time) for each tier by sampling from a target distribution. Specifically[2], we use a light-tailed exponential distribution with a mean service time of 10 ms, a heavy-tailed log-normal distribution with a mean of 10 ms and a standard deviation of 100 ms, and a trimodal distribution with three modes (5 ms, 50 ms, and 100 ms) with an overall mean of 10 ms. Each request executes an idle loop to emulate the service time specified in the request.

The production cluster workload consists of the publicly available Alibaba microservices workload trace collected from their production clusters in 2021 [32]. The trace provides detailed runtime metrics of microservice requests, such as the series of microservice calls of a request (i.e., call chain sequence) and the response time of each microservice call. We replay two common call chains by setting the reported response times as request processing demands at the workload generator. Each call chain exhibits distinct service time distributions across the tiers as shown in Figures 7(a) and 7(c). This workload trace allows us to study how each scheduling policy performs under real-world scenarios.

5.2 Real-world Application Performance

To assess the performance of the schedulers in a real-world application, we build an image processing application using the `opencv_dnn` module in OpenCV 4.9 [7]. This application comprises two microservices: (i) an image preprocessing microservice and (ii) an image classification microservice. The image preprocessing microservice invokes the OpenCV `median blur()` [8] to reduce the noise in a given image, using its output as the input of the subsequent tier. Subsequently, the image classification microservice predicts the image class using the pre-trained GoogLeNet network from the Caffe model zoo [24]. Here, tail latencies arise from varying input complexities. For example, the content of the images can affect the processing demand during the median blur operation, leading to variations in preprocessing time. We sequentially processed inputs through the application to measure latency at each stage and show the magnitude of their tail latencies in Figure 5.

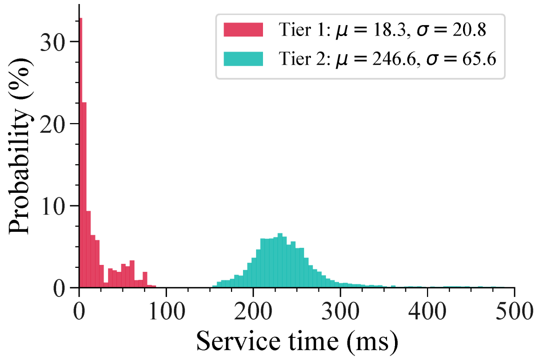
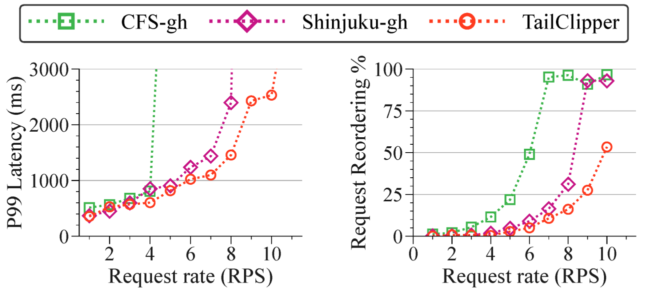


Figure 5: Service time distributions of the image preprocessing microservice and the image classification microservice.

Each client transmits a randomly selected image of 1 MB from a Flicker dataset [35] to the application entry point via TCP. The entry point timestamps the request with the local time to generate the GAT, encodes it into the request header, and forwards the request to the first tier. Upon receiving a request, the first tier decodes the request's GAT, spawns a thread to preprocess the image, and assigns the thread's GAT accordingly. The scheduler of the microservice then schedules the threads based on its scheduling logic (e.g., TailClipper, CFS-gh, etc.). Once processing is completed at the first tier, the intermediate result, along with the request's GAT, is transmitted to the second tier, where image classification takes place in a similar workflow. Finally, the predicted image class is returned to the client along the original path.

Figure 6 compares Shinjuku-gh, CFS-gh, and TailClipper under varying image processing request loads. Figure 6a shows the P99 latency using the three scheduling policies as a function of requests per second (RPS). Due to the high variance in processing demand at the first stage, requests

[2] Light-tailed distributions exhibit probabilities with tails that decay at an exponential rate, while heavy-tailed distributions are not bound by exponential decay.



(a) (b)

Figure 6: Comparing scheduling policies with (a) P99 latency and (b) request reordering % using image processing pipeline workload.

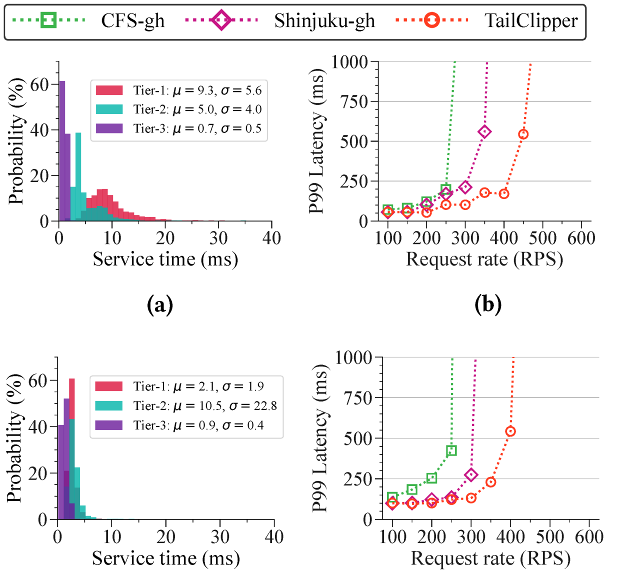
may arrive out of order at the second stage, leading to prolonged queueing delays. By effectively minimizing request reordering across all RPS values as shown in Figure 6b, TailClipper consistently outperforms CFS-gh and Shinjuku-gh under moderate and high loads. Specifically, under moderate load (RPS = 4), TailClipper reduces the P99 latency by up to 29% compared to Shinjuku-gh and CFS-gh. When the system is near saturation (RPS = 7), TailClipper also outperforms Shinjuku-gh and CFS-gh, achieving up to 24% lower P99 latency.

> **Key Takeaway** *When subject to real workloads, TailClipper outperforms state-of-the-art schedulers under moderate and high loads by effectively accounting for request reordering effects.*

5.3 Comparing TailClipper with Baselines

We then study the performance of the three scheduling policies under an existing cluster workload by replaying the Alibaba microservice traces. Figure 7 shows the service time distributions of the call chains and compares the P99 latency of all policies.

Figure 7a presents the service time distribution of each tier in the first call chain, where the first tier (Tier-1) approximately follows a normal distribution with a mean of 9 ms, the second tier (Tier-2) has a mean of 5 ms and a standard deviation of 4 ms, and the third tier's (Tier-3) service times lie within 0.5 to 1 ms. For this call chain sequence, request reordering occurs in both Tier-1 and Tier-2 as the processing demands in both tiers vary among requests, potentially resulting in prolonged queueing delays for older requests in Tier-2 and Tier-3. Figure 7b shows that as TailClipper prioritizes older requests by employing the ORF policy, it yields a 52% reduction in the P99 latency when compared to Shinjuku-gh under a moderate load of 300 RPS.



(a) (b)

(c) (d)

Figure 7: Comparing scheduling policies with P99 latency (right) using traces of Alibaba microservice call chains with different service time distributions (left): (a)-(b): Call chain 1; (c)-(d): Call chain 2;

Figures 7c and 7d present the second call chain, characterized by low processing demands for Tier-1 and Tier-3, while Tier-2 follows a bimodal distribution with the two modes peaking at 4 ms and 35 ms. Although request reordering may occur in Tier-2, TailClipper demonstrates a smaller performance improvement over Shinjuku-gh compared to the previous call chain. This is due to the small variability in processing demands at Tier-1, resulting in unnoticeable delays observed by reordered requests at Tier-2. In addition, processing demands in Tier-3 are low, which prevents any queue buildup. While older requests may undergo reordering after being processed by Tier-2, the impact on queuing delay in Tier-3 remains minimal. Consequently, latency reduction due to prioritizing old requests has little impact on overall latency, and TailClipper only brings a 9% improvement under low load (RPS = 200) compared to Shinjuku-gh.

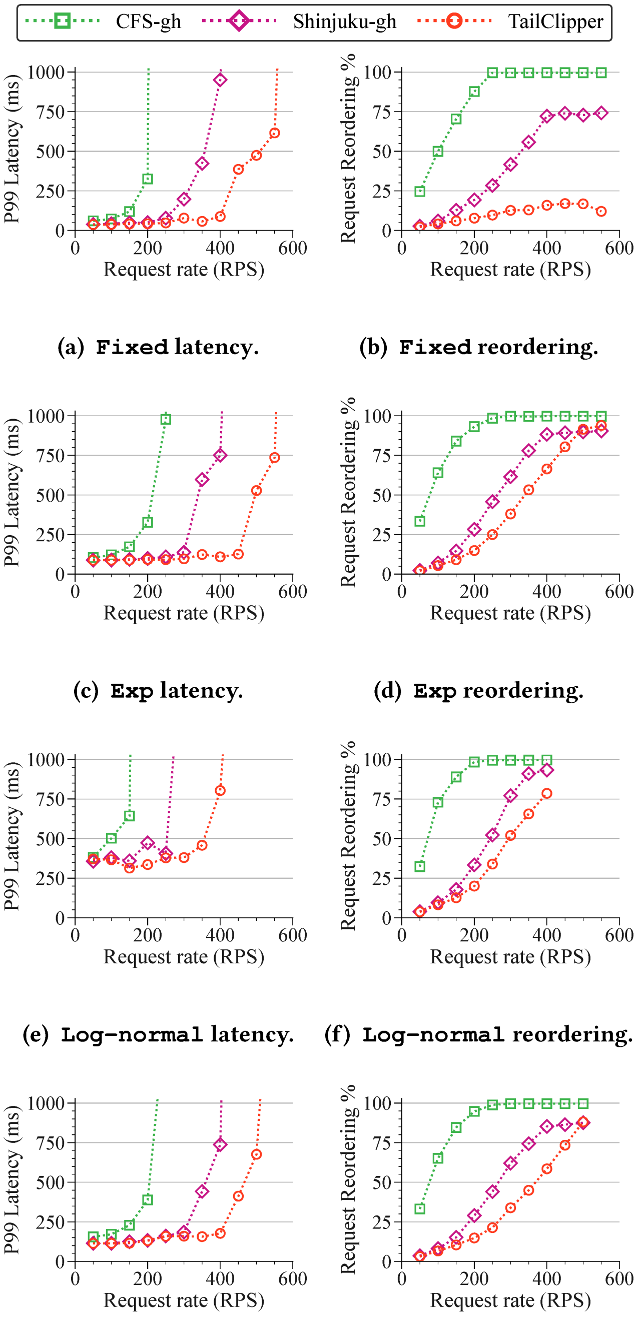
> **Key Takeaway** *The higher the amount of reordering, the more pronounced the effects on the tail latency. Under a cluster workload, TailClipper demonstrates superior performance compared to Shinjuku-gh and CFS-gh by minimizing request reorderings across application components.*

5.4 Impact of Light and Heavy-tailed Workloads

Next, we study the performance of schedulers using synthetic workloads with a more diverse and controlled set of scenarios. Figure 8 compares TailClipper against Shinjuku-gh and CFS-gh using four service time distributions – fixed, light-tailed, heavy-tailed, and trimodal workloads. To assess how effectively each scheduling policy prioritizes older requests to mitigate tail latency, we report the request reordering percentage observed in these experiments. This metric represents the fraction of requests whose completion order deviates from an "ideal" completion order, which assumes unlimited resources across all tiers such that requests would immediately execute upon arrival without any queuing delay. Figure 8 presents the P99 latency across different request rates for each workload in the left column and their corresponding request reordering percentages in the right column.

Fixed workload Figure 8a shows latency results obtained for the fixed distribution when requests execute for 10 ms at each tier. Importantly, Figure 8b illustrates that, although service times remain constant and theoretically should not cause request reordering, non-deterministic factors in the operating system, such as background tasks, preempt threads executing requests, can lead to request reordering. As a result, both Shinjuku-gh and CFS-gh exhibit high request reordering percentages, and the P99 latency exceeds 1s when the request rate exceeds 450 RPS. On the other hand, despite such non-determinism, TailClipper demonstrates a more stable control of the tail compared to Shinjuku-gh and CFS-gh even when the system is nearing saturation, surpassing the one-second mark only when the RPS exceeds 600. The improved performance is attributed to TailClipper's effective mitigation of request reordering percentage, which only reaches up to 25% across all request rates.

Light- and Heavy-tailed workloads Figures 8c and 8d report results for a light-tailed exponential distribution workload, which incurs additional request reordering compared to the fixed service time workload. At moderate load (RPS = 400), TailClipper improves upon Shinjuku-gh by up to 81% in P99 latency. By suppressing request reordering (Figure 8d), TailClipper can also achieve the lowest P99 latency under high loads (RPS > 400). Figure 8e shows the tail latencies for a heavy-tailed log-normal distribution workload, where requests with greater processing demands are more common. Consequently, when older requests are reordered, their queuing delays might be prolonged because preceding requests could be longer. In such scenarios, the performance improvements from using TailClipper intensify as it prioritizes older requests. As a result, TailClipper surpasses the



(a) Fixed latency. (b) Fixed reordering.

(c) Exp latency. (d) Exp reordering.

(e) Log-normal latency. (f) Log-normal reordering.

(g) Trimodal latency. (h) Trimodal reordering.

Figure 8: Comparing scheduling policies with P99 latency (left) and request reordering % (right) using synthetic workloads with different service time distributions: (a)-(b), Fixed(10); (c)-(d): Exp(10); (e)-(f): Log-normal(10,100); (g)-(h): Trimodal(90-5,9-50,1-100).

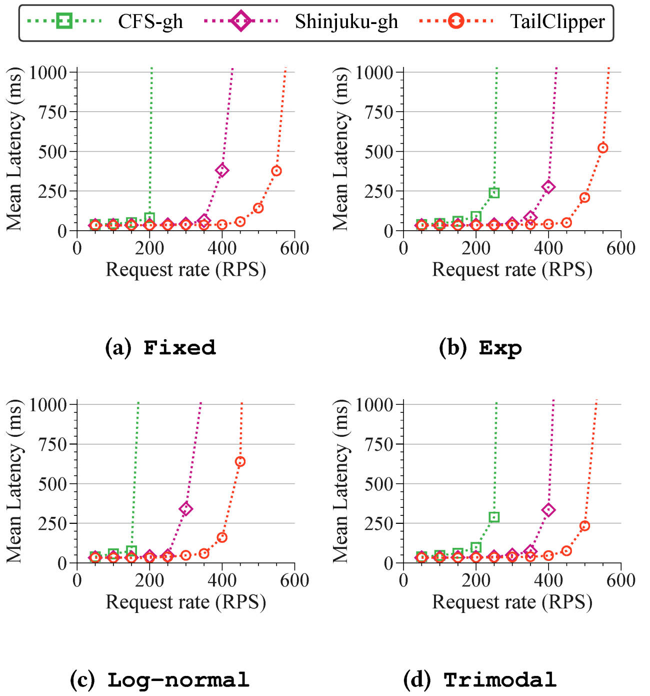


Figure 9: Mean latencies under different service time distributions: (a) `Fixed(10)`, (b) `Exp(10)`, (c) `Log-normal(10,100)`, and (d) `Trimodal(90-5,9-50,1-100)`.

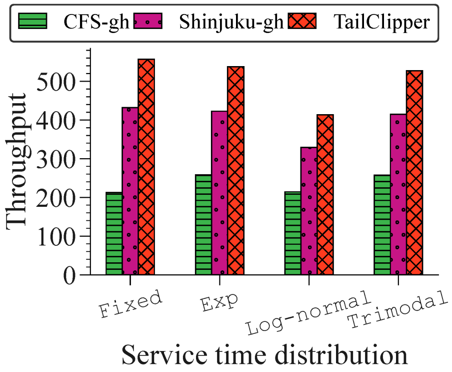


Figure 10: Systems throughputs under the corresponding distributions.

1s tail latency target only when the request rate exceeds 400 RPS, sustaining 1.6× more load than that of Shinjuku-gh.

Trimodal workload In a more complex setting, Figures 8g and 8h show results for a trimodal distribution workload consisting of requests with three possible processing demand distributions at each tier. Such a workload mimics scenarios in which the microservice threads have to perform different functionalities, such as local cache lookup, data transformation, and I/O operations. With a more distinct set of request lengths, where 10% of the requests have medium (50 ms) and high (100 ms) processing demands, reordered requests are more likely to experience long queueing delays (Figure 8h). By prioritizing older requests while preventing starvation through LPS, Figure 8g shows TailClipper reduces the P99 latency by 76% compared to Shinjuku-gh under a 1s target.

Mean latency and throughput Figure 9 reports the mean latency of the three scheduling policies under a workload with exponentially distributed service time. Overall, we observe that TailClipper exhibits the lowest mean latency under high loads (RPS > 350), whereas CFS-gh exhibits the highest mean latency. Specifically, for the `Exp` workload shown in Figure 9b, at RPS = 250, TailClipper can achieve up to 11% lower mean latency than Shinjuku-gh. At RPS = 350, TailClipper reduces mean latency by up to 54% compared to Shinjuku-gh. Figure 10 compares the throughput achieved under each service time distribution with a request rate of 600 RPS. We observe that TailClipper exhibits the highest throughput for all workloads tested, 29% more compared to Shinjuku-gh and up to 2.2× more compared to CFS-gh.

The improvements in mean latency and throughput under high loads can be attributed to TailClipper's limited CPU sharing strategy, which contrasts with the unlimited sharing strategy in Shinjuku-gh. In TailClipper, only a limited number of the oldest requests share CPU time at any given moment, while in Shinjuku-gh, all requests share CPU time uniformly. As a result, Shinjuku-gh is more susceptible to performance degradation, as a higher number of threads sharing the CPU can lead to more context switches, particularly under high job loads. Conversely, TailClipper's approach of limiting the number of requests sharing the CPUs helps mitigate this overhead, resulting in lower mean latency and higher throughput under high loads.

Long request-dominated workload Figure 11 examines how scheduling policies perform under a long request-dominated workload. We use a bimodal workload where 99% of requests have a service time of 100 ms, while the remaining 1% have a service time of 10 ms. Figure 11a shows that TailClipper can achieve up to a 17% decrease in tail latency compared to Shinjuku-gh under a moderate load (RPS = 20). One potential drawback of employing TailClipper is that when TailClipper prioritizes a subset of the long requests, short requests can be temporarily delayed behind long-running requests, which results in starvation. Figure 11b focuses on requests with the least 1% of total service time (i.e., short requests) to study this effect. We find that although TailClipper increases P99 latency for short requests under moderate loads (RPS < 36), this does not affect overall P99 latency, which is predominantly influenced by the long

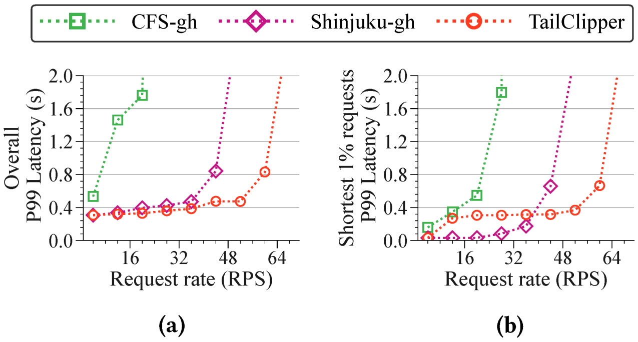


Figure 11: Comparing scheduling policies with (a) overall P99 latency and (b) the P99 latency of requests with the smallest 1% total service time under a long requests-dominated workload– `Bimodal`(99-100,1-10).

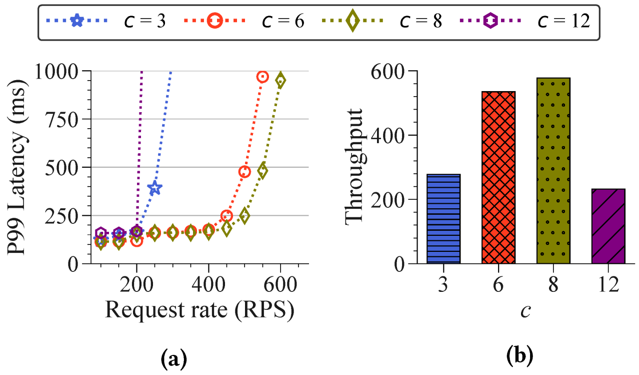


Figure 12: Comparing variations of TailClipper with (a) P99 latency and (b) throughput under a multimodal workload– `Trimodal`(90-5,10-50,1-100).

requests dominating the workload. Under high loads (RPS > 44), TailClipper outperforms Shinjuku-gh for both overall and short request-only P99 latency.

Key Takeaway *Request reordering occurs even under simple scenarios with constant loads. TailClipper consistently reduces the P99 latency over state-of-the-art scheduling policies across a wide range of workloads.*

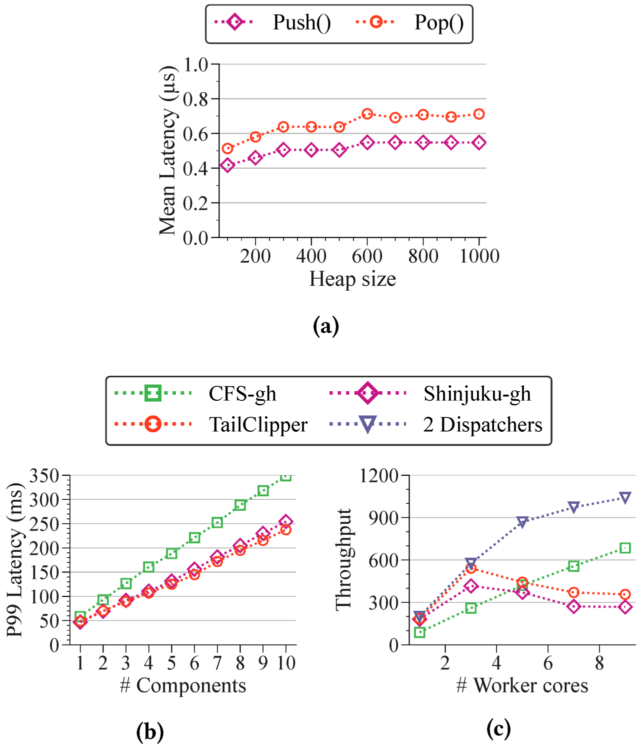
5.5 Performance and Sensitivity Analysis

Effect of limited processing sharing Figure 12 varies the limited processing sharing size c for a `Trimodal`(90-5,10-50,1-100) workload. In Figure 12a, we observe that with three different request lengths, setting c to a value between the number of cores and twice that number (4 to 8) can be a rough rule of thumb, where $c = 8$ results in the largest throughput (Figure 12b). With a small value of c, TailClipper essentially operates as a global-FCFS scheduler, which can potentially lead to request starvation and increase the tail latency. Conversely, a large value of c shifts TailClipper toward a PS scheduler, where more requests share CPU time, leading to prolonged queuing delays as older requests are less prioritized. Exploring the optimal selection of c for different workloads and hardware configurations is the subject of future work.

Effect of min-heap size Figure 13a evaluates the scheduling overhead associated with TailClipper. Specifically, we analyze two key operations in TailClipper's ORF scheduling: `Push()` and `Pop()` of its min-heap. We initialize the min-heap with varying sizes and study how its size impacts the mean latency of each operation. For `Push()`, a random item is inserted into the min-heap. Our measurements indicate that for both operations, the overhead of `Push()` and `Pop()` range from 0.4 to 0.6 microseconds and are minimal compared to the workload service time. Furthermore, the overheads only increase marginally with the min-heap size even for large heaps of up to 1000 requests. It is also important to note that a native kernel implementation could further minimize the overhead of the scheduling operations, as will be discussed in Section 6.

Effect of number of tiers Figure 13b shows the performance of the scheduling policies as the number of tiers varies using an `Exp`(10) workload with low load (RPS = 100). The result indicates that for all scheduling policies, the tail latency of requests scales linearly with the increasing number of tiers as the total service times of requests increase. Note that the performance margin between TailClipper and the other two scheduling policies increases with an increasing number of tiers. This phenomenon can be attributed to the higher probability of request reordering due to the presence of more tiers and the variability in service times at each tier. As a result, TailClipper outperforms the other policies by leveraging global arrival times, leading to improved performance.

Effect of number of cores Figure 13c assesses the scalability of the scheduling policies in relation to the number of worker CPUs. We observe that TailClipper and Shinjuku-gh achieve linear scalability up to 3 cores, while CFS-gh scales linearly. This is because for centralized schedulers, the performance on additional cores is limited by the message communication bottleneck between the dispatcher CPU and the worker CPUs. We note that this messaging bottleneck largely stems from our choice of the userspace ghOSt scheduling framework to implement TailClipper and can be alleviated in two ways. First, the issue can be resolved by incorporating additional dispatcher CPUs as shown in Figure 13c and also demonstrated in [25]. Second, any userspace scheduler generally incurs greater scheduling overhead than kernel schedulers, and a native in-kernel implementation



(a)

(b)

(c)

Figure 13: (a) Overhead measurements of TailClipper's two key scheduling operations. (b) Comparing scheduling policies with varying microservices tiers under a request rate of 100 RPS using `Exp`(10) workload. (c) Scheduling policies' throughput as we scale the number of worker cores.

of TailClipper can improve scalability by using kernel data structures and avoiding explicit message passing.

> **Key Takeaway** *Applications using TailClipper can scale seamlessly with additional microservice tiers, higher loads, and increased worker cores. In addition, TailClipper's configurable parameter enables performance fine-tuning in different workloads.*

6 DISCUSSION

Threading model TailClipper does not make assumptions about the threading models used by applications. Beyond the worker thread pool and thread-per-request models previously mentioned, TailClipper also supports applications using other threading models, such as coroutines, where a single thread handles multiple requests. Coroutines share the same thread asynchronously, aiming to improve concurrency throughput, especially with blocking I/O code. In such cases, when context-switching from one coroutine to another, the TailClipper application's scheduler would call the ghOSt API to update the ghOSt thread's GAT to match the GAT of the request currently being processed by the coroutine. It is important to note that different threading models have different target objectives. While thread-per-request offers flexible parallelism and intuitive design patterns, coroutines target computations with decomposable, pipelined tasks that depend on each other. Although they may offer a lightweight mechanism to optimize concurrency, some of these coroutine tasks may block and suspend while waiting, for example due to database or other I/O operations, causing request reordering. Using GAT, TailClipper can mitigate the effects of request reordering regardless of the threading model used.

Native implementation TailClipper is currently implemented as a userspace scheduler, and our evaluation with millisecond-scale workloads demonstrates that the overhead of ghOSt userspace scheduling does not compromise its superior performance. However, in scenarios where the server experiences a high volume of requests operating at a microsecond scale, the frequent communication overhead between the userspace scheduler and the kernel may impact system performance. In such cases, a native in-kernel implementation of TailClipper may be preferred over our proof-of-concept implementation. For example, we can use the Linux external scheduler extension `sched_ext` to implement TailClipper's ORF scheduler in a BPF program that runs in kernel space. In this setup, a TailClipper userspace component extracts the GAT from the request header and propagates it down to the ORF scheduler using BPF maps. These maps function as an efficient shared data structure between TailClipper's userspace component and the kernel-space ORF scheduler. The ORF scheduler can then leverage the GAT information in BPF maps to schedule threads accordingly. This native implementation will eliminate the need for the kernel to notify the userspace scheduler of thread updates, significantly reducing message overhead compared to our proof-of-concept userspace implementation. Furthermore, it decreases the number of system calls required to enforce scheduling decisions.

Application-level cluster schedulers Cluster computing frameworks, such as Spark [51] and Apollo [6], are commonly deployed to optimize resource management and job scheduling in distributed environments. TailClipper is designed as a CPU scheduler and, as such, can interoperate with these higher-level framework schedulers that already work with the existing Linux scheduler. The current implementation of TailClipper uses ghOSt, which requires software to be recompiled to link to the ghOSt library. However, if an in-kernel implementation is used, recompilation is unnecessary. In both cases, TailClipper needs to run at the system entry point for timestamping requests. When a high-level framework scheduler enforces a specific job scheduling order, TailClipper's ability to address reordering is constrained by those higher-level policies. Nonetheless, it can still provide tail latency improvements at the job's request level.

Microservices with KVS and Databases In web applications, microservices often utilize key-value stores (KVS) or database systems for efficient data management. To integrate TailClipper, these microservices can utilize ghOSt threads to handle I/O operations, as demonstrated in [23]. However, reordering may occur if a thread is blocked while waiting for an I/O operation to complete, which can negatively impact overall application performance. In such scenarios, TailClipper can mitigate the effects of such reordering.

7 RELATED WORK

Kernel-bypass techniques have been commonly employed to minimize operating system overheads and address tail latency issues [1, 26, 38–40, 50]. While these approaches have demonstrated effectiveness in reducing tail response times, some studies have highlighted potential negative impacts on collocated applications [36]. Replication techniques [11, 19, 42, 46, 49], novel system softwares [1, 16, 39], and new architectures [10, 22] have also been proposed to reduce tail latencies. In particular, Shinjuku [25] is a single-address space operating system that implements preemptive scheduling at the microsecond scale to minimize tail latencies and increase system throughput. Shenango [36] is a system that reallocates cores across applications at fine time scales. Arachne [40] implements a user-level scheduler of threads with applications determining the appropriate core allocation scheme based on load. RobinHood [3] dynamically reallocates cache resources to meet applications' needs. Brownout [14] aims to minimize the tail by adjusting the amount of work done by a request. Although these approaches have demonstrated effectiveness in reducing tail latency, they are primarily designed for single systems and do not explore the optimization potential for distributed request processing. TailClipper differs from the existing work in that it employs a holistic approach that schedules requests across application components, leveraging global arrival information to mitigate the effects of distributed processing.

In the literature, various percentiles have been used to quantify tail response time reduction, such as P95 [14, 27, 30], P99 [11, 21, 28] or P99.9 [12] percentiles. In contrast, TailClipper does not specifically aim at reducing a particular percentile response time and focuses on optimizing the execution order of requests to minimize tail latency. TailClipper can also seamlessly integrate with solutions addressing other sources of performance variability, such as garbage collection, wake-up from low-energy states, and interference from co-located workloads.

8 CONCLUSIONS

Minimizing tail latency has become a critical priority for enhancing the efficiency of online web services and distributed applications. However, traditional approaches that optimize latency independently at individual components overlook distributed processing effects, such as request reordering, which can lead to increased end-to-end tail latency. To address these challenges, this paper introduced TailClipper, a novel distributed scheduler designed to optimize request scheduling globally across the system rather than locally on individual components. TailClipper combines two querying theoretical results to deliver robust latency performance across various workloads. Our evaluations using cloud workload traces and a real-world application revealed that TailClipper can achieve tail latency reductions of up to 81%, while also improving mean latency and throughput under high loads compared to state-of-the-art scheduling policies.

ACKNOWLEDGMENTS

We thank our shepherd Sameh Elnikety and the reviewers for their valuable comments. This research is supported by NSF grants 2211302, 2211888, 2213636, 2105494, 23091241, the Army Research Laboratory under Cooperative Agreement W911NF-17-2-0196 (IoBT CRA), and VMWare. Ali-Eldin is supported by SSF future research leaders grant.

REFERENCES

[1] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2017. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Transactions on Computer Systems (TOCS)* 34, 4 (2017), 11. https://doi.org/10.1145/2997641

[2] Mike Belshe. 2010. More Bandwidth Doesn't Matter (Much). https://bit.ly/3RUykxs.

[3] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. 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)*. 195–212. https://doi.org/10.5555/3291168.3291183

[4] Peter Bodík, Ishai Menache, Joseph (Seffi) Naor, and Jonathan Yaniv. 2014. Brief Announcement: Deadline-Aware Scheduling of Big-Data Processing Jobs. In *SPAA*. https://www.microsoft.com/en-us/research/publication/brief-announcement-deadline-aware-scheduling-of-big-data-processing-jobs/

[5] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. 1998. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Interscience, USA.

[6] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 285–300.

[7] Gary Bradski, Adrian Kaehler, et al. 2000. OpenCV. *Dr. Dobb's journal of software tools* 3, 2 (2000).

[8] Gary Bradski, Adrian Kaehler, et al. 2024. OpenCV - Smoothing Images. https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html.

[9] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. 2021. Metastable Failures in Distributed Systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 221–227. https://doi.org/10.1145/3458336.3465286

[10] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A Cloud-Scale Acceleration Architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 7. https://doi.org/10.1109/MICRO.2016.7783710

[11] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013). https://doi.org/10.1145/2408776.2408794

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007). https://doi.org/10.1145/1323293.1294281

[13] Aldric Degorre and Oded Maler. 2008. On scheduling policies for streams of structured jobs. In *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 141–154.

[14] David Desmeurs, Cristian Klein, Alessandro Vittorio Papadopoulos, and Johan Tordsson. 2015. Event-Driven Application Brownout: Reconciling High Utilization and Low Tail Response Times. In *Cloud and Autonomic Computing (ICCAC)*. https://doi.org/10.1109/ICCAC.2015.25

[15] Ahmed Eleliemy and Florina M Ciorba. 2021. A Resourceful Coordination Approach for Multilevel Scheduling. *arXiv preprint arXiv:2103.05809* (2021).

[16] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[17] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux journal* 2004, 124 (2004), 5.

[18] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 281–297. https://doi.org/10.5555/3488766.3488782

[19] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, Esa Hyytiä, and Alan Scheller-Wolf. 2016. Queueing with Redundant Requests: Exact Analysis. *Queueing Systems* 83, 3 (2016). https://doi.org/10.1007/s11134-016-9485-y

[20] Google. [n. d.]. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. https://github.com/google/ghost-userspace https://github.com/google/ghost-userspace.

[21] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/2694344.2694384

[22] Md E Haque, Yuxiong He, Sameh Elnikety, Thu D Nguyen, Ricardo Bianchini, and Kathryn S McKinley. 2017. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 625–638. https://doi.org/10.1145/3123939.3123956

[23] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 588–604. https://doi.org/10.1145/3477132.3483542

[24] Yangqing Jia and Evan Shelhamer. 2024. Caffe Model Zoo. http://caffe.berkeleyvision.org/model_zoo.

[25] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazieres, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360. https://doi.org/10.5555/3323234.3323264

[26] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. 2012. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 9. https://doi.org/10.1145/2391229.2391238

[27] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 729–742. https://doi.org/10.1145/2644865.2541944

[28] Jinhan Kim, Sameh Elnikety, Yuxiong He, Seung-won Hwang, and Shaolei Ren. 2013. QACO: Exploiting Partial Execution in Web Servers. In *Cloud and Autonomic Computing Conference (CAC)*. ACM, Article 12. https://doi.org/10.1145/2494621.2494636

[29] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. 2013. Online Controlled Experiments at Large Scale. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1168–1176. https://doi.org/10.1145/2487575.2488217

[30] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *European Conference on Computer Systems (EuroSys)*. ACM, Article 4. https://doi.org/10.1145/2592798.2592821

[31] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Symposium on Cloud Computing (SoCC)*. ACM, Article 9. https://doi.org/10.1145/2670979.2670988

[32] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing*. 412–426. https://doi.org/10.1145/3472883.3487003 https://github.com/alibaba/clusterdata.

[33] Marissa Mayer. 2006. What Google Knows. *Proceedings of the Third Annual Web* 2 (2006).

[34] Jayakrishnan Nair, Adam Wierman, and Bert Zwart. 2010. Tail-Robust Scheduling via Limited Processor Sharing. *Performance Evaluation* 67, 11 (2010), 978–995. https://doi.org/10.1016/j.peva.2010.08.012

[35] Samuel S. Ogden, Xiangnan Kong, and Tian Guo. 2021. PieSlicer: Dynamically Improving Response Time for Cloud-based CNN Inference. In *12th ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery (ACM).

[36] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378. https://doi.org/10.5555/3323234.3323265

[37] Chandandeep Pabla. 2009. Completely Fair Scheduler. *Linux Magazine* 184 (2009).

[38] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2016. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)* 33, 4 (2016), 11. https://doi.org/doi/10.1145/2812806

[39] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 325–341. https://doi.org/10.1145/3132747.3132780

[40] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 145–160. https://doi.org/10.5555/3291168.3291180

[41] Ziv Scully, Lucas van Kreveld, Onno Boxma, Jan-Pieter Dorsman, and Adam Wierman. 2020. Characterizing Policies with Optimal Response Time Tails under Heavy-Tailed Job Sizes. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems* (Boston, MA, USA) *(SIGMETRICS '20)*. Association for Computing Machinery, New York, NY, USA, 35–36. https://doi.org/10.1145/3393691.3394179

[42] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 513–527. https://doi.org/10.5555/2789770.2789806

[43] Don Towsley and François Baccelli. 1991. Comparisons of Service Disciplines in a Tandem Queueing Network with Real Time Constraints. *Operations Research Letters* 10, 1 (1991), 49–55. https://doi.org/10.1016/0167-6377(91)90086-5

[44] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, and Pawan Goyal. 2005. Dynamic provisioning of multi-tier internet applications. In *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE, 217–228.

[45] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and TN Vijaykumar. 2015. Timetrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 585–597. https://doi.org/10.1145/2830772.2830779

[46] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. 2013. Low Latency via Redundancy. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 283–294. https://doi.org/10.1145/2535372.2535392

[47] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Chien-An Lai, Chien-An Cho, Yuji Nomura, and Calton Pu. 2014. Lightning in the Cloud: A Study of Very Short Bottlenecks on n-Tier Web Application Performance. In *Proceedings of USENIX Conference on Timely Results in Operating Systems*. https://doi.org/10.13140/2.1.1479.0402

[48] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/2018436.2018443

[49] Zhe Wu, Curtis Yu, and Harsha V Madhyastha. 2015. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 543–557. https://doi.org/10.5555/2789770.2789808

[50] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 43–56.

[51] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.

[52] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, 655–672. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou