A Tail Latency SLO Guaranteed Task Scheduling Scheme for User-Facing Services

Zhijun Wang, Huiyang Li, Lin Sun, Stoddard Rosenkrantz, Hao Che, and Hong Jiang, *Fellow, IEEE*

***Abstract*—A primary design objective for user-facing services for cloud and edge computing is to maximize query throughput, while meeting query tail latency Service Level Objectives (SLOs) for individual queries. Unfortunately, the existing solutions fall short of achieving this design objective, which we argue, is largely attributed to the fact that they fail to take the query fanout explicitly into account. In this paper, we propose TailGuard based on a Tail-latency-SLO-and-Fanout-aware Earliest-Deadline-First Queuing policy (TF-EDFQ) for task queuing at individual task servers the query tasks are fanned out to. With the task pre-dequeuing time deadline for each task being derived based on both query tail latency SLO and query fanout, TailGuard takes an important first step towards achieving the design objective. A query admission control scheme is also developed to provide tail latency SLO guarantee in the presence of resource shortages. TailGuard is evaluated against First-In-First-Out (FIFO) task queuing, task PRIority Queuing (PRIQ) and Tail-latency-SLO-aware EDFQ (T-EDFQ) policies by both simulation and testing in the Amazon EC2 cloud. It is driven by three types of applications in the Tailbench benchmark suite, featuring web search, in-memory key-value store, and transactional database applications. The results demonstrate that TailGuard can significantly improve resource utilization (e.g., up to 80% compared to FIFO), while also meeting the targeted tail latency SLOs, as compared with the other three policies. TailGuard is also implemented and tested in a highly heterogeneous Sensing-as-a-Service (SaS) testbed for a data sensing service, demonstrating performance gains of up to 33% . These results are consistent with both the simulation and Amazon EC2 results.**

***Index Terms*—Task scheduling, resource management, tail latency SLO, user-facing service.**

I. Introduction

IT has been widely recognized that the query tail latency for user-facing services, such as web searching and emergency response through edge-based crowdsensing, has a great impact on user experience and hence, business revenues. For example, for Amazon online web services, every 100-millisecond addition of query tail latency causes 1% decrease in sale [1]. To meet strict tail latency Service Level Objectives (SLOs), the resources for user-facing services are generally over-provisioned [2], at the cost of reduced profit. As a result, a key design objective of a user-facing service, called **the design objective** in short hereafter, *is to maximize the resource utilization or query throughput, while meeting tail latency SLOs for individual queries*.

However, achieving the above design objective is by no means easy. A query for a typical user-facing service may spawn a number of tasks, known as **query fanout**, to be dispatched to, queued and serviced in parallel in different servers or edge nodes where the data shards reside and the slowest task of the query determines the query response time [3], [4]. The range of query fanouts may differ from one service to another, e.g., up to several hundreds for online social networking [5], on the order of several thousands to tens of thousands for web search [3], and potentially up to millions for emergency response through edge crowdsensing [6]. A small number of outliers (caused by, e.g., skewed workloads [7] or software/hardware resource variations [8]) can significantly impact the query tail latency performance [3]. While a large body of works have been devoted to alleviating the impact of outliers on the query tail latency performance (e.g., [9], [11], [12], [13], [14], [15], [16], [17], [18]), to the best of our knowledge, no existing solution attempts to meet more than one query tail latency SLO to satisfy different performance requirements of individual users, while maximizing the resource utilization or query throughput, hence falling short of the design objective.

In this paper, we claim that a solution that stands a chance to achieve the design objective must be not only tail latency SLO aware but also query fanout aware. This is simply because *to meet a given tail latency SLO, the task resource demands for tasks belonging to queries with different fanouts are different*. For example, assume that with a given amount of resource allocated to process each task and the task response time for each task has 1% probability to be over 100 ms. Then the query response time for a query with fanout k_f has probability, 1-0.99^{k_f}, to be over 100 ms, meaning that a query with $k_f = 1$ and $k_f = 100$ have 1% and 63.4% probabilities of being over 100 ms, respectively. This implies that while a query with $k_f = 1$ can meet the tail latency SLO in terms of the 99th percentile tail latency of 100 ms, a query with $k_f = 100$ cannot. In order to allow the query with $k_f = 100$ to also meet the same tail latency SLO, a task associated with the query must be allocated a much larger amount of resource so that the chance it will exceed 100 ms is as small as 0.01% . This ensures that the probably that the query response time exceeds 100 ms is 1-0.9999^{100} = 0.01 or 1% , i.e., meeting the same tail latency SLO as the query with $k_f = 1$. This example clearly demonstrates that to meet a query

Received 8 November 2023; revised 23 January 2025; accepted 10 February 2025. Date of publication 14 February 2025; date of current version 5 March 2025. This work was supported US NSF under Grant CCF SHF-2008835 and Grant CCF-SHF2226117. Recommended for acceptance by E. Smirni. *(Corresponding author: Zhijun Wang.)*

The authors are with the Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019 USA (e-mail: zhijun.wang@uta.edu; huiyang.li@mavs.uta.edu; lxs5171@mavs.uta.edu; todd.rosenkrantz@mavs.uta.edu; hche@cse.uta.edu; hong.jiang@uta.edu).

Digital Object Identifier 10.1109/TPDS.2025.3542638

tail latency SLO for all queries regardless query fanouts, the task resource demands for tasks belonging to queries with different fanouts are different and a task belonging to a query with a larger fanout demands more resources, confirming our claim. Although some works [18], [19] use task replicas to speed up the straggled tasks for queries with task fanout, they do not take query fanout into individual task resource allocation. As a result, these approaches fail to maximize system resource utilization and hence, do not achieve the design objectives.

The implication of the above observation is significant. First, even with all the queries sharing a given tail latency SLO, the tasks belonging to queries with different fanouts should be treated differently. Any solution that fails to take the query fanout explicitly into account is guaranteed to result in resource overprovisioning, simply because such a solution will have to allocate task resources based on the worst-case task resource demand. This partially explains why the way to meet stringent tail latency SLOs for large-scale user-facing services in today's datacenters is normally through resource over-provisioning [2].

Second, consider a user-facing service that supports multiple classes of queries with a higher class requiring a more stringent tail latency SLO. Since the resource demand for a task is a function of not only the tail latency SLO but also the fanout of the query the task belongs to, it becomes apparent that a task associated with a query of a lower class but with a larger fanout may end up demanding more resources than a task in a query of a higher class but with a smaller fanout. This renders class-based task queue scheduling disciplines (e.g., PRIority-based task Queuing (PRIQ) [2], [20], [21]), task fanout-unaware queue management policies (e.g., the Tail-latency-SLO-aware Earliest-Deadline-First Queuing (T-EDFQ)), or task preemption [22] policies inadequate to achieve the design objective. This may also render some task reordering solutions solely based on task sizes [8], [23] inadequate.

In this paper, we propose TailGuard, a Tail-latency-SLO-and-Fanout-aware Earliest-Deadline-First Queuing(TF-EDFQ) policy, as a first step towards achieving the design objective for user-facing services in general. As a top-down approach, TailGuard decouples the upper query level design from the lower task level design. First, at the query level, a task decomposition technique is developed to translate the query tail latency SLO for a query with a given fanout into a task pre-dequeuing time[1] deadline for tasks spawned by the query at the task level, reflecting the resource demand of the tasks. This effectively decomposes a hard cotask scheduling problem at the query level into individual queue management subproblems at the task level. Second, at the task level, a single TF-EDFQ corresponding to a task server is used to enforce the task pre-dequeuing time deadlines, as a way to differentiate resource allocation for tasks with different resource demands. TailGuard addresses a critical challenge by prioritizing the most urgent tasks to meet query tail latency SLOs, thereby improving system utilization. In principle, TailGuard permits unlimited number of query classes and is lightweight, as it incurs minimum overhead for task pre-dequeuing deadline estimation and requires to implement only a single earliest-deadline-first queue per task server for any user-facing applications. A query admission control scheme is also developed to provide tail latency SLO guarantee in the face of resource shortages. A primary version of this paper appeared in [24].

TailGuard is evaluated against First-in-First-Out (FIFO) queuing, PRIQ and T-EDFQ by both simulation and testing in the Amazon EC2 cloud. Three traces generated from the Tailbench benchmark suite [25], featuring web search, in-memory key value store, and transactional database applications, are used as input. The results demonstrate that TailGuard can improve resource utilization by up to 80% , while meeting the targeted tail latency SLOs, as compared with the other three policies. The query admission control scheme is also tested and the results indicate that it can indeed provide query tail latency SLO guarantee. Finally, TailGuard is implemented and tested in a highly heterogeneous Sensing-*a*s-a-Service (SaS) testbed for an edge-based temperature-and-humidity sensing service, with test results in lines with the other ones.

II. Background and Related Work

A. User-Facing Services

User-facing services are a predominant class of workloads in today's cloud and have also emerged as an important class of workloads in an edge-cloud ecosystem, generally known as SaS[2] [26]. Predominant user-facing services are driven by queries that require query responsiveness in sub-seconds to seconds and may need to touch on massive datasets, which are typically carried out in a data parallel fashion. The working dataset for a service are distributed to many task servers. Accordingly, a query may spawn a number of tasks to be dispatched to some or all of these task servers to be processed.

A user-facing service may be launched in a dedicated datacenter cluster owned by a service provider, e.g., the web search service by Google, in a cloud by a tenant who rents cloud resources from a cloud service provider (e.g., Amazon cloud), or in an edge-cloud ecosystem owned by multiple stake-holders, including individuals who own the sensing data and/or edge devices and cloud service providers.

Fig. 1 depicts a generic user-facing service processing model [27], [28]. It is composed of three parts, including a front-end server, a mid-tier server (called query handler in this paper), and a set of back-end leaf servers (called task servers in this paper),[3] each hosting a piece of the total dataset, also known as a shard, a partition, or a published sensing dataset (e.g., in an edge node).

When a user request arrives at the front-end server, its workflow is parsed to generate a set of queries to be issued sequentially to the query handler at the mid-tier server. Due to

[1]A task pre-dequeuing time includes all the the delays a task experiences before it is dequeued from the task queue.

[2]For an SaS, users send sensing requests to the cloud. The cloud then dispatches related query tasks to geo-distributed edge nodes to acquire desired sensing data collected and processed through crowdsensing, which are subsequently merged in and returned to the users from the cloud.

[3]Task servers are also known as, e.g., workers, virtual-machines (VMs), containers, or edge nodes, depending on the specific services to be studied.

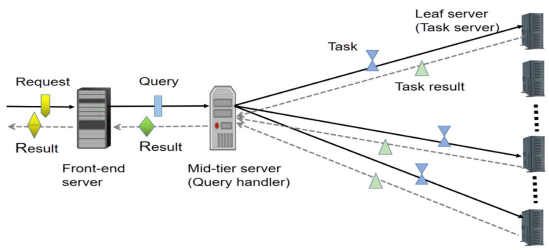


Fig. 1. A typical user-facing application process architecture.

query/task dependency, the next query cannot be issued until the current one finishes. For each query received, the query handler spawns a number of tasks for the query and dispatches them to the queues corresponding to the task servers[4] that will serve them when they reach the queue heads. The tasks for the same task server are queued based on a given queuing mechanism. In practice, task servers are usually allocated dedicated CPU/memory/storage resources in the form of, e.g., cores, VMs, containers, or pods, as well as fix-sized data shards, forming a more or less homogeneous task server cluster. As a result, the differentiation of resource allocation among tasks with different resource demands are mainly through task queuing policies, e.g., PRIQ [2], [20], [21], task-reordering-based queuing [8], [23], or EDFQ, unless task-aware resource auto-scaling [29] is allowed.

Upon completion of the execution of a task, the task result is returned to the query handler to be merged with the task results from the other tasks of the query. The query finishes when all the task results are merged and sent to the front-end server. Hence the task response time for the slowest task dictates the query response time. In turn, the request completes when the last query in the request finishes.

B. Tail Latency Aware Solutions for User-Facing Services

User-facing services generally call for query tail latency SLO. A query tail latency SLO can be interpreted in two different ways. For example, for a query tail latency SLO expressed as the 99th percentile query tail latency of 300 ms, it may mean that out of 100 queries, probabilistically 99 queries will experience query response times within 300 ms; it may also be interpreted as that with 99% probability, a query will be fulfilled within 300 ms. The second interpretation, applied in this paper, means that a query tail latency SLO can be applied to any given query, called per tail latency SLO hereafter.

Many works have been devoted to addressing query tail latency related issues for user-facing services, which can be broadly classified into two categories, i.e., *outlier-alleviation*, focusing on curtailing the tail length of the task response time to improve overall query tail latency performance, and *tail latency SLO guarantee* for queries sharing a single tail latency SLO. In what follows, we elaborate more on the solutions in the two categories, respectively.

Outlier-Alleviation: Most existing solutions fall into this category. Some typical examples in this category are listed as follows. Solutions based on task-size-aware task reordering in a task queue [8], [23], [30], [31] are proposed to avoid head-of-line blocking of small-sized tasks by large-sized ones to reduce the mean task latency. Task-aware scheduling schemes [13], [14], [15], [32], [40] are designed to shorten the tail latency for tail latency critical tasks in workloads with both batch and tail latency critical queries. Redundant-task-issue solutions [7], [12], [18], [19] are developed to reduce the query tail latency by allowing a task to be issued to multiple task server replicas for queries with task fanout. Task execution time prediction through workload profiling [9], [11], [16], [17], [27] and machine learning [33], [34] are widely employed to adjust the level of parallelism to remove task bottlenecks or to avoid sending tasks with predicted long execution time to poorly performing task severs to reduce task tail latency. Solutions based on synchronized garbage collection for all task servers [3], [35] are proposed to minimize variabilities of task execution times among parallel tasks to reduce query tail latency. Solutions that allow partial results to be returned to fulfill a query, e.g., [36], can maintain more predictable query tail latency at the cost of possible loss of partial results. Dynamic resource allocation based on the feedback loop control mechanisms [8], [37] are proposed to help reduce query tail latencies. CPU power control schemes [18], [38] are developed to dynamically adjust voltage and frequency scaling (DVFS) for task servers based on task execution time to save energy and maintain low task tail latency. A query fanout control scheme [4] is designed to control the fanout in queries to optimize the system performance. A transaction scheduling solution for geo-distributed databases [39] uses transaction timestamps to reduce both mean and tail latencies for edge computing. All these solutions help reduce the query tail latency, but cannot provide SLO guarantee.

Tail Latency SLO guarantee: There are a few existing solutions in this category, including Cake [41], PriorityMeister [42], SNC-Meister [43], WorkloadCompactor [44] and PSLO [45], all for shared datacenter storage applications. All these solutions, except Cake, aim at meeting a single query tail latency SLO for all queries with fanout of one only. Cake can handle fanout of more than one, but is unable to enable per-class or per-query tail latency SLOs, as it relies on direct measurement of the overall tail latency statistics as input for control, resulting in fanout-unaware resource overprovisioning. Clearly, a solution based on direct tail latency statistics measurement like Cake cannot be extended to allow per-query resource allocation, simply because the needed statistics are unavailable at this granularity. Some tail latency SLO guaranteed solutions for micro-service such as GrandSLAm [46] and Sinan [47] are proposed. But, again, they cannot support per-query tail latency SLO.

III. TAILGUARD

In this section, we first give the TailGuard query processing model. Then we present the task pre-dequeuing time budget decomposition solution and address its implementation issues. Finally we present the query admission control scheme. The major symbols used in TailGuard are listed and defined in Table I.

[4]Note that the queuing may take place either centrally at the query handler or at individual task servers.

TABLE I
THE SYMBOLS USED IN TAILGUARD

Symbol	Description
N	number of task servers
M	number of queries in a request
k_f	fanout of a query
T_b	task pre-dequeuing time budget for a query
t_0	query arrival time
t_D	task pre-dequeuing time deadline, $t_D = t_0 + T_b$
t_{pr}	task pre-dequeuing time
t_{po}	task post-queuing time or unloaded task response time
t_r	task response time, $t_r = t_{pr} + t_{po}$
x_p^{SLO}	pth percentile query tail latency SLO
$x_p^u(k_f)/x_p(k_f)$	unloaded/loaded pth percentile tail latency for a query with fanout k_f
$F_l^u(t)/F_l(t)$	CDF of unloaded/loaded task response time with respect to task server l
$F_Q^u(t)/F_Q(t)$	CDF of unloaded/loaded response time for a query
$P(k_f)$	probability of a query with fanout k_f

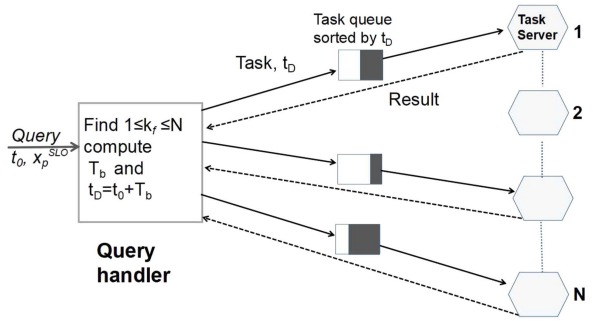


Fig. 2. TailGuard query processing model. A task queue for a task server can be set in the task server or in the query handler.

A. Problem Statement

The key to the design of TailGuard is the task pre-dequeuing time budget decompostion or pre-dequeuing time deadline estimation. The task pre-dequeuing time deadline estimation problem can be formally stated as follows: *For a query with fanout, k_f, a given tail latency SLO in term of x_p^{SLO}, and arrival time, t_0, find the task queuing deadline, $t_D = t_0 + T_b(x_p^{SLO}, k_f)$, for tasks spawned by the query*. Here, $T_b(x_p^{SLO}, k_f)$, the task pre-dequeuing time budget, is the maximum allowable task pre-dequeuing time before the task must be dequeued and given/sent to the task server to be processed, in order to meet the query tail latency SLO. Central to the TailGuard design is to calculate $T_b(x_p^{SLO}, k_f)$.

B. TailGuard Query Processing Model

Consider a query processing model directly derived from Fig. 1, as depicted in Fig. 2. It is composed of a query arrival process, a query handler, and N task servers. The query arrival process characterizes the randomness of queries arriving at the query handler.

At the query level, upon receiving a query at time, t_0, the query handler first determines how many tasks (i.e., the query fanout, k_f) need to be spawned and to which k_f task servers these tasks need to be dispatched. The query handler estimates task pre-dequeuing time budget T_b and computes the task pre-dequeuing time deadline $t_D = t_0 + T_b$, shared by all the tasks associated with the query.[5] Finally, the tasks, together with their deadlines, are dispatched to the queues corresponding to the task servers. Since task pre-dequeuing time budget, T_b, is an explicit function of both x_p^{SLO} and k_f for the query, *TailGuard by design permits per-query tail latency SLOs*. At the task level, each task queue adopts a TF-EDFQ, based on $t_D(x_p^{SLO}, k_f)$. When a task is to be enqueued at a task queue, if the corresponding task server is idle, the task is serviced immediately, otherwise, it is inserted into the task queue with tasks ordered in increasing order of t_D's, hence with the task of the smallest t_D at the head of the queue. Whenever a task in service finishes, the task at the head of the queue is put in service immediately. Finally, upon the completion of execution of a task, the task result is sent back to the query handler to be merged. A query finishes as soon as the merging of all the task results completes.

TailGuard ensures that tasks with a higher chance to cause the violation of the associated query tail latency SLO will be serviced earlier, thus improving the system utilization.

C. Estimation of Task Pre-Dequeuing Deadline

The key to the design of TailGuard is the task pre-dequeuing time budget decompostion or estimation of the pre-dequeuing time deadline. In this subsection, we first present the solution for estimation of task pre-dequeuing time deadline and then propose a way to implement it.

1) Solution: Now we present the solution of estimation of task pre-dequeuing deadline. First, we note that the task response time (also called loaded task response time), t_r, can be generally expressed as, $t_r = t_{pr} + t_{po}$, where t_{pr} represents the task pre-dequeuing time and t_{po} stands for task post-queuing time or unloaded task response time. t_{pr} is composed of task scheduling time and task queuing time, if task queuing takes place centrally at the query handler. It also includes task dispatching time, if task queuing occurs at the task server. t_{po} includes all the times the task incurs after de-queuing.

Now we assume that the Cumulative Distribution Function (CDF) of unloaded task response time t_{po}, $F_l^u(t)$, with respect to task server, l, can be measured and updated for all task servers $l = 1, \ldots, N$. Furthermore, let $x_p^u(k_f)$ and $F_Q^u(t, k_f)$ represent the pth percentile unloaded query tail latency and the CDF of unloaded query latency for queries with fanout k_f, respectively. Here, a query latency is considered as unloaded (loaded) if the query response time does not (does) include pre-dequeuing time, t_{pr}. Also define $n = n(i)$ to be the mapping between the i-th task in a query and the n-th task server the task is dispatched to, for $i = 1, \ldots, k_f$. Clearly, the unloaded query latency is the task post-queuing time of the slowest of all k_f tasks. According to the ordered statistics [10] by assuming that all task service

[5]The rationale for assigning the same budget to all the tasks of a query is as follows. Mathematically, with two reasonable assumptions made, i.e., a task resource demand is an decreasing function of the task budget and the sum of the task budgets for all the tasks in a query must be upper bounded to meet a given query tail latency SLO, it can be easily shown that assigning the same budget results in the minimum overall resource allocation.

times are independent, we have,

$$F_Q^u(t, k_f) = \prod_{i=1}^{k_f} F_{n(i)}^u(t). \tag{1}$$

By definition, we have,

$$\frac{p}{100} = F_Q^u\left(x_p^u(k_f), k_f\right). \tag{2}$$

Now assuming that all the tasks in a query experience the same pre-dequeuing time t_{pr}, then we have,

$$x_p(k_f) = x_p^u(k_f) + t_{pr}. \tag{3}$$

This means that the task pre-dequeuing time budget $T_b(x_p^{SLO}, k_f)$ can be defined as, $T_b(x_p^{SLO}, k_f) = x_p^{SLO} - x_p^u(k_f)$, or equivalently, the task pre-queuing time deadline can be defined as,

$$t_D = t_0 + T_b\left(x_p^{SLO}, k_f\right) = t_0 + x_p^{SLO} - x_p^u(k_f). \tag{4}$$

In other words, for a query arrived at $t = t_0$, as shown in Fig. 2, so long as all the tasks belonging to this query are dequeued no later than t_D, the query tail latency SLO, x_p^{SLO}, is guaranteed to be met.

Ideally, under the work conserving condition,[6] if a queuing policy can ensure that all the tasks exactly meet their queuing deadlines, the design objective is achieved. In practice, however, such a queuing policy may not exist. As a first step, TailGuard adopts EDFQ based on t_D, i.e., TF-EDFQ, to enforce the task queuing deadlines. This queuing policy can ensure that the task with the earliest queuing deadline is placed at the head of the queue before deadline. However, it cannot guarantee that the task at the head of the queue can always be served before deadline, simply because the task ahead of it may be still in service when the deadline is reached. On the other hand, the task may also have a chance to be dequeued before deadline, if the task server becomes idle before deadline. This implies that TailGuard may tolerate a small percentage of tasks missing their deadlines without violating the tail latency SLOs as the tail latency is a probabilistic measure.

A remark on meeting request tail latency SLO: For a user-facing request with multiple round queries, we can prove that the pre-dequeuing time budget for the request is the request tail latency SLO minus the unloaded request tail latency. This budget can be arbitrarily assigned to any query, provided that the sum of total query budgets does not exceed the request budget.

Consider a request composed of M queries to be issued sequentially and with the request tail latency SLO expressed in terms of the pth percentile of request latency of, $x_p^{R,SLO}$. Now, the request response time $t_r^R = \sum_{i=1}^M t_{r,i}$, where $t_{r,i}$ is the query response time for the i-th query. Although this relationship is an additive one, the one for the corresponding tail latency is not. As the CDF of the request response time, $F_R(t)$, is the convolutions of all the CDFs of the constituent query response times, in general, $x_p^{R,SLO} < \sum_{i=1}^M x_{p,i}^{SLO}$, making query decomposition for requests difficult. In what follows, we show that the above task decomposition technique can be generalized to establish an additive relationship between the request pre-dequeuing time budget and task pre-dequeuing time budgets for the constituent queries, paving the way for the development of a task decomposition technique for requests.

Define unloaded request latency, $t_{po}^R = \sum_{i=1}^M t_{po,i}$, and the CDF of the unloaded request response time, $F_R^u(t)$, to be the CDF of t_{po}^R, where $t_{po,i}$ is the unloaded query latency for the i-th query. Further assume that all the tasks of query i have the same pre-dequeuing time, $t_{pr,i}$, and define request pre-dequeuing time, $t_{pr}^R = \sum_{i=1}^M t_{pr,i}$. Then we have the loaded request response time $t_r^R = \sum_{i=1}^M (t_{po,i} + t_{pr,i}) = t_{po}^R + t_{pr}^R$. Clearly, by substituting t_r, t_{pr}, t_{po}, F_Q, and F_Q^u with t_r^R, t_{pr}^R, t_{po}^R, F_R, and F_R^u, respectively, and following (3), we have,

$$x_p^R = x_p^{Ru} + t_{pr}^R = x_p^{Ru} + \sum_{i=1}^M t_{pr,i}, \tag{5}$$

where x_p^R and x_p^{Ru} are the loaded and unloaded pth percentile tail latency of the request. (5) means that the request pre-dequeuing time budget, $T_b^R = x_p^{R,SLO} - x_p^{Ru}$, and it is additive, i.e., $T_b^R = \sum_{i=1}^M T_{b,i}$, here $T_{b,i}$ is the task pre-dequeuing budget for query i, for $i = 1, \ldots, M$.

Note that as long as T_b^R (i.e., $t_{pr}^R \le T_b^R$) is met, the request tail latency SLO will be met, regardless the assignments of $T_{b,i}$'s. However, different assignments may lead to different resource utilizations. Hence, a key challenge that will be the main focus of our future work is: with a given total budget T_b^R, how to assign budgets $T_{b,i}$ to individual queries so that the resource utilization is maximized.

A remark on the relationship with outlier-alleviation solutions: Finally, note that TailGuard is orthogonal and complementary to many of the existing outlier-alleviation solutions. An outlier-alleviation solution can help effectively reduce the tail lengths of the task service time distributions, $F_l(t)$'s, hence, improving the achievable query tail latency SLOs. With an outlier-alleviation solution, Tailguard only sees a different unloaded query response time distribution, and hence can directly use it to compute the task pre-dequeuing time budget.

2) Practical Considerations in TailGurad's Implementation: Now we discuss the implementation issues including how to estimate and update the unloaded task CDFs. The above solution for estimation task pre-dequeuing time deadline requires the availability of the task post-queuing time distributions, $F_l(t)$, for all the task servers, which must be conveyed to the query handler for task pre-dequeuing time budget estimation. Here, we propose an approach to estimate $F_l(t)$'s by means of a combined initial offline estimation process and a periodical online updating process.

Offline Estimation Process: As mentioned earlier, user-facing services are likely to run in a more or less homogeneous cluster. So before the service starts, we set $F_l(t) \approx F(t)$, for $l = 1,...N$. This lends us a handy way to perform an initial offline estimation of only a single distribution function $F(t)$, which serves as the initial distribution for all the task servers.

[6]The work conserving condition refers to the condition whereby the task server is always busy as long as there are unfinished tasks at the server.

More specifically, use a query handler and single task server and load it with a typical task workload trace to collect a sufficient number of samples of task post-queuing times offline. Then use these samples to construct $F(t)$ to be used as the initial distribution function for all task servers. This will allow task queuing deadlines to be estimated at the very start of a user-facing service.

Online updating process: To account for the inevitable heterogeneity in practice (e.g., due to skewed workloads, uneven resource allocation and resource availability changes), $F_l(t)$'s must be periodically updated online. Fortunately, this can be done with low cost. When the query handler receives and merges the task result for a task from task server l, it uses the current time minus the task dequeue time (which is either locally available if the queuing takes place in the query handler, or comes with the task result from the task server l) as the post-queuing time for the task to update $F_l(t)$. This updating process accounts for all the possible post-queuing delays incurred by the tasks, including the long delays caused by outliers. Hence TailGuard captures heterogeneity through online updating process.

TailGuard implementation complexity: The fact that the unloaded task tail latency is independent of system load, and hence the query queuing deadline is independent of system load. This simplifies the implementation of TailGuard because it only needs to measure a single unloaded tail latency. The computation complexities for both task queuing deadline estimation and queuing management in TailGuard are low. The former entails the evaluation of two equations, i.e., (2) for $x_p^u(k_f)$, which can be done in the background for all possible k_f's in advance and updated when $F_l(t)$'s change and (4) for each query. The latter requires the management of a single EDFQ. As a result, TailGuard is a lightweight solution.

D. Query Admission Control

TailGuard can provide tail latency SLO guarantee for all queries, when there are enough resources to sustain the workload. In the presence of resource shortages due to, e.g., sudden surges of workloads or hardware/software failures, some upcoming queries may need to be rejected to ensure that all admitted queries can meet the prepaid tail latency SLOs. Query admission control is particularly desirable in the case where resource auto-scaling cannot be done, e.g., due to monetary budget or resource constraints.

With EDFQ queuing, a task may be dequeued before its deadline. As a result, statistically, the query tail latency SLO can still be met even if some tasks are dequeued after their deadlines (i.e., violating their deadlines). We tested TailGuard using various workloads and found that the query tail latency SLOs can still be met, when a small portion (less than 2% in our tests) of tasks miss their deadlines. With this understanding, TailGuard sets an upper bound threshold for the percentage of tasks missing their deadlines, R_{cth}^U, for query admission control. Whenever the task deadline miss ratio hits the upper bound threshold, the system immediately triggers admission control. If the task queuing takes place centrally at the query handler, the information on whether a task misses its deadline or not is immediately available to the query handler, otherwise, this information can be piggybacked on the task results returned from the task severs. The query handler can update the task deadline violation ratio in a given moving time window. When the ratio exceeds the upper bound threshold, upcoming queries are rejected, till the ratio falls back below the threshold again. The moving time window can be set to be the same as the time window in which the tail latency SLOs should be guaranteed.

E. Resource Auto-Scaling

In case resource can be dynamically scaled, a resource auto-scaling scheme should be enabled to guarantee the query tail latency SLO while maximizing system resource utilization. Resource auto-scaling can be categorized into two types: vertical and horizontal. Vertical scaling involves the addition or removal of CPU and memory resources, while horizontal scaling pertains to the management of the number of containers (i.e., adding or removing containers). The mechanism used depends on resource availability.

We set an upper bound threshold, R_{sth}^U, and a low bound threshold, R_{sth}^L, for the task deadline miss ratio to facilitate resource upscaling and downscaling. Whenever the task deadline miss ratio hits the upper or lower bound threshold, the resource is scaled up or down. Clearly, as long as the task deadline miss ratios for all tasks fall between the lower and upper bound thresholds, the system is expected to work at high resource utilization without violating request tail latency SLOs. Both thresholds can be set offline through measurement and be dynamically adjusted online.

IV. Performance Evaluation

To cover a wide range of applications, TailGuard is firstly evaluated based on simulation using the workload statistics for three datacenter applications available in Tailbench [25] as input. We first characterize the workload and present the simulation results along the fanout and service class dimensions; with an outlier-alleviation scheme; and with query admission control. We then implement and test TailGuard in the Amazon EC2 cloud. Finally we verify TailGuard in a highly heterogeneous SaS testbed.

The performance of TailGuard will be compared against FIFO, PRIQ and T-EDFQ. In terms of queuing policy, FIFO is simply a first-in-first-out queuing policy. PRIQ assigns tasks of different classes to different queues with strict priorities given to the queue of a higher class over that of a lower class. T-EDFQ works the same way as TailGuard without considering the query fanout, i.e., the pre-dequeuing time deadline is set as $t_D = t_0 + x_p^{SLO}$. Clearly, both PRIQ and T-EDFQ degenerate to FIFO if all queries have the same tail latency SLO, i.e., the case with a single class.

A. Workloads

For simulation, a user-facing workload must be characterized by a query arrival process, a query fanout distribution and a task post-queuing time distribution. Unfortunately, the available real

traces simply do not contain the needed information. Although traces for commercial user-facing services in cloud are available, e.g., those made available by Google [20], [29] and Alibaba [2], [21], they only include the CPU and memory usage information for task servers, not the information needed to drive the simulation at the task level, including the arrival process, query fanouts and task service times. Hence, we resort to modeling for the first two and benchmarks for the third one, as described in detail below.

First, since the Poisson process has been widely recognized as a good model for cloud applications in general [27], [48], [49], by default, we assume that the query arrival process is Poisson with mean arrival rate, λ, a tunning knob to adjust the system load. Meanwhile, to test the performance sensitivity of TailGuard with respect to the burstiness of query arrivals, a burstier arrival process, i.e., the Pareto arrival process [50], is also used in one simulation case. In the Pareto process, the time interval between any two consecutive query arrivals is randomly sampled from the following heavy-tailed Pareto distribution,

$$F_X(\alpha, t) = \begin{cases} 1 - \left(\frac{x_m}{x}\right)^{\alpha} & x > x_m \\ 0 & x < x_m, \end{cases} \tag{6}$$

where x_m is the minimum time interval, a tuning knob of the load. α determines the variance of the time interval, a measure of burstiness, which is less than 2 in practice [51]. Hence we set $\alpha = 1.4$ to catch heavier burstiness than the Poisson process.

Second, although a few publications do offer fanout distribution, $P(k_f)$, for k_f = 1,...,N, for the user-facing services, e.g., the Facebook social networking service [5], they do not provide task service times needed for the task-level simulation. This, however, should not be too much of a concern, as TailGuard needs to be applicable to both the existing and future workloads whose $P(k_f)$'s are not known yet. Hence, we adopt quite different $P(k_f)$ models for different case studies to gain a wide coverage. As we shall see, for all those cases tested, TailGuard consistently outperforms the FIFO, PRIQ and T-EDFQ queuing policies, which strongly suggests that the TailGuard's performance gain is insensitive to $P(k_f)$'s.

Third, as a solution meant to be used by the current and future user-facing services in general, TailGuard should be tested against user-facing services with a wide range of task service time distributions. To this end, we resort to Tailbench [25] to gain access to applications with a wide range of task service time distributions. Tailbench provides eight user-facing task benchmarks. Each of these workloads allows a sufficiently large number of task service time samples to be collected to construct $F(t)$ for task service time, assuming that the post-queuing time, t_{po}, is dominated by the task service time, for the lack of the information about the rest of the post-queuing delays. We further assume that $F_l(t) = F(t)$ for l = 1,...,N, i.e., the homogeneous case, which do not change over time (all the other delays and heterogeneity will be accounted fully in the SaS case study). These workloads can be classified into three groups with distinct characteristics for $F(t)$, and the CDFs of each group of applications are similar. We select one workload from each group to be tested, including Masstree for in-memory key-value store, Shore for SSD-based transactional database and Xapian for web search.

TABLE II
THE MEAN TASK SERVICE TIME T_m (ms) AND THE UNLOADED 99TH PERCENTILE QUERY TAIL LATENCY x_{99}^u (ms) WITH VARIOUS FANOUTS

Bench	T_m	$x_{99}^u(1)$	$x_{99}^u(10)$	$x_{99}^u(100)$	$x_{99}^u(1000)$
Masstree	0.176	0.212	0.247	0.473	1.041
Shore	0.341	2.117	2.721	2.829	6.019
Xapian	0.925	2.592	2.998	3.307	7.149

Fig. 3 depicts the CDFs and the unloaded 95/99th percentile task tail latencies for the three workloads. Table II also gives the related statistics, including the mean task service time (T_m) and the unloaded 99th percentile query tail latency at fanouts k_f = 1, 10, 100, and 1000, derived from (1) and (2). First, we note that Masstree has the smallest mean and variance of task service time with a sharp increase of CDF around the mean. Shore exhibits the largest variability of unloaded task service time of all the three with $x_{99}^u(1000)$ reaches 6.019 ms, about 18 times of its mean service time, exhibiting relatively long tail behavior. Xapian has the largest mean task service time of the three.

B. Impact of Query Fanout

In this subsection, we focus on testing the impact of the query fanout. We present two cases, i.e., a single class case and a two-class case. Consider a cluster of size $N = 100$ and three different types of queries corresponding to three different fanouts 1, 10 and 100, similar to the testing scenario in [54], in which fanouts 1, 8 and 33 are used. Further assume $P(1) = 100/111$, $P(10) = 10/111$, and $P(100) = 1/111$, i.e., the probability for a fanout is inversely proportional to the fanout itself, similar to the one observed by Facebook [5]. This makes the total numbers of tasks from the three query types to be, on average, the same. For a given tail latency SLO of x_{99}^{SLO}, the task pre-dequeuing time budget for a query with fanout k_f (1, 10 or 100) is $T_b = x_{99}^{SLO} - x_{99}^u(k_f)$.

Note that meeting the tail latency SLO for queries as a whole does not guarantee that queries of individual types can meet the tail latency SLO. Hence, in the following simulation, we measure the tail latency for each type of queries and identify the maximum load at which all three types of queries meet their tail latency SLOs.

We first consider the case with a single service class, i.e., all the queries have to meet a single SLO. In this case, both PRIQ and T-EDFQ behave exactly the same as FIFO and hence, we only compare TailGuard against FIFO. Fig. 4 depicts the maximum loads that can meet the tail latency SLO for TailGuard and FIFO for four different tail latency SLOs (these SLOs are chosen such that the corresponding maximum loads fall in the range of 20% to 60% which are the typical system loads for the current commercial clouds serving user-facing services [20], [21]). This gives us a good idea about TailGuard's performance gain/loss with respect to those of the currently practiced ones. As we can see, for all the cases, TailGuard achieves higher loads compared to FIFO, while meeting the same tail latency SLO.

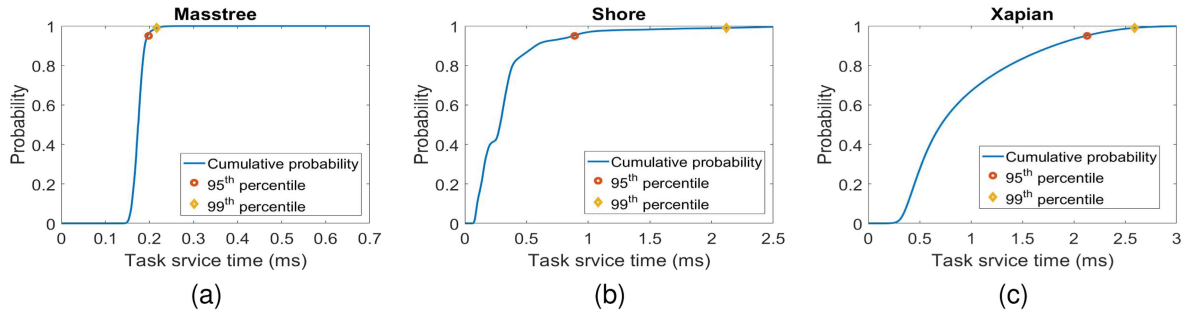


Fig. 3. The CDFs and the unloaded 95th and 99th percentile task tail latencies of the three Tailbench workloads: (a) Masstree; (b) Shore; and (c) Xapian.

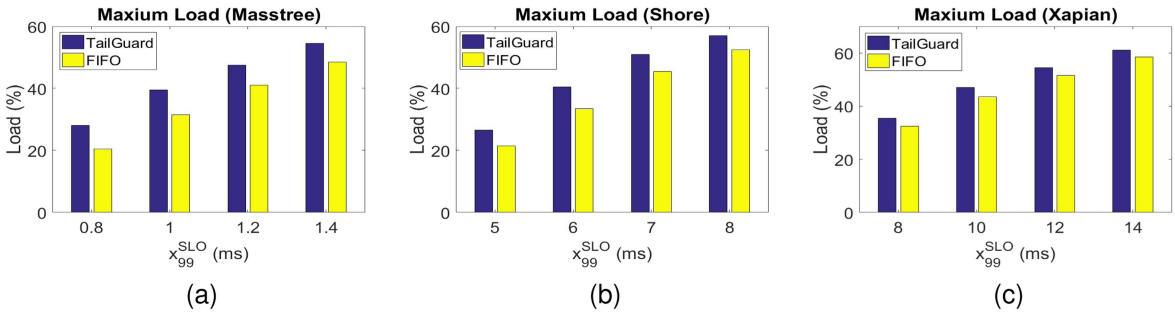


Fig. 4. The maximum loads with a single service class. in the three workloads. (a) Masstree; (b) Shore; and (c) Xapian.

TABLE III
THE 99TH TAIL LATENCY (ms) OF THREE TYPES OF QUERIES AT MAXIMUM LOADS FOR THE MASSTREE WORKLOAD

		K_f= 1	K_f= 10	K_f=100
x_{99}=0.8	FIFO	0.439	0.594	0.798
	TailGuard	0.572	0.745	0.797
x_{99}=1.0	FIFO	0.533	0.731	0.997
	TailGuard	0.705	0.941	0.994
x_{99}=1.2	FIFO	0.647	0.889	1.192
	TailGuard	0.817	1.098	1.193
x_{99}=1.4	FIFO	0.751	1.061	1.389
	TailGuard	0.945	1.262	1.392

The performance gain increases as the tail latency SLO becomes tighter. This is because a query with a higher fanout has a tighter task pre-dequeuing deadline and hence, higher chance to violate the tail latency SLO. Therefore, TailGuard that reorders the tasks based on pre-dequeuing deadlines can help meet the tail latency SLO for all queries, resulting in higher performance than FIFO, especially when the tail latency SLO becomes more stringent. For example, for Masstree, the maximum load increases from 20% for FIFO to 28% for TailGuard at $x_{99}^{SLO} = 0.8$ ms, resulting in about 40% higher resource utilization. In other words, TailGuard can save 40% task server resources over FIFO (also PRIQ and T-EDFQ), while meeting the same tail latency SLO, hence reducing the cost.

To gain more insights, for Masstree, Table III gives the breakdowns of the tail latencies at the maximum loads for the three types of queries. First, we note that at the maximum loads, the query type with $k_f = 100$ barely meets the tail latency SLOs for both schemes. In other words, the maximum achievable load for both queuing policies are constrained by the query type with the highest k_f. For the other two query types, the tail latencies are smaller than the corresponding tail latency SLOs, implying that they get more resources than they need, especially for the one with $k_f = 1$. The performance gain for TailGuard comes from more balanced resource allocation among the three types, as evidenced by the closer tail latencies among the three types than those for FIFO. The results clearly indicate that the query fanout has to be taken into consideration in task resource allocation for meeting query tail latency SLO to maximize the system performance.

Nevertheless, for TailGuard, there is still much room left to be improved, with e.g., a tail latency gap of 32.5% (i.e., 100x(1.392-0.945)/1.392) at $k_f = 1$ and 9.9% (i.e., 100x(1.392-1.262)/1.392) at $k_f = 10$ left to be closed (see the case of $x_{99} = 1.392$ ms at the bottom). This happens simply because TailGuard provides less accurate deadline guarantee for tasks with tighter deadlines. More specifically, as mentioned earlier, TailGuard cannot guarantee that a task can always be put in service at the instant it reaches its deadline, as the processing unit in the task server may not be idle at that instant. This additional delay is likely to be statistically the same for all tasks regardless of their deadlines, meaning that tasks with tighter budgets will see relatively larger additional delays and hence, can sustain smaller maximum loads. To further balance the resource allocation among queries of different types and hence, improve the overall load, a possible solution is to adjust the deadline budget T_b for each query type dynamically based on the tail latency gap measured periodically at runtime. This approach, however, may not be scalable, as in practice, the number of types of queries, or the number of queries with distinct fanouts, may be large. In the following subsection, we propose to do so on a per-service-class basis instead to improve the scalability.

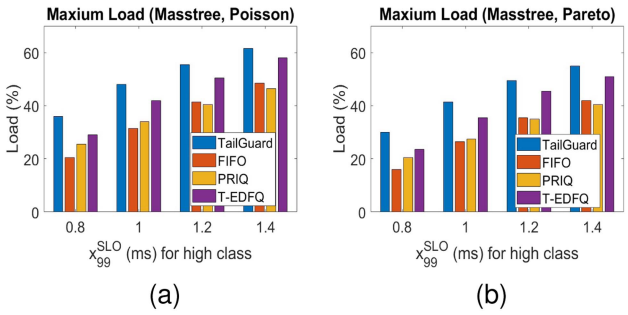


Fig. 5. The Maximum loads with two classes for the Masstree workload: (a) Poisson and (b) Pareto arrival process.

Now we consider the case with two service classes with the tail latency SLO of the lower class being 1.5 times of that of the higher class, i.e., $1.5x_{99}$, where x_{99} is the tail latency SLO for the higher class. Each query is randomly assigned to one of the two classes with equal probability. Both the Poisson and Pareto arrival processes are considered. Only the results for the Masstree workload are given as the the results for the other two workloads are similar.

Fig. 5 shows the maximum loads under which all queries can meet their tail latency SLOs. From the results (Fig. 5(a)) with the Poisson arrival process, we can see that the performance gains of TailGuard over FIFO increase to up to 80% , much higher than that in the single class case (i.e., up to 40%). FIFO treats every task equally. Hence its performance is constrained by the most resource demanding queries, i.e., the higher class queries with the largest fanout. The TailGuard performance gain is up to 40% with respect to PRIQ. PRIQ gives higher priority to the higher class queries, resulting in lower class queries having less resources to meet their tail latency SLOs. The TailGuard performance gain is up to about 22% with respect to T-EDFQ, smaller than that with respect to PRIQ. This means that by incorporating the actual tail latency SLO, rather than just the class information, T-EDFQ can allocate task resources more accurately than PRIQ does. In turn, TailGuard improves over T-EDFQ by further incorporating query fanout information in task resource allocation.

The performance gains for TailGuard against the other three schemes with the Pareto arrival process (Fig. 5(b)) are similar to those with the Poisson arrival process. Meanwhile, the maximum loads decease about 10% to 18% for all schemes compared to those with the Poisson arrival process. This means that the burstiness of query arrivals mainly impact the overall achievable load, but much less on the relative performance of different queuing policies. Hence, in the following cases studies, we only present those with the Poisson arrival process.

C. Multiple Tail Latency SLOs At a Fixed Fanout

Again, consider the cluster of size $N = 100$. Now all queries have the same fanout of $k_f = 100$, i.e., for each query, its tasks are served by all the task servers in the cluster in parallel, which is the case for web searching service. We evaluate the performance of TailGuard for workloads with two different service classes, denoted as Classes I and II. The tail latency SLOs for Class I/II

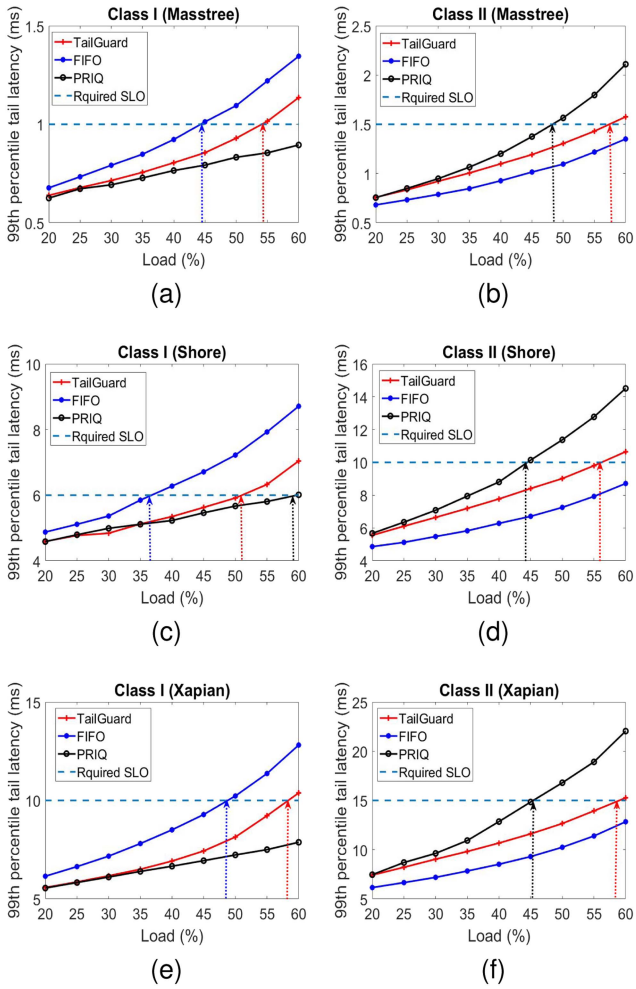


Fig. 6. The 99th percentile latency at different loads. The cyan line indicates the required tail latency and the arrows points to the maximum load that the tail latency can be met.

are 1/1.5, 6/10 and 10/15 ms for Masstree, Shore and Xapian, respectively. Again, these tail latency SLOs are chosen such that the achievable maximum load ranges from 20% to 60% . A query has equal probability to request for either of the two classes. For any query of a given class, by substituting the corresponding x_{99}^{SLO} and $x_{99}^{u}(100)$ from Table II into (4), we arrive at the task pre-dequeuing time budgets. For example, for Masstree, the budgets for classes I and II are 1-0.473=0.527 ms and 1.5-0.473=1.027 ms, respectively. As the fanout is the same for all queries, T-EDFQ behaves the same as TailGuard, and hence we compare the performance of TailGuard against both FIFO and PRIQ.

Fig. 6 presents the simulation results. For each plot, the cyan dash line represents the tail latency SLO for that class and the arrows, each having the same color as the tail latency curve for a queuing policy, indicate the maximum achievable loads that meet the tail latency SLOs.

As one can see, for all three workloads, FIFO, which is class unaware, gives equal resources to queries from both classes. Since the task resource demands or task pre-dequeuing time budgets for tasks from classes I and II are quite different, e.g.,

0.527 ms and 1.027 ms, respectively, as calculated above, for Masstree, indiscriminately allocating equal resources to tasks results in a very low achievable load for class I queries but very high achievable load for class II queries, e.g., 45% for class I, as shown in Fig. 6(a), and higher than 60% for class II, as shown in Fig. 6(b). Consequently, to meet the tail latency SLOs for both classes, FIFO allows the cluster to run at 45% for Masstree, 36% for Shore (see Fig. 6(c)) and 49% for Xapian (see Fig. 6(e)).

PRIQ, on the other hand, is class aware, but it gives strict priority to tasks in Class I over Class II. This results in unbalanced resource allocation in favor of Class I over Class II. Consequently, the maximum load for class II is about 48% for Masstree, and about 45% for both Shore and Xapian, while the maximum load for class I reaches more than 60% for all three workloads. Again, the low load for class II limits the overall achievable load that meets both tail latency SLOs.

In contrast, as a class-aware approach and with task budgeting, TailGuard can balance the resources allocated to tasks closely in proportion to their resource demands, resulting in much closer maximum loads for the two classes (i.e., within 5% difference) for all three workloads. As shown in Fig. 6, the maximum loads for Classes I and II for Masstree/Shore/Xapian are about 54% /51% /58% and 57% /56% / 59% , respectively. Hence, the maximum loads that meet both tail latency SLOs are 54% /51% /58% for the three workloads, respectively. The performance gain of TailGuard over FIFO and PRIQ are up to 40% (i.e, from 36% to 51%) compared to FIFO and up to 30% (i.e., from 45% to 58%) compared to PRIQ.

The above results also indicate that for TailGuard, the maximum achievable load is constrained by Class I, meaning that there is still room for improvement. This can be easily understood by the same reasoning we made earlier. Namely, TailGuard provides less accurate deadline guarantee for tasks with tighter budgets. As a result, tasks belonging to queries from Class I, which have a tighter budget, are likely to miss their pre-dequeuing deadlines at a lower load than those from the other class. As alluded to earlier, a potential solution to further improve the performance of TailGuard is to adjust the task delay budgets dynamically to close the gap. This can be done much easier between classes than between fanouts, as the number of classes to be enforced is likely to be small, say, a dozen at most. Although developing such a solution is left as our future work, in what follows, we want to test how much performance improvement one may achieve by properly adjusting the budgets. Since the load gap between the two classes for Xapian is already very small, i.e., 1% (58% versus 59%), we consider Masstree and Shore only, which have load gaps of 3% and 5% , respectively.

We adjust the task delay budgets for Class II tasks for both Masstree and Shore. By experiment, we find that by adding about 15% and 18% to the original budgets of 1.027 ms and 7.171 ms for Masstree and Shore, respectively, the maximum achievable loads for both classes become almost indistinguishable, reaching about 56% and 53% (i.e., improving about 2% compared without budget adjustment) for Masstree and Shore, respectively, as shown in Fig. 7. The performance improvement is relatively small due to the presence of only two classes. As the number of classes increase, the disparity among maximum loads may become more pronounced. The budget adjustment scheme could be put into action when the performance gain is sufficiently large.

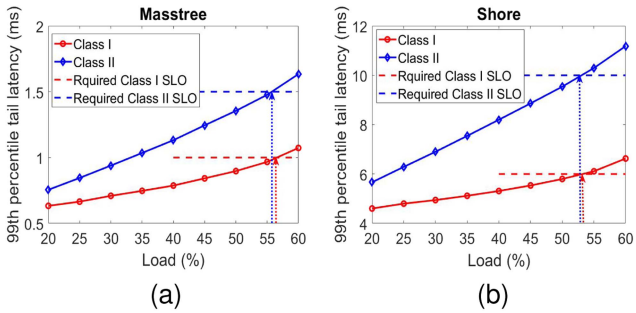


Fig. 7. The 99th percentile tail latency of two classes after adjusted task pre-dequeuing time budgets.

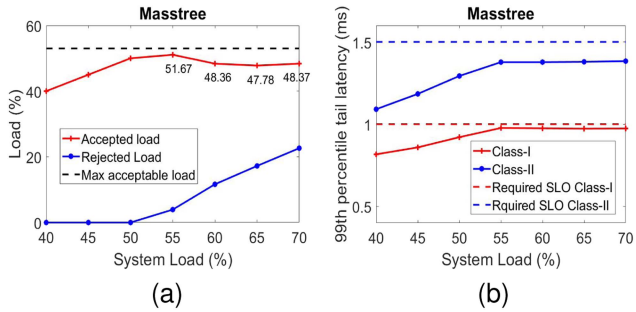


Fig. 8. TailGuard with query admission control. (a) is the accepted/rejected load; and (b) is the query tail latency for Class I and II.

D. TailGuard With Query Admission Control

The TailGuard query admission control scheme is tested here. Consider the same case presented in Section IV-C (only the result of Masstree is given due to limited space). We first run TailGuard without admission control to find the task queuing deadline violation threshold R_{cth}^{U} at the maximum acceptable load when TailGuard can barely provide the tail latency SLO guarantee. The maximum acceptable load thus found is about 54% and the corresponding threshold is 1.7% . We use a moving window with size of 1000 queries (or 100000 tasks) to compute the task queuing deadline violation ratio.

Fig. 8 shows the accepted/rejected loads and the query tail latencies at different loads. First, we see that the query tail latency SLOs for both classes are guaranteed at all loads. When the load is over the maximum acceptable loads, the query tail latency of Class I closely approaches its tail latency SLO, while the tail latency of Class II is a little below its SLO. This is due to the fact that Class I tasks have tighter pre-dequeuing time budgets to meet and hence have higher chances to miss the pre-dequeuing time deadlines as we explained in Section IV-C. Second, we note that the accepted loads (the load is computed using the accepted queries only) closely approach its respective maximum acceptable loads (within 2.5%). Further increasing the load beyond the maximum acceptable loads, the accepted load drops to about 6% below the maximum acceptable loads. There are two reasons for this to happen. First, TailGuard may

not drop the exact number of queries needed to perfectly meet the tail latency SLO. Second, just like any feedback loop control solutions, TailGuard incurs a delay between the measurement and control, which inevitably makes the achievable load to be lower than the maximum acceptable load. Nevertheless, these results demonstrated that the TailGuard query admission control can indeed provide tail latency SLO guarantee, while maintaining high resource utilization.

In our simulation, we assume all incoming queries are rejected when the system exceeds a defined threshold. However, in real systems, job rejection rules can be customized to achieve specific objectives, such as maximizing overall business profit. For instance, queries with lower class or lower price, or queries with larger fanout that demand more resources, are more likely to be rejected under these rules. In such cases, the upper bound threshold R^U_{cth} would need to be recalculated to align with the query control scenario.

E. TailGuard With Resource Auto-Scaling

Now we test the resource auto-scaling scheme, based on the upper and lower bound thresholds of task deadline miss ratio. We consider the same case as one for query admission control, except the system loads changing over time. The test runs for a total of 6 seconds. For the first 2 seconds, the system operates at 50% load. It then increases to 60% load for the next 2 seconds, and finally drops back to 50% load for the last 2 seconds.

Upper bound threshold $R^U_{cth} = 1.7\%$ used for query admission control is a bit too loose for auto-scaling, causing query tail latency SLO violation for a short period of time. In case of admission control, incoming queries are dropped, allowing task queue length to be reduced rapidly. But with resource auto-scaling, the queue length persists for a longer period as all queries are admitted. Consequently, we set a smaller upper bound threshold R^U_{sth} to be 1.5% . Additionally, we observed that the system operates at a load that is at least 20% below its maximum allowable load when the task deadline miss ratio is less than 0.5% . Therefore, we set 0.5% as the lower bound threshold.

In our approach, we implement resource vertical scaling by adjusting resources in a manner that allows the average task service time to decrease by 20% or increase by 25% per round of scaling. This strategy ensures that the system's resources remain consistent after completing one cycle of scaling up and scaling down. When the task deadline miss ratio hits the upper or lower bound threshold, the resource auto-scaling is triggered.

Fig. 9 presents the 99th percentile query tail latency and task deadline miss ratio at the different runtimes. When the system loads increases to 60% at second 2, the task deadline miss ratio quickly goes over the upper bound threshold, triggering a resource upscale. Despite the task queue remaining lengthy for some time after the resource upscale, the reduced upper bound threshold helps preventing a breach of the tail latency SLO. At second 4, the system load decreases, causing the task deadline miss ratio to fall below the lower bound threshold and prompting resource downscale. Due to inherent delay, the query tail latency experiences a temporary reduction before returning

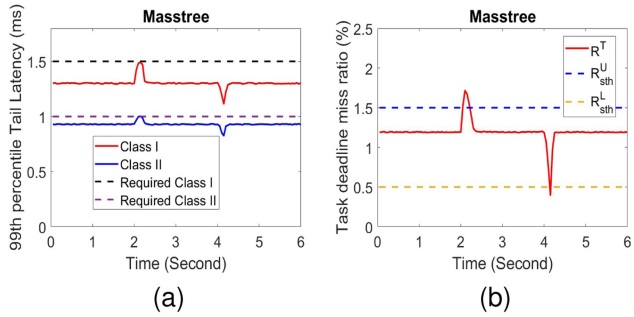


Fig. 9. TailGuard with auto-scaling scheme. (a) 99th percentile query tail latency (ms); and (b) Task deadline miss ratio (%), R^T, R^U_{sth} and R^L_{sth} are the task deadline miss ratio, upper and low bound thresholds of task deadline miss ratio, respectively.

to its previous level. The results indicates that the upper and lower bound thresholds for task deadline miss ratio are effective for resource auto-scaling.

F. Multiple Tail Latency SLOs and Mixed Fanouts

Now we study the performance of TailGuard on systems with multiple service classes of distinct query tail latency SLOs and mixed query fanouts. Consider the cluster of size $N = 1000$. The query fanout, k_f, is uniformly distributed from 1 to 1000, i.e., $P(k_f) = 1/1000$ for $k_f = 1,...,1000$. Due to limited space, we only present the results of the Masstree workload (similar results are obtained for the other two workloads). Consider four service classes, i.e., Classes 1, 2, 3 and 4, having requested 99th tail latency SLOs of 1.4, 1.8, 2.2 and 2.6 ms, respectively. Again, the tail latency SLOs are set such that the system runs at medium loads, while meeting the tail latency SLOs. With the given tail latency SLOs, the task pre-dequeuing time budgets for queries of different classes and fanouts can then be readily calculated from (4). Further assume that a query has equal probability to request any one of the four classes of service.

Although queries from any given class may have fanouts ranging from 1 to 1000, TailGuard can meet the tail latency SLO for all queries as long as tasks meet their pre-dequeuing time deadlines. However, to verify that this is the case, the simulation must track the maximum allowable load that meets the tail latency SLO for queries with fanout of 1000. This is because queries with the highest fanout have the tightest budget and hence, the lowest maximum allowable load that meets the tail latency SLO among all queries of the same class, as demonstrated in Section IV-B. Unfortunately, a practical concern is that it may take too long in terms of the sampling time to be effective. For example, consider the mean query arrival rate, $\lambda = 1000$ queries per second. With four classes and the uniform fanout distribution, on average, each class will see only one query with fanout of 1000 every 4 seconds. In order to get a reasonably reliable estimation of the 99th percentile tail latency, a sampling window of at least 10000 samples must be used, which however, corresponds to a time window of 40000 seconds! This is definitely too large to capture the impact of both system and workload variabilities on TailGuard performance, which may happen at a much smaller timescale, say, on the order of

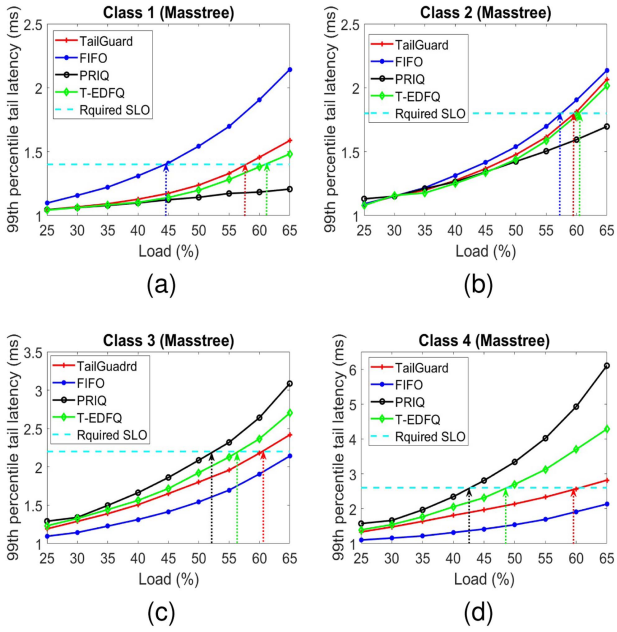


Fig. 10. Mixed fanout and multiple tail latency SLOs.

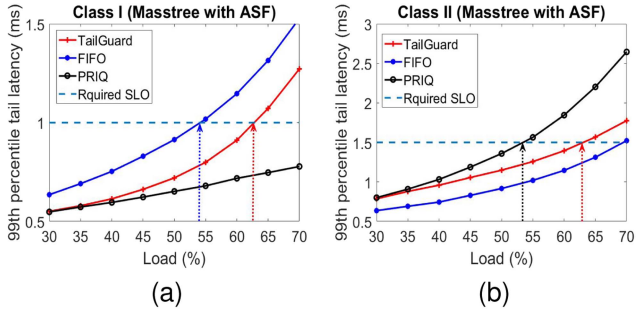


Fig. 11. Masstree workload with ASF scheme.

minutes. This example also demonstrates that a solution based on direct query tail-latency measurement like Cake [41] is not viable to enable per-query tail-latency SLOs.

Taking the above concerns into consideration, in this study, we only track the maximum load that meet the tail latency SLO for all the queries in each class as a whole, regardless of query fanout. Using the same example above, this allows 250 samples to be collected per second or a time window of 40 seconds, small enough to test the TailGuard performance, which however, is done at the per-class, rather than per-query granularity.

Fig. 10 presents the results for the Masstree workload. Similarly to the previous cases, compared to the other three schemes, the resources allocated by TailGuard are very well balanced, resulting in the maximum loads that meet the tail latency SLOs for all four classes within 4% of one another. Consequently, TailGuard achieves better overall performance than the other three. More specifically, maximum loads for TailGuard, FIFO, PRIQ and T-EDFQ are 57% , 45% , 43% and 48% , respectively. The performance gains of TailGuard over FIFO, PRIQ and T-EDFQ are about 27% , 33% and 19% , respectively.

G. Joint TailGuard and Outlier-Alleviation Solution

As we mentioned in Section III-B, TailGuard is orthogonal and complementary to most existing outlier-alleviation solutions. To demonstrate this is indeed the case, we test the performance of TailGuard, along with the Adaptive Slow-to-Fast task scheduling scheme (called ASF in this paper) based on DVFS [38]. In ASF, a task server starts to serve a task at a low power level and switches to a higher power level to shorten the task execution time if a task service time runs longer than a threshold. The goal of ASF is to meet the query tail latency SLO, while minimizing the power consumption. In our simulation, a task runs at a normal (low) power level until its service time reaches twice the mean task service time, when it switches to run at a higher power level so that the remaining task service time is reduced by half (i.e., the task service speed doubles). Clearly, TailGuard is orthogonal to ASF. From TailGuard's point of view, the only difference ASF makes is that the CDF of the task service time, $F(t)$, seen by TailGuard is changed to a new one with a shorter tail. Hence ASF helps TailGuard to achieve either a higher query throughput or tighter query tail latency SLOs. We use the same case given in Section IV.C to test it. Again, we only present the results for the Masstree workload due to space limitation.

Fig. 11 depicts the performance of TailGuard compared with FIFO and PRIQ. We can see that all three schemes can run at higher loads to support the required tail latency SLOs, thanks to ASF (see Fig. 6(a) and (b)). The performance gains of TailGuard over FIFO and PRIQ are almost the same as that without ASF. Namely, the maximum load with guaranteed tail latency SLO increases from about 54% , 44% and 48% to about 63% , 54% and 54% for TailGuard, FIFO and PRIQ, respectively. These results demonstrate that TailGuard may work with some of the outlier-alleviation solutions seamlessly to further improve the performance.

H. Evaluation in the Amazon EC2 Cloud

To verify the simulation results, we implement TailGuard in a small cluster in the Amazon EC2 cloud. The TailGuard implementation includes the query scheduling code and the Spark plug-in code. TailGuard is implemented by modifying the open source implementation codes for Eagle [52] and Pigeon [31]. The query scheduler is deployed at the application front-end, exposing services to allow the framework to submit query scheduling requests using remote procedure calls (RPCs). Apache Thrift [53] is used by all RPCs for internal communications between modules of a scheduler. Both the query handler and task servers are all hosted on m4.large instances.

The cluster is composed of 1 query handler and 100 task servers. Although the dedicated computing resources are used, the task non-service time t_x that accounts for the query processing delay, task dispatching delay, task result returning delay, and merging processing delay is much larger than the mean service times for all three workloads, which are in sub-milliseconds. Hence, to alleviate the domination of the task response time by t_x, for our experiment, we increase the mean task service

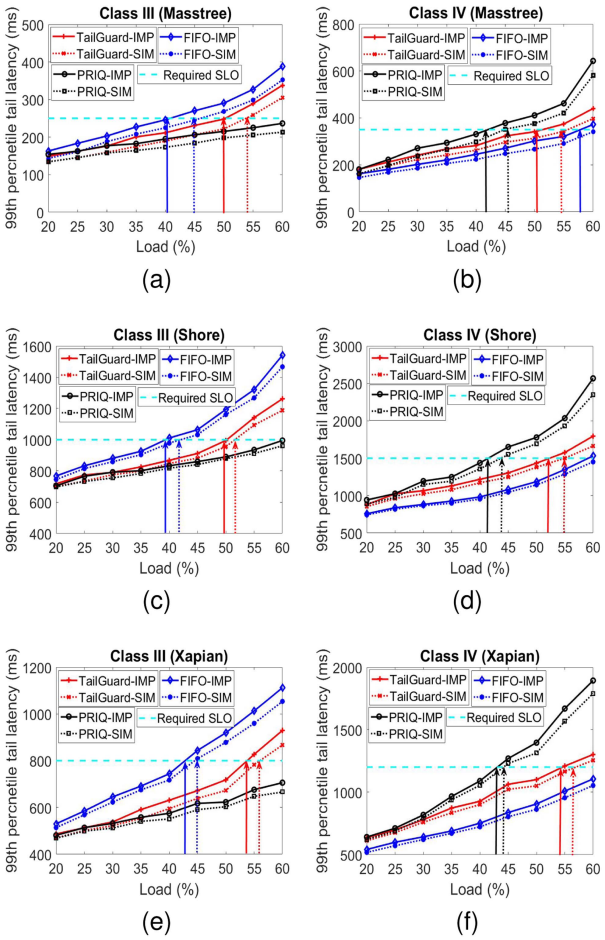


Fig. 12. Amazon EC2 implementation (IMP) vs simulation (SIM).

times by 300, 200 and 100 times, i.e., 52.8 ms, 68.3 ms and 92.5 ms for Masstree, Shore and Xapian, respectively. With this setup, the task service time is on the same order as the task non-service time. We introduce two service classes, i.e., Classes III and IV. The corresponding tail latency SLOs for Class III/IV are 250/350, 1000/1500 and 800/1200 ms for Masstree, Shore and Xapian, respectively. Similar to Case IV.C, all queries have the same fanout 100.

Fig. 12 presents the results for both experiments in EC2, denoted as IMP, and simulation, denoted as SIM. As expected, TailGuard achieves better performance than both FIFO and PRIQ for all the three workloads (again T-EDFQ behaves the same as TailGuard in this case). Clearly, the experiment results are consistent with the simulation results, with the differences within 10% . The difference increases as the load increases. This is because t_x, which is overlooked in simulation, becomes larger for the experiment as the load increases. The increased t_x value makes the tail latency for the experiment alway larger than that for the simulation. Nevertheless, we can see that the performance gains of TailGuard over FIFO and PRIQ for the experiment are almost the same as those for the simulation for all the three workloads. These results, to some extent, verify the effectiveness and accuracy of simulation results presented in the previous sections.

I. Evaluation in an SaS Testbed

Finally, we evaluate and compare TailGuard against the other three schemes in an on-campus SaS testbed being developed.

Testbed Setup: The testbed is currently composed of four clusters of edge nodes, located in four rooms in two buildings, including a server room and a Graduate Research Assistant (GRA) office next to a wet lab in one building, and a faculty office and a Graduate Teaching Assistant (GTA) office in another building. Each of these four clusters, referred to as Server-room, Wet-lab, Faculty and GTA clusters hereafter, consists of 8 Raspberry Pi devices, serving as edge nodes, with each currently attached with a temperature sensor and humidity sensor and connected to the Internet through an Ethernet switch. Each edge node receives sensing data from both sensors periodically and keeps up to eighteen-month-worth of the data records. Since the Wet-lab cluster may require low delay sensing data, we use the higher performing Raspberry Pi's to furnish the cluster than the ones for the other three and have the query handler co-located with the cluster to minimize the communication delay.

Use Cases: We consider three likely use cases belonging to three distinct classes, A, B, and C, to stress test TailGuard, with the 99th percentile tail latency SLOs equal to 800, 1300, and 1800 ms, respectively.

First, we note that the server room and wet lab are shared by many research groups and individuals, who may want to closely monitor individual devices they own to track the sensing data. This use case can stress test TailGuard by generating heavier workload on these two clusters than the other two. To create even more unbalanced load, instead of evenly distributing the load on these two clusters,[7] we place 80% of such workload on the Server-room cluster and the rest 20% randomly assigned to the others. Moreover, queries of this use case are considered class A with the most stringent tail latency SLO and constitute 50% of the total queries.

Second, we consider a use case targeting at potential users who may want to get an overall reading of the temperature and humidity in all areas with low delay. For such use case, a query fans out 4 tasks, each accessing a randomly selected edge node in a separate cluster. This use case is considered less time critical than the previous one and thus designated class B. We assume that it takes up 40% of the total queries.

Third, some users may require detailed, relatively longer term sensing data records to be retrieved from all edge nodes with a loose tail latency SLO. Hence, all the queries in this use case have fanout 32 and are assigned as class C, and 10% of the total queries are assigned to this class.

SaS testbed Architecture: Fig. 13 depicts the SaS testbed architecture. The query handler runs in a PC and consists of a query/task process module and an aggregator module. Queuing takes place centrally in the query/task process module with 32 sets of queuing buffers allocated, one for each edge node. The testbed resources are managed by K3s [55], which orchestrates the pod resource allocation in edge nodes. All the

[7]Note that equipped with the highest performing nodes and closest to the query handler, the Wet-lab cluster can hardly pose a performance bottleneck.

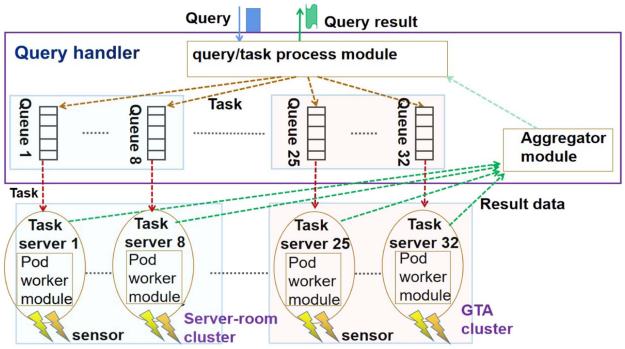


Fig. 13. SaS testbed architecture.

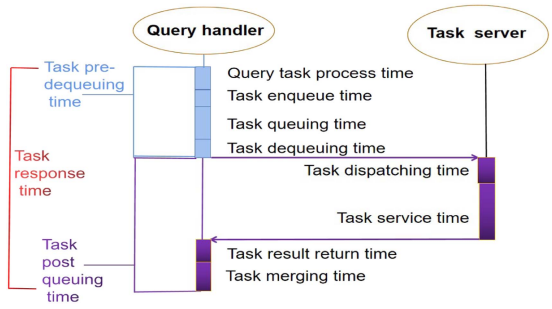


Fig. 14. Time sequence of scheduling tasks.

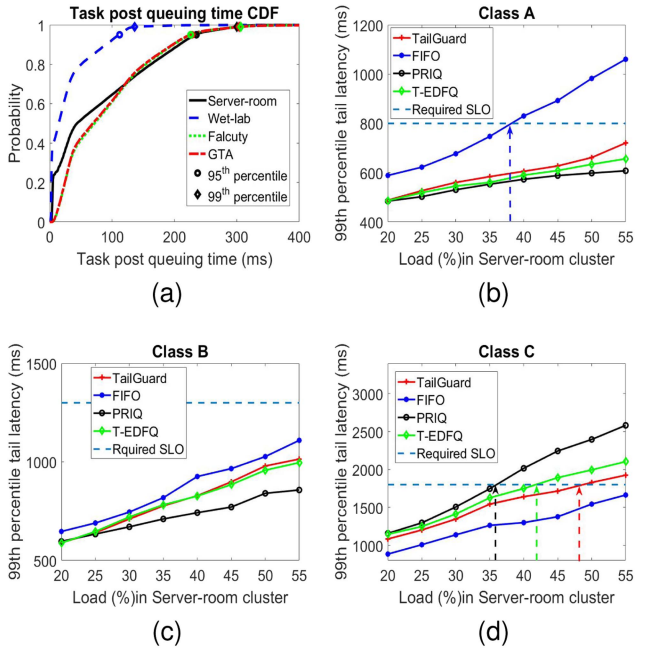


Fig. 15. (a) The task post-queuing time CDFs in four clusters. Circle and diamond represent the 95th and 99th percentile tail latencies, respectively. (b), (c) and (d) are the 99th percentile query tail latency of the three classes at various loads.

communications between the query handler and an edge node use keep-alive HTTP/1.1 connections.

When a query of a given class arrives, the query/task process module generates the right number of tasks including their intended edge nodes, and then sends the tasks to the corresponding edge nodes. A task arriving at an edge node retrieves one or multiple temperature and/or humidity records from the local database. It has an equal probability of retrieving one to up to thirty-day-worth of consecutive records starting from a random time in the eighteen-month period. After retrieving the records, the edge node sends the records to the aggregator module and an edge-node-idle message to the process module. Upon receiving the records for all the query tasks, the aggregator merges the records for the query, which are finally sent to the user. Fig. 14 gives the detailed breakdown of the time sequence per query processing. The task response time consists of task pre-dequeuing time and post-queuing time. The task pre-dequeuing time includes task process time, enqueue time, queuing time and dequeuing time, and the task post-queuing time is composed of task dispatching time, service time, result return and merging times.

To further test if TailGuard can perform well with inaccurate CDFs of unloaded task post-queuing times, we let all 8 edge nodes in each cluster share the same CDF based on the samples evenly collected from all edge nodes in the cluster. Fig. 15(a) presents the CDFs for all four clusters. First, we note that the CDFs (red and green lines) for Faculty and GTA clusters are almost identical, as they use the same model of Raspberry Pi's and located in the same building. With the same model of Raspberry Pi's but located in a different building and closer to the query handler, the CDF for the Sever-room cluster concentrates more in the lower post-queuing time region than the previous two. In contrast, equipped with the highest performing Raspberry Pi's and co-located with the query handler, the Wet-lab cluster offers significantly smaller overall post-queuing time than the other three. More specifically, The mean, 95th, and 99th task post queuing times are about 82/31/92/91 ms, 235/112/226/228 ms, and 300/136/306/304 ms for the Server-room/Wet-lab/Faculty/GTA clusters, respectively, making the system heterogeneous. With class A queries highly concentrated on the Server-room cluster, we create a highly heterogeneous scenario where the Server-room cluster is the most heavily loaded, whereas the Wet-lab cluster is highly under utilized. This is an ideal scenario to stress test TailGuard. The reason is that a query from any class that has a task using the Server-room cluster has a higher probability to be the slowest one and hence a high chance to determine the query response time. In this case, the query fanout impact on the query performance is much reduced, making TailGuard less effective with respective to the other three queuing policies, which are fanout agnostic.

Results and Analysis: Fig. 15(b), (c) and (d) present the results. We note that TailGuard, FIFO, PRIQ and T-EDFQ can achieve the maximum load of about 48% , 38% , 36% and 42% , respectively. This results in the performance gains of TailGuard over FIFO, PRIQ and T-EDFQ to be 26.3% , 33.3% and 14.3% , respectively. As one can see, both the performance gains and the maximum load differences in such a highly heterogeneous system are in line with the simulated ones (homogeneous systems).

The above stress test, together with the simulation, demonstrates that TailGuard is effective to improve resource allocation

performance for user-facing services, even in a heterogeneous system with highly unbalanced workload patterns, and varied processing and communication delays. As the testbed grows larger, one can expect that the performance gains of TailGuard over the other three fanout-agnostic schemes will further increase, because the average query fanout is likely to increase with the number of edge nodes in the testbed.

V. Conclusion

In this paper, we propose TailGuard for user-facing services, aiming at maximizing resource utilization, while providing tail latency SLO guarantee. TailGuard decouples the upper query level design from the lower task level design. First, at the query level, a decomposition technique is developed to compute the task pre-dequeuing time deadline for a query with the given tail latency SLO and fanout. Second, at the task level, based on the task pre-dequeuing time deadline, a simple EDFQ policy is employed to manage task queues to improve the resource utilization. A query admission control and a resource auto-scaling scheme are also developed based on the task deadline miss ratio. TailGuard is evaluated by simulation and implementation in Amazon EC2 Cloud using three Tailbench workloads as input. The results demonstrate that TailGuard can improve resource utilization by up to 80% while meeting tail latency SLOs, compared to the FIFO, PRIQ and T-EDFQ queuing policies. TailGuard is also implemented and tested in a heterogeneous SaS testbed and the test results agree with the simulated ones.

References

[1] "Storage: How tail latency impacts customer-facing applications," (n.d.), 2021. [Online]. Available: https://www.computerweekly.com/opinion/Storage-How-tail-latency-impacts-customer-facing-applications

[2] Y. Cheng, A. Anwar, and X. Duan, "Analyzing alibaba's co-located datacenter workloads," in *Proc. IEEE Int. Conf. Big Data*, 2018, pp. 292–297.

[3] J. Dean and L. Barroso, "The tail at scale," *Commun. ACM*, vol. v56, no. 12, pp. 74–80, 2013.

[4] S. Cho, A. Carter, J. Ehrlich, and J. Jan, "Moolle: Fan-out control for scalable distributed data stores," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 1206–1217.

[5] R. Nishtala et al., "Scaling memcache at Facebook," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.

[6] S. Rosenkrantz et al., "JADE: Tail-latency-SLO-aware job scheduling for sensing-as-a-service," in *Proc. IEEE/ACM 13th Int. Conf. Utility Cloud Comput.*, 2020, pp. 366–373.

[7] S. Lalith et al., "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 513–527.

[8] J. Li et al., "Tales of the tail: Hardware, OS, and application-level sources of tail latency," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.

[9] W. Reda et al., "Rein: Taming tail latency in key-value stores via multiget scheduling," in *Proc. Eur. Conf. Comput. Syst.*, 2017, pp. 95–110.

[10] "Order Statistic," (n.d.), 2023. [Online]. Available: https://en.wikipedia.org/wiki/Order_statistic

[11] M. Jeon et al., "Predictive parallelization: Taming tail latencies in web search," in *Proc. 37th Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2014, pp. 253–262.

[12] A. Vulimiri et al., "Low latency via redundancy," in *Proc. ACM Conf. Emerg. Netw. Experiments Technol.*, 2013, pp. 283–294.

[13] D. Lo et al., "Heracles: Improving resource efficiency at scale," in *Proc. ACM/IEEE Annu. Int. Symp. Comput. Architecture*, 2015, pp. 450–462.

[14] C. Delimitrou and K. Christos, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. v48, no. 4, pp. 77–88, 2013.

[15] H. Yang et al., "Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computersqq," *ACM SIGARCH Comput. Architecture News*, vol. v41, no. 3, pp. 127–144, 2020.

[16] M. Haque et al., "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," *ACM SIGPLAN Notices*, vol. v50, no. 4, pp. 161–175, 2015.

[17] Y. Xu et al., "Bobtail: Avoiding long tails in the cloud," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 329–342.

[18] C. Stewart, A. Chakrabarti, and R. Griffith, "Zoolander: Efficiently meeting very strict, low-latency SLOs," in *Proc. Int. Conf. Automat. Comput.*, 2013, pp. 265–277.

[19] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 185–198.

[20] M. Tirmazi et al., "Borg: The next generation," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, Art. no. 30.

[21] J. Guo et al., "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *Proc. Int. Symp. Qual. Service*, 2019, Art. no. 39.

[22] W. Chen, J. Rao, and X. Zhou, "Preemptive, low latency datacenter scheduling via lightweight virtualization," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 251–263.

[23] P. Misra et al., "Managing tail latency in datacenter-scale file systems under production constraints," in *Proc. Eur. Conf. Comput. Syst.*, 2019, pp. 17:1–17:15.

[24] Z. Wang, H. Li, L. Sun, T. Rosenkrantz, H. Che, and H. Jiang, "TailGuard: Tail latency SLO guaranteed task scheduling for data-intensive user-facing applications,," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2023, pp. 898–909.

[25] H. Kasture and D. Sanchez, "TailBench: A benchmark suite and evaluation methodology for latency-critical applications," in *Proc. IEEE Int. Symp. Workload Characterization*, 2016, pp. 1–10.

[26] C. Perera, A. Zaslavsky, and D. Georgakopoulos, "Sensing as a service model for smart cities supported by Internet of Things," *Wiley Trans. Emerg. Telecommun. Technol.*, vol. 25, pp. 81–93, 2014.

[27] A. Sriraman and T. Wenisch, "μtune: Auto-tuned threading for OLDI microservices," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2017, pp. 177–194.

[28] A. Sriraman and T. Wenisch, "μSuite: A benchmark suite for microservices," in *Proc. IEEE Int. Symp. Workload Characterization*, 2018, pp. 1–12.

[29] K. Rzadca et al., "Autopilot: Workload Autoscaling at Google," in *Proc. Eur. Conf. Comput. Syst.*, 2020, Art. no. 16.

[30] A. Mirhosseini et al., "Q-Zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 207–219.

[31] Z. Wang et al., "Pigeon: An effective distributed, hierarchical datacenter job scheduler," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 246–258.

[32] A. Mirhosseini and T. Wenisch, "μSteal: A theory-backed framework for preemptive work and resource stealing in mixed-criticality microservices," in *Proc. ACM Int. Conf. SuperComput.*, 2021, pp. 102–114.

[33] R. Nishtala et al., "Twig: Multi-agent task management for colocated latency-critical cloud services," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 167–179.

[34] I. Gog et al., "Firmanent: Fast, centralized cluster scheduling at scale," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 99–115.

[35] K. Suo et al., "Characterizing and optimizing hotspot parallel garbage collection on multicore systems," in *Proc. Eur. Conf. Comput. Syst.*, 2018, pp. 35:1–35:15.

[36] Y. He, S. Sameh, J. Larus, and C. Yan, "Zeta: Scheduling interactive services with partial execution," in *Proc. ACM Symp. Cloud Comput.*, 2012, Art. no. 12.

[37] B. Cai et al., "Less provisioning: A hybrid resource scaling engine for long-running services with tail latency guarantees," *IEEE Trans. Cloud Comput.*, vol. v10, 3, pp. 1941–1957, Third Quarter 2022.

[38] M. Haque et al., "Exploiting heterogeneity for tail latency and energy efficiency," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2017, pp. 625–638.

[39] X. Chen et al., "Achieving low tail-latency and high scalability for serializable transactions in edge computing," in *Proc. Eur. Conf. Comput. Syst.*, 2021, pp. 210–227.

[40] Z. Zhang et al., "CRISP: Critical path analysis of large-scale microservice architectures," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2022, pp. 655–672.

[41] A. Wang et al., "Cake: Enabling High-level slos on shared storage systems," in *Proc. ACM Symp. Cloud Comput.*, 2012, Art. no. 14.
[42] T. Zhu et al., "PriorityMeister: Tail latency QoS for shared networked storage," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 29:1–29:14.
[43] T. Zhu, D. Berger, and M. Harchol-Balter, "SNC-meister: Admitting more tenants with tail latency SLOs," in *Proc. ACM Symp. Cloud Comput.*, 2016, pp. 374–387.
[44] T. Zhu, D. Berger, and M. Harchol-Balter, "WorloadCompactor: Reducing datacenter cost while providing tail latency SLO guarantees," in *Proc. ACM Symp. Cloud Comput.*, 2017, pp. 598–610.
[45] N. Li et al., "PSLO: Enforcing the Xth percentile latency and throughput SLOs for consolidated VM storage," in *Proc. Eur. Conf. Comput. Syst.*, 2016, Art. no. 28.
[46] R. Kannan et al., "Grandslam: Guaranteeing SLAs for jobs in microservices execution frameworks," in *Proc. Eur. Conf. Comput. Syst.*, 2019, pp. 34:1–34:16.
[47] Y. Zhang et al., "Sinan: ML-based and QoS-aware resource management for cloud microservices," in *Proc. Int. Conf. Architectural Support Prog.. Lang.s Operating Syst.*, 2021, pp. 167–181.
[48] J. Yin, X. Lu, H. Chen, and X. Liu, "BURSE: A bursty and self-similar workload generator for cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. v26, no. 3, pp. 668–680, Mar. 2015.
[49] S. E. Kafhali and K. Salah, "Stochastic modeling and analysis of cloud computing data center," in *Proc. Conf. Innov.s Clouds Internet Netw.*, 2017, pp. 122–126.
[50] "Pareto distribution," [Online]. Available: https://en.wikipedia.org/wiki/Pareto_distribution
[51] A. G. Fayoumi, "Performance evaluation of a cloud based load balancer server serving Pareto traffic," *J. Theor. Applicaed Inf. Technol.*, vol. v32, no. 1, pp. 28–34, 2011.
[52] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware scheduling in eagle: Divide and stick to your probes," in *Proc. ACM Symp. Cloud Comput.*, 2016, pp. 497–509.
[53] "Apache thrift," 2017. [Online]. Available: https://thrift.apache.org/
[54] K. Ousterhout et al., "Sparrow: Distributed, low latency scheduling," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 69–84.
[55] "Kubernetes (K3s)," (n.d.), 2024. [Online]. Available: https://k3s.io/

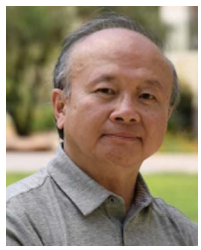
Zhijun Wang received the PhD degree in computer science from the University of Texas at Arlington (UTA), and the MS degree in electrical engineering from the Pennsylvania State University. He is now an associate professor in research with the ACES Lab, UTA. His research interests include task scheduling, resource management, performance analysis and traffic control in cloud and edge computing, web cache, and network processor design.

Huiyang Li received the BS degree in electronic information engineering from the TianJin Collage, University of Science & Technology, Beijing, in 2014, and the MS and PhD degrees in computer science from the University of Texas at Arlington, in 2017 and 2023, respectively. Now, she is working as a software engineer with Uber Technology Inc. Her research interests include the SLO Tail latency guaranteed data center job scheduling and resource management in cloud and edge computing.

Lin Sun received the BS degree in computer science from Shandong University, and the MS degree in computer science from Fudan University. He is currently working toward the PhD degree with the Computer Science and Engineering Department, University of Texas at Arlington. His research interests include focused on latency-SLO-guaranteed task scheduling in cloud data centers, and on the edge.

Stoddard Rosenkrantz received the BS degree from Texas Wesleyan University, and the PhD degree from the Computer Science and Engineering Department, University of Texas at Arlington, in 2024. His research interests include focused on performance measurement and prediction for constrained edge devices. Previously, he was employed by IBM, McAfee, and Dell Inc. in various technical and consultative roles.

Hao Che received the BS degree from Nanjing University, in 1984, the MS degree in physics from the University of Texas at Arlington, in 1994, and the PhD degree in electrical engineering from the University of Texas at Austin, in 1998. He was an assistant professor in electrical engineering with the Pennsylvania State University from 1998 to 2000, and a System Architect with Santera Systems, Inc., from 2000 to 2002. Since September 2002, he has been with the Department of Computer Science and Engineering, University of Texas at Arlington. His current research interests include network architecture and network resource management, performance analysis of large-scale distributed computing systems, including many-core processors, warehouse-scale computing, cloud computing, edge computing, and IoTs.



Hong Jiang (Fellow, IEEE) received the BSc degree in computer engineering from the Huazhong University of Science and Technology, the MASc degree in computer engineering from the University of Toronto, and the PhD degree in computer science from the Texas A&M University. He is currently chair and Wendell H. Nedderman Endowed professor with the Computer Science and Engineering Department, University of Texas at Arlington. Prior to joining UTA, he served as a Program director with National Science Foundation (2013.1-2015.8) and he was at University of Nebraska-Lincoln since 1991, where he was Willa Cather professor of Computer Science and Engineering. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, Big Data computing, cloud computing, and performance evaluation.