TailGuard: Tail Latency SLO Guaranteed Task Scheduling for Data-Intensive User-Facing Applications

Zhijun Wang
The University of Texas at Arlington
zhijun.wang@uta.edu

Huiyang Li
The University of Texas at Arlington
huiyang.li@mavs.uta.edu

Lin Sun
University of Texas at Arlington
lxs5171@mavs.uta.edu

Todd Rosenkrantz
The University of Texas at Arlington
todd.rosenkrantz@mavs.uta.edu

Hao Che
The University of Texas at Arlington
hche@cse.uta.edu

Hong Jiang
The University of Texas at Arlington
hong.jiang@uta.edu

***Abstract*—A primary design objective for Data-intensive User-facing (DU) 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 queuing 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. 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 simulation. It is driven by three types of applications in the Tailbench benchmark suite. 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. TailGuard is also implemented and tested in a highly heterogeneous Sensing-as-a-Service (SaS) testbed for a data sensing service, with test results in line with the other ones.**

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

I. Introduction

It has been widely recognized that the query tail latency for Data-intensive User-facing (DU) services, such as web searching, online social networking, 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 DU services are generally over-provisioned [2], at the cost of reduced profit. As a result, a key design objective of a DU 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*.

This work was supported the US NSF under Grants CNS CSR-2008835 and CCF-SHF2226117.

However, achieving the above design objective is by no means easy. A query for a typical DU 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 crowdsening [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]–[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

DOI 10.1109/ICDCS57875.2023.00042

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 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.

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, e.g., by being allocated different amounts of resource to closely match their resource demands so that all the queries can meet the tail latency SLO at the lowest possible resource consumption. 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 DU services in today's datacenters is normally through resource overprovisioning [2]. Our simulation results (see Section IV.B for details) indicate that by taking fanout into account, TailGuard can improve resource utilization by 80% compared to the First-In-First-Out (FIFO) queuing policy, while meeting a stringent query tail latency SLO for DU workloads.

Second, consider a DU 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], [19], [20]), task fanout-unaware queue management policies (e.g., the Tail-latency-SLO-aware Earliest-Deadline-First Queuing (T-EDFQ)), or task preemption [21] policies inadequate to achieve the design objective. This may also render some task reordering solutions solely based on task sizes [8], [22] inadequate. Our simulation results (see details in Section IV.B) demonstrate that TailGuard can improve overall resource utilization by 40% over the PRIQ policy and 22% over T-EDFQ in supporting two classes of tail latency SLOs for DU workloads.

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 DU 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 queuing 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 queuing deadlines, as a way to differentiate resource allocation for tasks with different resource demands. In principle, TailGuard permits unlimited number of query classes and is lightweight, as it incurs minimum overhead for task queuing deadline estimation and requires to implement only a single earliest-deadline-first queue per task server for any DU applications. A query admission control scheme is also developed to provide tail latency SLO guarantee in the face of resource shortages.

TailGuard, or equivalently, TF-EDFQ, is evaluated against FIFO, PRIQ and T-EDFQ (Section III.A gives their exact definitions) by simulation. Three traces generated from the Tailbench benchmark suite [23] 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-as-a-Service (SaS) testbed for an edge-based temperature-and-humidity sensing service, with test results in lines with the other ones.

The remainder of this paper is organized as follows. Section II presents the background and related work. Section III introduces TailGuard. Performance evaluation is given in Section IV. Finally Section V concludes the paper.

II. Background and Related Work

A. Data-Intensive User-Facing Services

DU 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[1] [24]. Predominant DU 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 (e.g., the total amount of crowdsensing data in the case of an SaS) in this class are distributed to a large number of task servers/edge nodes. Accordingly, a query may spawn a number of tasks to be dispatched to some or all of these task servers/edge nodes to be processed. A notable subclass of such services is OnLine Data-intensive (OLDI) services [25]. A query for an OLDI service needs to touch upon every part of the working dataset, i.e., the query fanout for each query is equal to the total number of servers involved (ranging from a

[1]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.

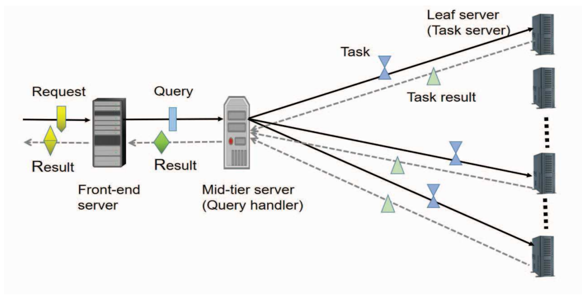


Fig. 1. A typical DU application process architecture

few to tens of thousands). Large online search products, online advertising and online machine translation, are examples of OLDI services. For other DU services, different queries may need to touch upon different parts of the working dataset. A notable example of such a service is social networking services, such as Facebook and LinkedIn. For instance, the fanout for a typical Facebook page query is in the range of one to several hundreds with 65% under 20 [5]. Other examples are emergency response SaSes, e.g., finding a missing person through surveillance cameras and fire detection and alert via crowd temperature sensing. A query of such a service is expected to have a fanout anywhere between one to a few millions depending on the scope of sensing.

A DU 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 stakeholders, including individuals who own the sensing data and/or edge devices and cloud service providers.

Figure 1 depicts a generic DU application processing model [25], [26]. 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[2]), 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 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[3] 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], [19], [20], task-reordering-based queuing [8], [22], or EDFQ, unless task-aware resource auto-scaling [27] 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 DU Services

Many works have been devoted to addressing query tail latency related issues for DU 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], [22], [28], [29] 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]–[15], [30], [38] 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] are developed to reduce the task tail latency by allowing a task to be issued to multiple task server replicas. Task execution time prediction through workload profiling [9], [11], [16], [17], [25] and machine learning [31], [32] 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], [33] 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., [34], 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], [35] are proposed to help reduce query tail latencies. CPU power control schemes [18], [36] 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 [37] uses transaction timestamps to reduce both mean and tail latencies for edge computing. All

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

[3]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 queuing 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

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 [39], PriorityMeister [40], SNC-Meister [41], WorkloadCompactor [42] and PSLO [43], 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 [44] and Sinan [45] 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 decomposition, or equivalently, task queuing deadline estimation 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.

A. *TailGuard Query Processing Model*

Consider a query processing model directly derived from Figure 1, as depicted in Figure 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

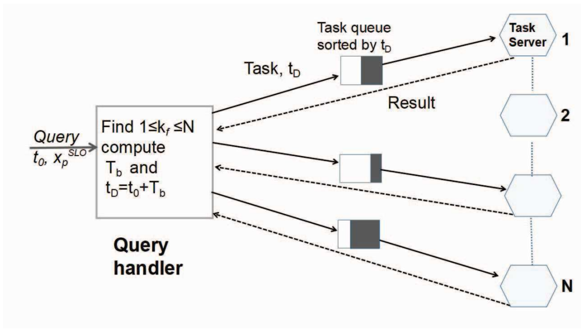


Fig. 2. TailGuard query processing model. A task queue for a task server can be set in the task server or in the query handler.

servers these tasks need to be dispatched. The query handler estimates task pre-dequeuing time budget T_b and computes the task queuing deadline $t_D = t_0 + T_b$, shared by all the tasks associated with the query [4]. Here t_D is defined as the deadline when the task must be dequeued and given to the corresponding task server to be processed in order to meet the tail latency SLO for the query. As we shall show in the next subsection, T_b (or t_D) is a function of both query tail latency SLO in terms of the pth percentile query latency of x_p^{SLO} and query fanout, k_f, i.e., $T_b = T_b(x_p^{SLO}, k_f)$ and $t_D = t_D(x_p^{SLO}, k_f)$. 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.

Finally, as mentioned in Section I, 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

[4]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.

the same way as TailGuard except that $t_D = t_0 + x_p^{SLO}$. In other words, the queuing deadline for a task is dependent on the corresponding query tail latency SLO, x_p^{SLO}, but independent of query fanout, k_f. 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.

B. Task Queuing Deadline Estimation

The key to the design of TailGuard is the task queuing deadline estimation or task decomposition. In this subsection, we first present the task queuing deadline estimation solution and then propose a way to implement it.

1) Solution: The task queuing 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.

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 the unloaded task response time t_{po}, $F_l^u(t)$, with respect to task server, l, can be measured and updated (see Section III.B.2 for details) for all task servers $l = 1, ..., N$. Furthermore, let $x_p^u(k_f)$ and $F_Q^u(t, k_f)$ represent the pth percentile unloaded query tail latency for a query with fanout k_f and the CDF of unloaded query latency, respectively. Here, a query latency is considered as unloaded (loaded) if the query response time does not (does) include pre-dequeuing delay, t_{pr}. Also define $n = n(k)$ to be the mapping between the k-th task in a query and the n-th task server the task is dispatched to, for $k = 1, ..., k_f$. 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], we have,

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

By definition, we have,

$$x_p^u(k_f) = F_Q^{u^{-1}}(\frac{p}{100}), \qquad (2)$$

where $F_Q^{u^{-1}}(.)$ is the inverse function of $F_Q^u(.)$.

Assuming that all the tasks in a query experience the same pre-dequeuing delay t_{pr}, we can express the CDF of the response time for task l, $F_l(t)$, as follows,

$$F_l(t) = \begin{cases} F_l^u(t - t_{pr}), & if\ t \geq t_{pr} \\ 0, & otherwise. \end{cases} \qquad (3)$$

Hence

$$F_Q(t, k_f) = \prod_{k=1}^{k_f} F_{n(k)}(t) = \begin{cases} F_Q^u(t - t_{pr}, k_f), & if\ \ t \geq t_{pr} \\ 0, & otherwise, \end{cases}$$

and

$$x_p(k_f) - t_{pr} = F_Q^{u^{-1}}(\frac{p}{100}). \qquad (4)$$

From Eqns. (2) and (4), we have,

$$x_p(k_f) = x_p^u(k_f) + t_{pr}. \qquad (5)$$

This result means that with any given query tail latency SLO, x_p^{SLO}, as long as, $t_{pr} \leq x_p^{SLO} - x_p^u(k_f)$, the query tail latency SLO is guaranteed to be met, i.e., $x_p(k_f) = x_p^u(k_f) + t_{pr} \leq x_p^{SLO}$. 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 queuing deadline can be defined as,

$$t_D = t_0 + T_b(x_p^{SLO}, k_f) = t_0 + x_p^{SLO} - x_p^u(k_f). \qquad (6)$$

In other words, for a query arrived at $t = t_0$, as shown in Figure 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[5], 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: Here we present preliminary ideas on how to extend the above task decomposition technique for queries to a task decomposition technique for requests that account for query dependencies.

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

[5]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.

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 Eqs. (4) and (5), we have,

$$x_p^R = x_p^{Ru} + t_{pr}^R = x_p^{Ru} + \sum_{i=1}^{M} t_{pr,i}, \tag{7}$$

where x_p^R and x_p^{Ru} are the loaded and unloaded pth percentile tail latency of the request. Eq. (7) 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 $i = 1, ..., M$.

Note that as long as T_b^R (i.e., $t_{pr}^R \leq T_b^R$) 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.

2) Implementation: The above task queuing deadline estimation solution 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, DU 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.

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 DU 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 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., Eq. (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 Eq. (6) for each query. The latter requires the management of a single EDFQ. As a result, TailGuard is a lightweight solution.

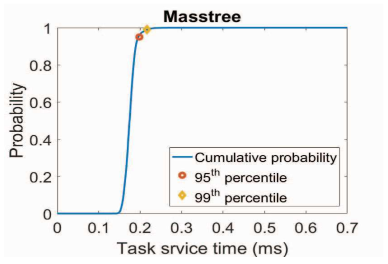
C. Query admission control

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 (e.g., edge resources may be quite limited to allow an SaS to scale).

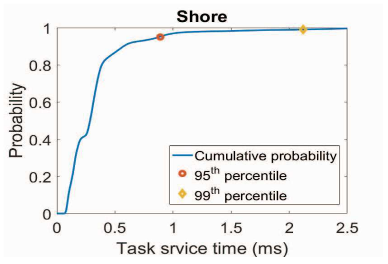
We tested TailGuard using various workloads and found that the query tail latency SLOs can still be met, when a small portion (less than 2% in our tests) of tasks miss their deadlines, confirming the aforementioned observation. With this understanding, TailGuard sets a threshold for the percentage of tasks missing their deadlines, R_{th}, for query 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 sever. The query handler can update the task deadline violation ratio in a given moving time window. When the ratio exceeds R_{th}, upcoming queries are rejected, till the ratio falls back below R_{th} 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.

IV. Performance Evaluation

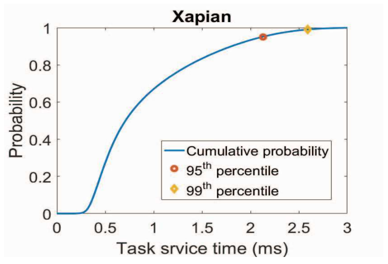
To cover a wide range of applications, TailGuard is firstly evaluated based on simulation using the workload statistics



(a)



(b)



(c)

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

for three datacenter applications available in Tailbench [23] as input. We first characterize the workload and then present the simulation results along the fanout and service class dimensions; and then with query admission control. Finally we verify TailGuard in a highly heterogeneous SaS testbed.

A. Workloads

For simulation, a DU 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 DU services in cloud are available, e.g., those made available by Google [19], [27] and Alibaba [2], [20], 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 [46] has been widely recognized as a good model for cloud applications in general [25], 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 [47], is also used in one simulation case.

Second, although a few publications do offer fanout distribution, $P(k_f)$, for k_f=1,...,N, for the DU 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 DU services in general, TailGuard should be tested against DU services with a wide range of task service time distributions. To this end, we resort to Tailbench [23] to gain access to applications with a wide range of task service time distributions. Tailbench provides eight DU 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 $F(t)$. 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.

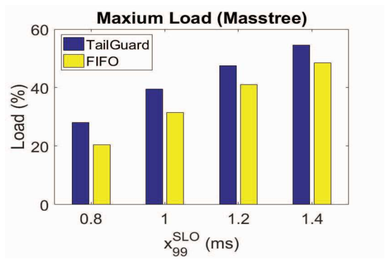
Figure 3 depicts the CDFs and the unloaded 95/99th percentile task tail latencies for the three workloads. Table II also gives the related statistics, including the mean task service time (T_m) and the unloaded 99th percentile query tail latency at fanouts k_f=1, 10 and 100, derived from Eqs. (1) and (2).

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

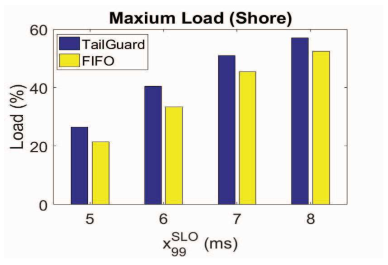
Bench	T_m	$x^u_{99}(1)$	$x^u_{99}(10)$	$x^u_{99}(100)$
Masstree	0.176	0.219	0.247	0.473
Shore	0.341	2.095	2.721	2.829
Xapian	0.925	2.590	2.998	3.308

B. Impact of query fanout

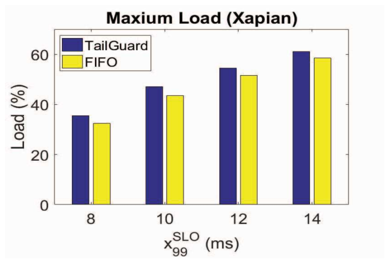
In this subsection, we focus on testing the impact of the query fanout. We present two cases, i.e., a single class case and a two-class case. Consider a cluster of size N=100 and three different types of queries corresponding to three different fanouts 1, 10 and 100, similar to the testing scenario in [48], 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^{SLO}_{99},



(a)



(b)



(c)

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

the task pre-dequeuing time budget for a query with fanout k_f (1, 10 or 100) is T_b=$x_{99}^{SLO} - x_{99}^{u}(k_f)$.

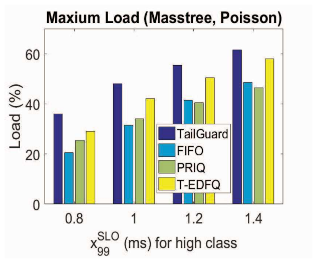
Note that meeting the tail latency SLO for queries as a whole does not guarantee that queries of individual types can meet the tail latency SLO. Hence, in the following simulation, we measure the tail latency for each type of queries and identify the maximum load at which all three types of queries meet 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. Figure 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 DU applications [19], [20]). 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. The performance gain increases as the tail latency SLO becomes tighter. This is because a query with a higher fanout has a tighter task queuing deadline and hence, higher chance to violate the tail latency SLO. Therefore, TailGuard that reorders the tasks based on queuing 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 $x_{99}^{SLO} = 0.8ms$, 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.

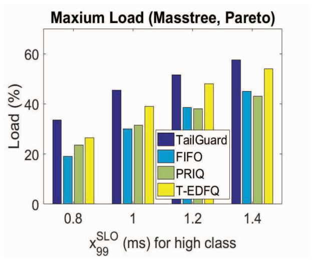
To gain more insights, for Masstree, Table III gives the breakdowns of the tail latencies at the maximum loads for the three types of queries. First, we note that at the maximum loads, the query type with k_f=100 barely meets the tail latency SLOs for both schemes. In other words, the maximum achievable 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.

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



(a)



(b)

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. Due to limited space, only the results for the Masstree workload are given (the results for the other two workloads are similar).

Figure 5 shows the maximum loads under which all queries can meet their tail latency SLOs. From the results (Figure 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 (Figure 5(b)) are similar to those with the Poisson arrival process. Meanwhile, the maximum loads decease about 2% to 6% 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. Impact of service class

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 OLDI services. 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 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 Eq. (6), we arrive at the task pre-dequeuing time budgets. For example, for Masstree, the budgets for classes I and II are 1-0.473=0.527ms and 1.5-0.473=1.027ms, 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.

Figure 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.,

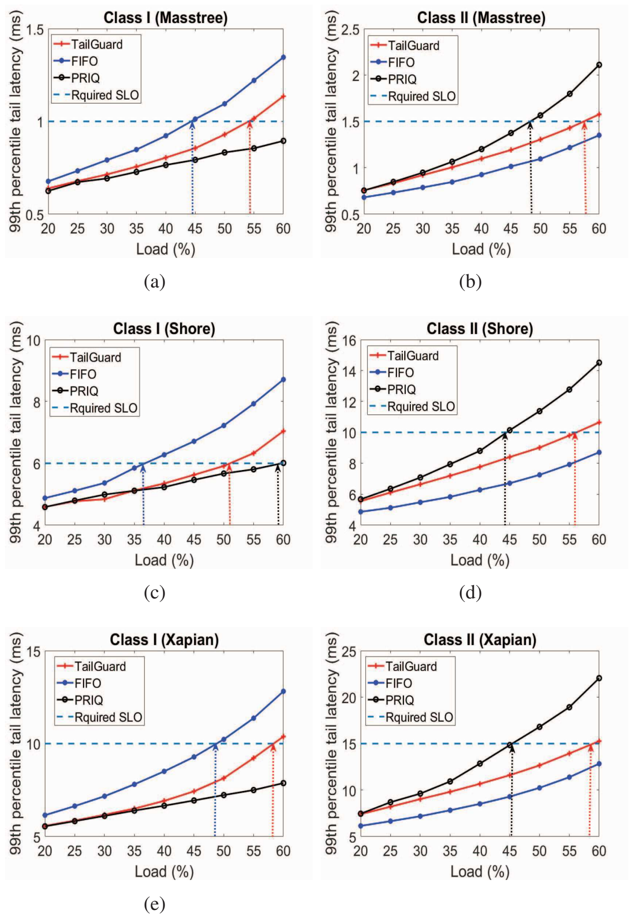
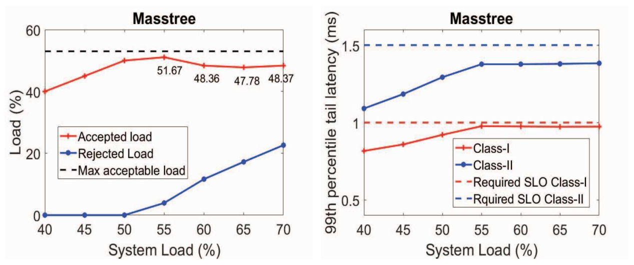


Fig. 6. The 99th percentile latency at different loads. The cyan line indicates the required tail latency and the arrows points to the maximum load that the tail latency can be met.

0.527ms and 1.027ms, 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 Figure 6(a), and higher than 60% for class II, as shown in Figure 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 Figure 6(c)) and 49% for Xapian (see Figure 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, re-



(a) (b)

Fig. 7. TailGuard with query admission control. (a) is the accepted/rejected load; and (b) is the query tail latency for Class I and II.

sulting in much closer maximum loads for the two classes (i.e., within 5% difference) for all three workloads. As shown in Figure 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.

D. TailGuard with Query Admission Control

Now we test the TailGuard query admission control scheme. 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_{th} 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.

Figure 7 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 queuing 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.

Finally, we note that the simulation results for cluster size, N=1,000, and in the presence of 4 classes are also available and consistent with the ones above, which however, are not presented here for the lack of space.

E. 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[6], 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.

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

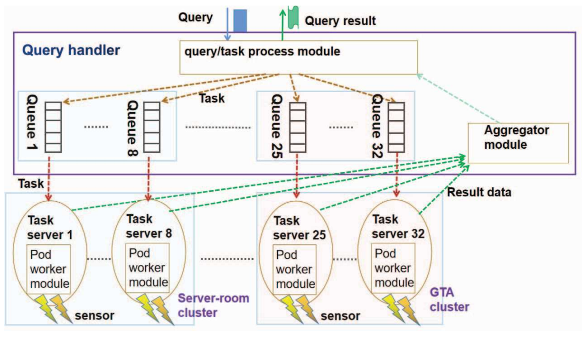


Fig. 8. SaS testbed architecture.

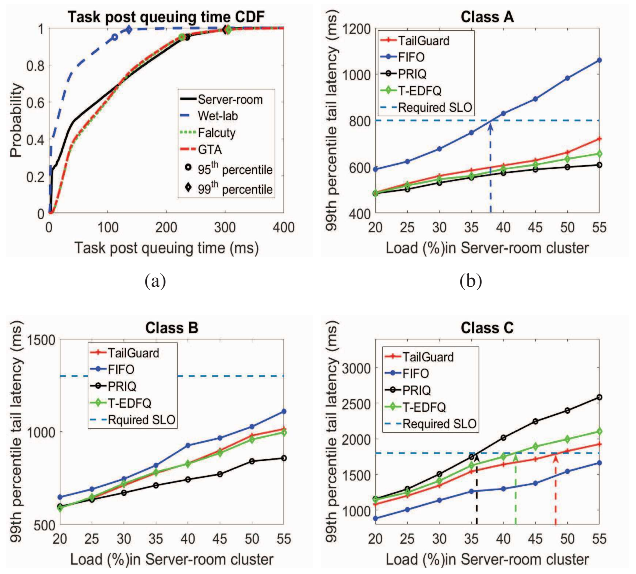


Fig. 9. (a) The task post-queuing time CDFs in four clusters. Circle and diamond represent the 95th and 99th percentile tail latencies, respectively. (b), (c) and (d) are the 99th percentile query tail latency of the three classes at various loads.

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: Figure 8 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 [49], which orchestrates the pod resource allocation in edge nodes. All the communications between the query handler and an edge node use keep-alive HTTP/1.1 connections.

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.

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. Figure 9 (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: Figures 9 (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 DU applications, 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.

All simulation source codes can be found on https://github.com/zjwang68/Tailguard.

V. Conclusions

In this paper, we propose TailGuard for data-intensive user-facing applications, 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 queuing deadline for a query with the given tail latency SLO and fanout. Second, at the task level, based on the task queuing deadline, a simple earliest-deadline-first queuing policy is employed to manage task queues to improve the resource utilization. TailGuard is evaluated by simulation 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," https://www.computerweekly.com/opinion/Storage-How-tail-latency-impacts-customer-facing-applications.
[2] Y. Cheng and A. Anwar and X. Duan, "Analyzing Alibaba's Co-located Datacenter Workloads," Proceedings of IEEE BIGDATA, 2018.
[3] J. Dean and L. Barroso, "The Tail at Scale," Communications of the ACM, v56(12), 2013.
[4] S. Cho, A. Carter, J. Ehrlich, and J. Jan, "Moolle: Fan-out Control for Scalable Distributed Data Stores,"Proceedings of IEEE ICDE, 2016.
[5] R. Nishtala, et al., "Scaling Memcache at Facebook," Proceedings of USENIX NSDI, 2013.
[6] S. Rosenkrantz et al., "JADE: Tail-Latency-SLO-Aware Job Scheduling for Sensing-as-a-Service," Proceedings of CloudAM, 2020.
[7] S. Lalith at al.,"C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection," Proceeding of NSDI, 2015.
[8] J. Li at al., "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency," Proceedings of ACM SoCC, 2014.
[9] W. Reda et al., "Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling," Proceedings of ACM Eurosys, 2017.
[10] "Order Statistic," https://en.wikipedia.org/wiki/Order_statistic.
[11] M. Jeon et al., "Predictive Parallelization: Taming Tail Latencies in Web Search," Proceedings of ACM SIGIR, 2014.
[12] A. Vulimiri et al., "Low Latency via Redundancy", Proceedings of ACM CoNEXT, 2013
[13] D. Lo, David at. al., "Heracles: Improving Resource Efficiency at Scale," Proceedings of ACM ISCA, 2015.
[14] C. Delimitrou and K. Christos, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters," ACM SIGPLAN Notices, v48(4), 2013.
[15] H. Yang at al., "Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," ACM SIGARCH Computer Architecture News, v41(3), 127-144, 2020.
[16] M. Haque at al., "Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services," ACM SIGPLAN Notices, v50(4), 2015.
[17] Y. Xu et al., "Bobtail: Avoiding Long Tails in the Cloud," Poceedings of the USENIX NSDI, 2013.
[18] C. Stewart, A. Chakrabarti and R. Griffith, "Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs," Proceedings of ICAC, 2013.
[19] M. Tirmazi et al., "Borg: the Next Generation," Proceedings of ACM Eurosys, 2020.
[20] J. Guo et al., "Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces,"Proceedings of IWQoS, 2019.
[21] W. Chen, J. Rao and X. Zhou, "Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization," Proceedings of ATC, 2017.
[22] P. Misra, et al., "Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints,"Poceedings of Eurosys, 2019.
[23] H. Kasture and D. Sanchez, "TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications," Proceedings of IEEE IISWA, 2016.
[24] C. Perera, A. Zaslavsky, and D. Georgakopoulos, "Sensing as a service model for smart cities supported by Internet of Things," Wiley Transactions on Emerging Telecommunications Technologies, 2013.
[25] A. Sriraman and T. Wenisch, "μtune: Auto-tuned threading for OLDI microservices, Proceedings of ISCA, 2017.
[26] A. Sriraman and T. Wenisch, "μSuite: A Benchmark Suite for Microservices," Proceedings of IISWA, 2018.
[27] K. Rzadca et al., "Autopilot: Workload Autoscaling at Google,"Proceedings of Eurosys, 2020.
[28] A. Mirhosseini et al., "Q-Zilla: A Scheduling Framework and Core Microarchitecture for Tail-Tolerant Microservices," Proceedings of HPCA, 2020.
[29] Z. Wang, H. Li, Z. Li, X. Sun, J. Rao, H. Che, Hao and H. Jiang, "Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler," Proceedings of the ACM Symposium on Cloud Computing (SOCC), 2019.
[30] A. Mirhosseini and T. Wenisch, "μSteal: a Theory-backed Framework for Preemptive Work and Resource Stealing in Mixed-criticality Microservices," Proceedings of ICS, 2021.
[31] R. Nishtala et al., "Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services," Proceedings of HPCA, 2020.
[32] I. Gog et al., "Firmanent: Fast, Centralized Cluster Scheduling at Scale," Proceedings of OSDI, 2016.
[33] K. Suo et al., "Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems," Proceedings of EuroSys, 2018.
[34] Y. He, S. Sameh, J. Larus and C. Yan, " Zeta: Scheduling Interactive Services with Partial Execution," Proceedings of ACM Symposium on Cloud Computing (SoCC), 2012.
[35] B. Cai at al., "Less Provisioning: A Hybrid Resource Scaling Engine for Long-running Services with Tail Latency Guarantees," IEEE Transactions on Cloud Computing, v10(3), pp1941-1957, 2020.
[36] M. Haque et al., "Exploiting Heterogeneity for Tail Latency and Energy Efficiency," Proceedings of MICRO, 2017.
[37] X. Chen et al., "Achieving Low Tail-latency and High Scalability for Serializable Transactions in Edge Computing," Proceedings of Eurosys, 2021.
[38] Z. Zhang et al., "CRISP: Critical Path Analysis of Large-Scale Microservice Architectures," Proceeding of ATC,2022.
[39] A. Wang et al., "Cake: Enabling High-level SLOs on Shared Storage Systems," Proceedings of SoCC, 2012.
[40] T. Zhu et al., "PriorityMeister: Tail Latency QoS for Shared Networked Storage," Proceedings of SoCC, 2014.
[41] T. Zhu, D. Berger and M. Harchol-Balter,"SNC-Meister: Admitting More Tenants with Tail Latency SLOs, Proceedings of SoCC, 2016.
[42] T. Zhu, D. Berger and M. Harchol-Balter,"WorloadCompactor: Reducing Datacenter Cost While Providing Tail Latency SLO Guarantees," Proceedings of SoCC, 2017.
[43] N. Li et al., "PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage," Proceeding of EuroSys, 2016.
[44] R. Kannan et al., "Grandslam: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks," Proceedings of EuroSys, 2019.
[45] Y. Zhang et al., "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," Proceeding of ASPLOS, 2021.
[46] "Poisson Distribution," https://en.wikipedia.org/wiki/Poisson-distribution.
[47] "Pareto Distribution," https://en.wikipedia.org/wiki/Pareto_distribution.
[48] K. Ousterhout et al., "Sparrow: Distributed, Low Latency Scheduling," Proceedings of SOSP, 2013.
[49] "Kubernetes (K3s)," https://k3s.io/.