Pyxis: Scheduling Mixed Tasks in Disaggregated Datacenters

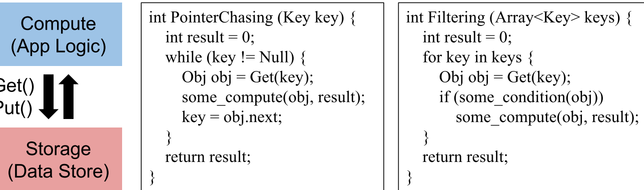
Sheng Qi, Chao Jin, Mosharaf Chowdhury, *Member, IEEE*, Zhenming Liu, Xuanzhe Liu, *Senior Member, IEEE*, and Xin Jin, *Senior Member, IEEE*

***Abstract*—Disaggregating compute from storage is an emerging trend in cloud computing. Effectively utilizing resources in both compute and storage pool is the key to high performance. The state-of-the-art scheduler provides optimal scheduling decisions for workloads with homogeneous tasks. However, cloud applications often generate a mix of tasks with diverse compute and IO characteristics, resulting in sub-optimal performance for existing solutions. We present Pyxis, a system that provides optimal scheduling decisions for mixed workloads in disaggregated datacenters with theoretical guarantees. Pyxis is capable of maximizing overall throughput while meeting latency SLOs. Pyxis decouples the scheduling of different tasks. Our insight is that the optimal solution has an "all-or-nothing" structure that can be captured by a single *turning point* in the spectrum of tasks. Based on task characteristics, the turning point partitions the tasks either all to storage nodes or all to compute nodes (none to storage nodes). We theoretically prove that the optimal solution has such a structure, and design an online algorithm with sub-second convergence. We implement a prototype of Pyxis. Experiments on CloudLab with various synthetic and application workloads show that Pyxis improves the throughput by 3–21× over the state-of-the-art solution.**

***Index Terms*—Disaggregated datacenter, resource allocation, task scheduling.**

I. Introduction

RECENT advances in high-speed datacenter networks [1], [2], [3], [4], [5] along with low-latency transport protocols [6], [7], [8], [9], [10] have brought compute-storage disaggregation to the forefront. The emerging paradigm, serverless computing, adopts compute-storage disaggregation, and relies on disaggregated storage to store application data. Disaggregated storage such as AWS S3 [11], Azure Blob Storage [12], and Google Cloud Storage [13]—whereby remote storage nodes are accessed over the network from compute nodes—forms the backbone of modern cloud applications (e.g., Snowflake [14]), both for scalability and high resource utilization.




```
int PointerChasing (Key key) {
  int result = 0;
  while (key != Null) {
    Obj obj = Get(key);
    some_compute(obj, result);
    key = obj.next;
  }
  return result;
}
```

```
int Filtering (Array<Key> keys) {
  int result = 0;
  for key in keys {
    Obj obj = Get(key);
    if (some_condition(obj))
      some_compute(obj, result);
  }
  return result;
}
```

Fig. 1. Network overhead of remote data accesses.

However, accessing remote data still has a fundamental network cost. For IO-intensive tasks that require multiple network round-trip times (RTTs) or a large amount of data, it is very expensive to move data from storage nodes to compute nodes. Fig. 1 shows a pointer chasing and a data filtering example that stresses the latency and bandwidth overheads, respectively.

Storage-side computation via remote procedure calls (RPCs) has been proposed to alleviate this problem by offloading compute to storage nodes. Commercial products have already deployed storage-side computation that allows predefined or restricted functions to execute on storage nodes [15], [16], [17]. For example, AWS S3 Select [15] enables S3 to execute data filtering functions written in a restricted SQL API. Recent academic efforts [18], [19], [20], [21] propose to offload more flexible user-defined functions to storage.

Storage-side computation is not a panacea either, as storage nodes have rather limited compute power and are not intended for compute-intensive tasks. Recent works argue that tasks should be split between compute/storage nodes to fully utilize resources on both sides, and the split ratio should be based on how compute- or IO-intensive the tasks are [22], [23]. In this regard, Kayak [22] is the state-of-the-art that proactively finds the best split ratio for a given workload. It uses a gradient-based algorithm to dynamically compute the split ratio at the workload level, and then probablistically schedules each task in the workload to compute or storage nodes based on the split ratio. Kayak is capable of maximizing the overall throughput while meeting latency service level objectives (SLOs).

The key limitation of Kayak is that it assumes all tasks in a workload are similar in terms of resource consumption. It does not differentiate between potentially heterogeneous tasks and

Manuscript received 23 December 2023; revised 14 June 2024; accepted 14 June 2024. Date of publication 24 June 2024; date of current version 18 July 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500700, in part by the National Natural Science Foundation of China under Grant 62172008 and Grant 62325201, and in part by the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Recommended for acceptance by R. Prodan. *(Corresponding author: Xin Jin.)*

Sheng Qi, Chao Jin, Xuanzhe Liu, and Xin Jin are with the School of Computer Science, Peking University, Beijing 100871, China (e-mail: shengqi2018@pku.edu.cn; chaojin@pku.edu.cn; liuxuanzhe@pku.edu.cn; xinjinpku@pku.edu.cn).

Mosharaf Chowdhury is with the Computer Science and Engineering Division, University of Michigan, Ann Arbor, MI 48109-2121 USA (e-mail: mosharaf@umich.edu).

Zhenming Liu is with the Department of Computer Science, College of William & Mary, Williamsburg, VA 23187-8795 USA (e-mail: zliu@cs.wm.edu).

Digital Object Identifier 10.1109/TPDS.2024.3418620

1045-9219 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

computes a single split ratio to schedule them all. In reality, however, cloud applications usually generate a mix of tasks with diverse compute-IO characteristics. For example, the Snowflake dataset [14] shows that the ratio between total CPU time and amount of data storage IO can vary by up to six orders of magnitude (Section II). Consequently, existing solutions are not adequate for such workloads. For example, given a workload with a mix of compute-intensive and IO-intensive tasks, the scheduler should assign the former to compute nodes and latter to storage nodes, instead of using a common split ratio to schedule both tasks.

We propose Pyxis, a system that optimally schedules mixed workloads in disaggregated datacenters. Pyxis maximizes the overall throughput while meeting latency service level objectives (SLOs) of each task. It decouples the scheduling of different tasks with individual split ratios. However, finding the optimal split ratios for all tasks is challenging. Computing each split ratio independently, e.g., applying Kayak's algorithm to each task, does not work well in practice. We empirically validate that this approach has poor convergence even for a static workload with only two type of tasks (Section II). On the one hand, concurrently solving the split ratios results in the poor convergence due to interference across tasks. On the other hand, sequentializing this process, i.e., solving split ratios one at a time, avoids interference but inevitably slows down the convergence. Therefore, the key to solving the problem is to effectively prune the solution space.

Our insight is that the optimal solution has a clean structure that can be captured by a single *turning point* among tasks. For tasks that are more compute-intensive than the one at the turning point, it is always better to schedule *all* of them to compute nodes. Conversely, those that are more IO-intensive should all be scheduled to storage nodes, i.e., *none* to compute nodes. Only the task at the turning point may be split between compute/storage nodes. This "all-or-nothing" structure translates to a compact solution space, where per-task split ratios can be derived from a single variable. We theoretically prove that the optimal solution has such a structure (Section IV-B).

The turning point depends on the workload and the SLO targets. There is no closed-form expression that accurately accounts for the relationship between split ratios, latency, and throughput. Based on the "all-or-nothing" structure, we design a gradient-based search algorithm to compute the turning point, and combine it with a rate limiter that throttles the throughput to meet SLO targets. The two components form a dual loop control [22]. We theoretically prove that our algorithm can converge to the optimal solution with *logarithmic* running time. In addition, we introduce another control loop that performs elastic scaling to meet real-time demand, and provide mechanisms to handle load skew. In summary, we make the following contributions.

- We propose Pyxis, a system that optimally schedules mixed workloads in disaggregated datacenters. Pyxis maximizes the overall throughput while meeting latency SLOs.
- We identify and prove that the optimal solution has an "all-or-nothing" structure with a single turning point (Section IV-B), and design an online algorithm to dynamically compute the turning point with theoretical guarantees (Section IV-C).

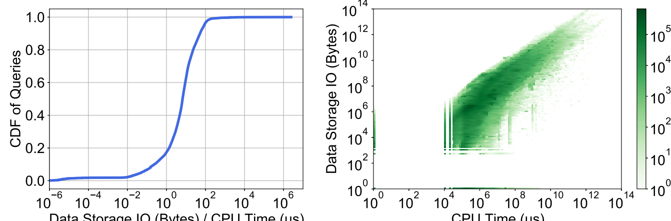


Fig. 2. Diverse characteristics of Snowflake queries.

- We implement a system prototype of Pyxis. Experiments on CloudLab [24] with a variety of synthetic and application workloads show that Pyxis improves the throughput by 3–21× over the state-of-the-art solution (Section VI). Pyxis is open-sourced at https://github.com/TomQuartz/Pyxis.

II. Motivation

Task model. Modern cloud applications typically comprise a collection of fine-grained tasks, e.g., microservices [25], [26], [27] or serverless functions [28], [29], [30], [31]. To be explicit, we use *task* to denote a user program that can be invoked upon requests, and *instance* to denote the running process of a task.

Following the compute-storage disaggregation architecture in cloud computing platforms [14], [28], [32], a task is by itself stateless, and relies on an external storage service (e.g., data store or message queue) for data passing and state management, which simplifies routing and data dependencies [33], [34], [35], [36]. Consequently, tasks have two kinds of basic operations: IO (accessing the data store) and compute (executing the application logic). In case the external data store is sharded, we also assume that each task only accesses data from a single storage shard, which is common for the fine-grained tasks and functions that constitute cloud workloads [37], [38], [39], [40]. This access pattern distinguishes between the cost of compute- and storage-side execution, where the latter retrieves data from local storage and pays no network overhead.

Pyxis accepts a user-provided SLO target for each task and focuses on completing the tasks within their SLOs. We allow users to specify an SLO target relative to (e.g., as multiples of) the task's average (or other percentiles of) service time and dynamically adjust SLOs.

Pyxis does not explicitly track inter-task dependency, so it should be coupled with a DAG scheduler [41], [42] that monitors task completion and submits ready tasks to Pyxis. We include furthur disscussion in Section VII about Pyxis' support for DAG workflows.

Diverse characteristics of tasks. Tasks may significantly differ in their compute and IO characteristics. Snowflake [14] is a representative data warehouse that disaggregates compute from storage. Fig. 2 visualizes the statistics of ∼70 million queries in its public dataset.[1] The ratio between CPU time and data storage IO can vary by up to six orders of magnitude. Even queries with

[1] https://github.com/resource-disaggregation/snowset

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

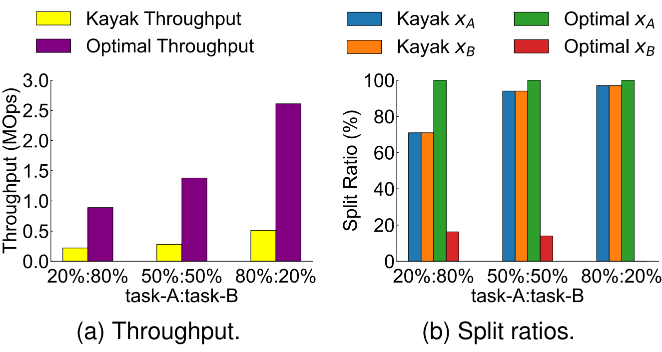


Fig. 3. The optimal solution outperforms Kayak by a wide margin through per-task scheduling.

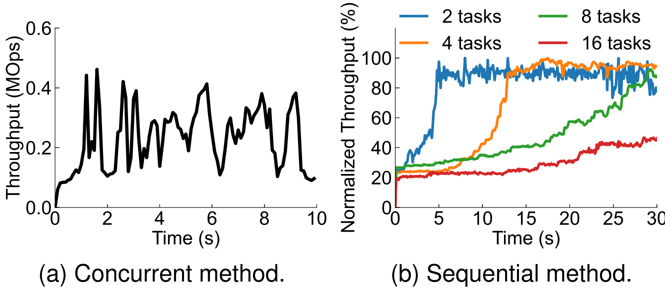


Fig. 4. Applying Kayak to each task is impractical. (a) Oscillations caused by concurrent updates. (b) Slow convergence for sequential updates.

the same amount of computation can markedly differ in their amounts of IO.

Opportunity: per-task scheduling. A *workload* is a set of tasks that collaboratively implement end-to-end functionalities. The scheduling problem we focus on is to find the best way to split tasks in a given workload between compute and storage nodes. Our goal is to maximize overall throughput while satisfying the SLO of each task. For a particular task, we use *split ratio* to denote a fraction $x \in [0, 1]$ such that x and $1 - x$ of its instances are executed on storage and compute nodes, respectively. The state-of-the-art solution, Kayak [22], proactively computes the optimal split ratio at the workload level, i.e., a single split ratio is configured for all tasks. Kayak assumes that tasks are homogeneous, i.e., with similar compute-IO characteristics. For generic mixed workloads in the real world, Kayak is sub-optimal as its assumption no longer holds. Consider a workload consisting of an IO-intensive task A and a compute-intensive task B. An intuitively better solution is to perform per-task scheduling, i.e., assigning task-A to storage nodes where it can avoid network overheads, and task-B to compute nodes to utilize the high compute power.

To confirm our conjecture, we perform an experiment with a synthetic workload. Task-A issues four storage accesses to a table of 48 KB large entries and performs 50 ns computation; task-B issues one access to a table of 64B small entries and performs 50 μs computation. We vary the fractions of task-A and task-B to make the workload more biased towards IO or compute. We compare Kayak against the optimal solution obtained through a parameter sweep over the split ratios of the two tasks. As shown in Fig. 3(a), the optimal solution improves the overall throughput by 3–5$\times$ over Kayak, which motivates the need to decouple the scheduling of different tasks with individual split ratios. We validate in Section VI that there could be a wider gap between Kayak and the optimal solution depending on the number of compute and storage nodes.

Challenge: jointly solving split ratios. While in general the optimal solution prefers to send compute-intensive tasks to compute nodes and IO-intensive tasks to storage nodes, the exact split ratios depends on the workload. Let (x_A, x_B) be the split ratios for task-A and task-B. Fig. 3(b) shows that the optimal split ratios are $(100\%, 16\%)$, $(100\%, 14\%)$ and $(100\%, 0\%)$ under three different settings. Besides workload configurations, the optimal solution also depends on latency SLO targets. However, there is no closed-form expression that explicitly models throughput as a function of split ratios and SLO targets. Kayak proposes a dual loop algorithm to solve this blackbox optimization problem, at the workload level. The inner loop performs rate limiting to enforce SLO, and the outer loop performs gradient ascent over the split ratio to optimize throughput. Since it is impossible to derive analytical gradient from the blackbox, we have to estimate the gradient by waiting for a time window after updating the split ratio and observe the delta in throughput.

To solve the split ratios per task, a naive approach is to apply Kayak's algorithm to each task independently. However, while the gradient estimation method is well-defined for a single variable, it is problematic when optimizing multiple split ratios. In fact, we have to update the split ratios one-at-a-time; otherwise, the delta in throughput reflects the composite effect of several concurrent updates, which leads to corrupted gradients and hence oscillations. To demonstrate this effect, we apply the concurrent method to the two tasks in the previous experiment. Fig. 4(a) confirms that concurrently solving the split ratios results in poor convergence, even if there are only two tasks. Note that using randomized time windows per task is not helpful, since the updating process would still be interleaved.

In contrast, the one-at-a-time counterpart avoids interference across tasks with sequential updates, but inevitably leads to slow convergence. This approach does not scale with the number of tasks due to the large solution space, which is an n-dimensional cube for n tasks. To evaluate this method, we randomly generate 16 tasks (Section VI) and vary the number of tasks in the workload from 2 to 16. Fig. 4(b) shows the dynamics of normalized throughput. The sequential method becomes computationally intractable as the number of tasks grows up. It cannot converge in 30 seconds with 16 tasks. Therefore, the critical problem is how to effectively prune the solution space.

III. Pyxis Overview

The insight of Pyxis is that the optimal solution has an "all-or-nothing" structure with a single *turning point*, which greatly simplifies the solution space.

Fig. 5 shows the overall architecture of Pyxis. It consists of three components: the disaggregated compute and storage nodes, and the scheduler. Tasks are installed at compute/storage

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

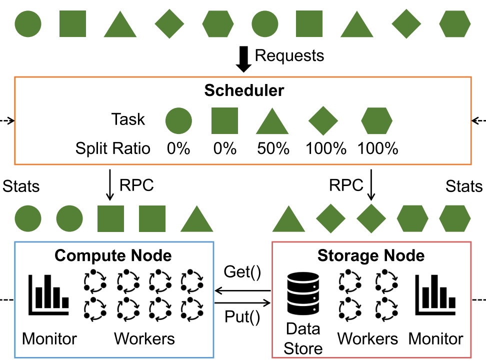


Fig. 5. Pyxis architecture.

TABLE I
KEY NOTATIONS IN PROBLEM FORMULATION

Notation	Definition
R	Overall throughput
$\vec{x}$	Split ratios
$R(\vec{x})$	Maximum throughput under SLO constraints
$\tau_i, \tau_i^C, \tau_i^S$	Random variables of task latency
T_i, T_i^C, T_i^S	Latency SLO metrics, e.g., 99%-tile latency
t_i, t_i^C, t_i^S	Latency SLO targets
c_i	Bandwidth cost of task-i on compute nodes
s_i	CPU cost of task-i on storage nodes
ρ_C	Total load on compute-storage bandwidth
ρ_S	Total load on storage CPU
α	The turning point

For latency terms τ; T; t, subscript i relates to instances of task-i, while superscripts C; S relate to instances running on Compute/storage nodes.

nodes as user-defined functions (Section V). Clients send task invocation requests to the scheduler. The scheduler identifies tasks from the task ID embedded in the request, and forwards it to compute/storage nodes.

Compute and Storage nodes dispatch requests to its workers. The number of workers is configured based on the capacity of the server (e.g., the number of CPU cores). The storage nodes collectively stores the data of applications. Storage node workers directly access the data in local storage, while compute node workers send remote key-value queries. Note that the storage nodes in Pyxis can be based on persistent storage or memory, offering the service of a key-value store or shared cache. Data may be sharded across storage nodes. For the sake of clarity when deriving our solution in Section IV, we assume for now that each task only accesses a single shard, which is a common requirement for stored procedures [43], [44], [45]. We discuss Pyxis' support of multi-shard tasks in Section VII. Clients include necessary information in the request for the scheduler to infer data locality. The candidate node of scheduling must host the requested shard if it is a storage node; compute nodes are not restricted. Both compute and storage nodes monitor runtime costs of tasks, and piggyback statistics on responses to the scheduler.

The Scheduler consists of a Rate Limiter and a Request Arbiter. Together they proactively compute the optimal assignment of tasks to compute or storage nodes. The Rate Limiter monitors end-to-end latencies and enforces SLO constraints. It throttles overflowing requests when the servers are under high load. The Request Arbiter proactively computes the optimal turning point based on task characteristics collected from compute/storage nodes. The two components form a dual loop control [22]. The Rate Limiter runs the fast/inner loop and the Request Arbiter runs the slow/outer loop. Given the current turning point of the outer loop, the inner loop determines the best throughput under SLO constraints. The outer loop in turn adapts the turning point using the outcome of the inner loop.

IV. PYXIS DESIGN

In this section, we first formulate the scheduling problem for mixed workloads in disaggregated datacenters (Section IV-A). Then we analyze the structure of the optimal solution (Section IV-B). We present an online gradient-based algorithm to find the optimal solution in Section IV-C. Finally we provide extensions of the algorithm to support resource elasticity (Section IV-D) and handle load skew (Section IV-E).

A. Problem Formulation

The blackbox. Table I summarizes the key notations. Let p_i be the fraction of task-i in the workload, $\sum_{i=0}^{n-1} p_i = 1$. We assume that p_i and the number of tasks n is stable (but unkown a priori) in a given period of time, since it usually takes a fixed combination of tasks to compose an application. Section V describes how we handle the dynamic case where p_i, n, and the compute-IO characteristics of tasks change over time. Let R be the overall throughput, and the throughput of task-i is $p_i R$. Denote the per-task split ratios by $\vec{x} = (x_0, x_1, \ldots, x_{n-1})$, where $x_i \in [0, 1]$ is the fraction of requests for task-i that are scheduled to *storage* nodes. Let τ_i be the random variable of latency for task-i instances, and T_i the SLO *metric* for τ_i, e.g., 99%-tile latency. Given the fraction p_i of each task, τ_i and T_i are functions of both R and $\vec{x}$, since resources are shared between collocated instances. Let t_i be the user-provided SLO target of task-i. We aim to maximize R while satisfying each t_i. We have the following problem.

$$\max_{\vec{x}} \ R \tag{1}$$

$$\text{s.t. } T_i(R, \vec{x}) \leq t_i, \ 0 \leq i \leq n-1 \tag{2}$$

We can apply Kayak's dual loop control [22] to solve the blackbox problem. Formally, given split ratios $\vec{x}$, let $R(\vec{x})$ be the maximum throughput under latency SLOs, which is implicitly determined by constraint 2 [22]. The inner loop approximates $R(\vec{x})$ using standard rate limiting techniques, e.g., AIMD [46]. The outer loop then performs gradient ascent over $\vec{x}$ to optimize $R(\vec{x})$. As described in Section II, concurrently solving x_i leads to oscillations, while updating x_i one-at-a-time is computationally intractable due to the vast solution space of $\vec{x}$. We next derive a fine-grained model of the system so that we can prune the solution space.

Fine-grained model. While there is no closed-form expression for τ_i, queuing theory predicts that latency is a monotonic

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

function of system load,[2] which gives us the opportunity to shed some light on the blackbox. We start by factoring τ_i into two parts, τ_i^C and τ_i^S, which are the random variables of latency for task-i instances when they are scheduled to compute/storage nodes. Similar to T_i, let T_i^C, T_i^S be the tail latency metrics for τ_i^C, τ_i^S, respectively. Depending on the split ratio x_i, we have the following relationship between T_i^C, T_i^S and T_i

$$T_i = T_i^C,\ x_i = 0 \tag{3}$$

$$T_i = T_i^S,\ x_i = 1 \tag{4}$$

$$T_i \le \max\left\{T_i^C, T_i^S\right\},\ x_i \in (0,1) \tag{5}$$

Consequently, in order to satisfy the SLO constraint 2 of the blackbox model, it suffices to bound T_i^C and T_i^S respectively.

We next incorporate the system load into our model. Let c_i, s_i be the end-to-end cost of a task-i instance on compute/storage nodes,[3] and let C, S be the capacity of respective resources. c_i, s_i are general abstractions of runtime cost and depend on the actual deployment. As for our underlying framework Splinter [18], c_i refers to the compute-storage bandwidth cost (measured as the amount of remote data transfer), and s_i refers to the CPU cost on storage nodes. c_i and s_i highlights the remote data transfer overhead and the limited compute power on storage nodes. CPU is typically not the contended resource for compute nodes, especially when we support elastic scaling of compute nodes (Section IV-D). We extend our cost model to multiple resources in Section VII.

Define $\rho_C(R, \vec{x}) := \sum_i p_i R(1-x_i)c_i/C$ and $\rho_S(R, \vec{x}) := \sum_i p_i R x_i s_i/S$ as the load on bandwidth and storage CPU,[4] which is the occupied capacity normalized by the total capacity. Following the literature of queuing theory [47], we assume that the arrival of tasks follows Poisson distribution, and T_i^C, T_i^S are *monotonic* functions of ρ_C and ρ_S, respectively. This suggests that we can bound T_i^C, T_i^S by minimizing ρ_C, ρ_S. Finally, we derive an alternative formulation as follows.

$$\max_{\vec{x}}\ R \tag{6}$$

$$\text{s.t.}\quad \rho_C(R, \vec{x}) \le 1 \tag{7}$$

$$\rho_S(R, \vec{x}) \le 1 \tag{8}$$

$$T_i^C(\rho_C) \le t_i^C,\ 0 \le i \le n-1 \tag{9}$$

$$T_i^S(\rho_S) \le t_i^S,\ 0 \le i \le n-1 \tag{10}$$

where t_i^C, t_i^S are the surrogate SLO targets for task-i instances on compute/storage nodes.

Remark. The equivalence between the original formulation (1)–(2) and the alternative formulation (6)–(10) is established by (3)–(5). The alternative formulation is easier to solve because ρ_C, ρ_S have closed-form expressions. However, the surrogate SLO targets t_i^C, t_i^S are still unknown. Fortunately, the optimal solution has an appealing structure that allows us to bypass t_i^C, t_i^S and develop an efficient algorithm.

B. Structure of the Optimal Solution

The insight of Section IV-A is to minimize ρ_C and ρ_S. Moreover, to compute the optimal solution, we only need to consider $\vec{x}$ where the corresponding pair (ρ_C, ρ_S) is Pareto Optimal, i.e., cannot decrease one variable without increasing the other. In other words, for each fixed ρ_C, the best $\vec{x}$ should minimize ρ_S, and vice versa. The formal result is given in the appendix [51], available online. The intuition is that any (ρ_C, ρ_S) pair that is not Pareto Optimal translates to non-tight SLO constraints, which possibly allows the throughput to increase. Since ρ_C, ρ_S have closed-form expressions, we can solve it analytically. We first use an example to illustrate the structure of the optimal solution.

Example. Denote $(c_i/C, s_i/S)$ as the normalized cost vector of task-i. Consider three tasks A,B,C with equal throughput of 0.5 and cost vectors (1,0.1),(1,1) and (0.1,1). Initially, suppose all tasks are scheduled to compute nodes, which is not feasible with $\rho_C = 1.05$. Suppose the desired ρ_C is 0.3, then we have to push some instances to storage nodes. In order to minimize ρ_S, for every unit of load removed from ρ_C, the increase in ρ_S should be as small as possible. In fact, by pushing task-i instances, we have $\frac{\Delta\rho_S}{\Delta\rho_C} = \frac{s_i/S}{c_i/C}$. It follows that task-A has the highest priority to run on storage nodes, followed by B and C. The optimal split ratios are $x_A = 100\%, x_B = 50\%, x_C = 0\%$, which yield $\rho_C = \rho_S = 0.3$.

All-or-nothing scheduling with a single turning point. We observe from the example that there is no *absolute* classification for a task to be compute-intensive or IO-intensive. Instead, whether a task is more suitable to run on compute or storage nodes depends on the *relative* ranking of s_i/c_i. After sorting the tasks in ascending order, we start from the first task and gradually push its instances to storage nodes, until ρ_C reaches the optimal value. This implies that most tasks should have a split ratios of 100% (*all* to storage nodes, for tasks at the front) or 0% (*none* to storage nodes, for those at the end). Only a single task in the middle may have a partial split ratio, i.e., splitting its instances between compute/storage nodes. This particular task, denoted by task-k, appears as the *turning point* of "all-or-nothing" scheduling. We formalize our findings in Theorem 4.1.

We highlight the intuition here and defer the full proof to the appendix [51], available online. Let $\vec{x}$ be an optimal solution. Let pair $i < j$ be a *bad* pair if $x_i < 1$ and $x_j > 0$. When the number of bad pairs is non-zero, we can always build another optimal solution with less bad pairs. We do so by trading task-i on compute nodes for task-j on storage nodes, under the constraint that ρ_C remains the same. Because $s_i/c_i < s_j/c_j$, for any unit load of ρ_C exchanged between i and j, ρ_S always decreases, so the SLO constraints are still satisfied. We keep trading until $x_i = 1$ or $x_j = 0$. Consequently, the new solution eliminates one bad pair and preserves the same optimal throughput. We may derive an "all-or-nothing" solution after eliminating all bad pairs. We empirically verify the correctness of Theorem 4.1 in Section VI-A and discuss the our limitation in Section VII.

[2] Applies to M/G/n queues under generic task distributions [47], [48]. M/G/n is widely adopted in both thoery and system literature [49], [50].

[3] Formally, c_i, s_i are random variables, but it suffices to use their expectation for the queuing theory involved in this paper [47], [48].

[4] For now, we omit the locality constraints and assume that any node can be a candidate of scheduling. We incorporate data locality in Section IV-E.

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

Theorem IV.1. Let tasks be sorted by s_i/c_i in ascending order. There exists an optimal solution $\vec{x}$ with a turning point k, s.t.

$$x_0 = x_1 = \cdots = x_{k-1} = 100\%, \quad (11)$$

$$x_k \in [0, 100\%], \quad (12)$$

$$x_{k+1} = x_{k+2} = \cdots = x_{n-1} = 0\% \quad (13)$$

Note that it is possible for multiple tasks to have the same s_i/c_i as their compute-IO characteristics. In that case, we can break ties by comparing task IDs. Therefore, there is a total order on the set of tasks such that the turning point is well-defined. Tasks having the same compute-IO characteristics are equivalent in terms of resource efficiency in our formulation, so it does not matter which task is split by the turning point. Therefore, we can break ties arbitrarily.

C. Scheduling Algorithm

Theorem 4.1 suggests that the solution space can be captured by a single variable, as opposed to the n-dimensional cube of $\vec{x}$. We use a real number $\alpha \in [0, 1]$ to represent the turning point in the theorem, where $\vec{x}(\alpha)$ is given by

$$x_i = \begin{cases} 1 & \alpha \in [(i+1)/n, 1] \\ \alpha n - i & \alpha \in (i/n, (i+1)/n) \\ 0 & \alpha \in [0, i/n] \end{cases} \quad (14)$$

Consequently, we can simplify $R(\vec{x})$ of the blackbox model (Table I) to $R(\alpha)$, and the outer loop is responsible for finding $\alpha^* := \arg\max_\alpha R(\alpha)$.

Algorithm for outer loop. Our algorithm assumes the unimodality of $R(\alpha)$ [52], [53], [54], i.e., monotonically increasing on one side and monotonically decreasing on the other side, which is a weaker condition than the concavity assumption of standard maximization methods [55]. We verify the unimodality of $R(\alpha)$ in Section VI-A.

Specifically, we maintain a lower bound $\underline{\alpha}$ and an upper bound $\overline{\alpha}$ for α^*, and iteratively prune the search space. In each iteration, we equally divide $[\underline{\alpha}, \overline{\alpha}]$ into M segments, where M is a constant, and perform grid search over the endpoints. Our goal is to find a local maximum on the grid, i.e., three endpoints $\alpha_{i-1} < \alpha_i < \alpha_{i+1}$ such that $R(\alpha_i) > R(\alpha_{i-1})$ and $R(\alpha_i) > R(\alpha_{i+1})$. The unimodality of $R(\alpha)$ implies that $\alpha_{i-1} < \alpha^* < \alpha_{i+1}$, so we can shrink the search space to $[\alpha_{i-1}, \alpha_{i+1}]$, which consists of two adjacent segments. Since each segment is a $1/M$ fraction of the original search space, we can guarantee that the problem size is reduced by a constant factor after each iteration.

Algorithm 1 shows the pseudocode. We make one step forward at a time, and wait for a time window after updating α to estimate the gradient $R'(\alpha)$ (line 3). We use $R'(\alpha)$ to determine the local maximum (line 5) and the direction of the next step (line 11). If the local maximum is found, we shrink $[\underline{\alpha}, \overline{\alpha}]$ and set step size Δ to a $1/M$ fraction of the interval for the next iteration. The direction of steps alternates across iterations, because the sign of $R'(\alpha)$ is reversed upon detecting the local maximum.

In practice, the search space is initialized as $[0, 1]$, and Algorithm 1 runs in the outer loop until $\overline{\alpha} - \underline{\alpha}$ is sufficiently small, where an arbitrary point in $[\underline{\alpha}, \overline{\alpha}]$ is used as α^*. The scheduler continues to profile the number of tasks and their runtime costs after convergence. Upon detecting a change in the workload (Section V), it resets $\underline{\alpha}$ and $\overline{\alpha}$ and restart the loop.

Algorithm 1: Iterative Algorithm for Finding α^*.

```
1:  procedure Update-α
2:    /∗ estimate gradient ∗/
3:    R′(α_t) ← (R(α_t) − R(α_{t−1})) / (α_t − α_{t−1})
4:    /∗ find local maximum ∗/
5:    if R′(α_{t−1})·R′(α_t) < 0 then
6:      /∗ prepare for next iteration ∗/
7:      ᾱ ← max{α_{t−2}, α_t}
8:      α̲ ← min{α_{t−2}, α_t}
9:      Δ ← (ᾱ − α̲)/(2 M)
10:   /∗ determine the direction of next step ∗/
11:   α_{t+1} ← α_t + sign(R′(α_t))Δ
12:   t ← t + 1
```

Convergence analysis. Algorithm 1 is guaranteed to converge with *logarithmic* running time (full proof in the appendix [51]), available online.

Theorem IV.2. For arbitrarily small δ and $M > 2$, Algorithm 1 converges to $[\alpha^* - \delta, \alpha^* + \delta]$ within $O(log(1/\delta))$ steps, regardless of the initial state.

The intuition of the proof is that the search space is reduced by $M/2$ after each iteration. By the unimodality of $R(\alpha)$, once we find a local maximum during the grid search, we can be sure that α^* is within the most recent *two* consecutive steps. We then shrink $[\overline{\alpha}, \underline{\alpha}]$ to cover the two steps, which gives the desired reduction factor. It follows that the number of iterations is logarithmic, and the number of steps in each iteration a constant M, concluding the proof.

Remark. A key advantage of Algorithm 1 over convex optimization methods [55] is that our approach is more robust to variance in throughput. Algorithm 1 only uses the *sign* of gradient, while convex methods uses the *value* of gradient. This allows us to use a small time window in gradient estimation to speed up convergence. In addition, convex methods also use the *value* of gradient to as the stopping criterion, which is sensitive to fluctuations, while ours is based on length of search interval. Finally, our assumption of unimodality is weaker than concavity required by the convex methods (for maximization). We compared our method against convex methods in Section VI-B.

D. Supporting Resource Elasticity

Given a static provision, the rate limiter of the scheduler (Section III) enforces SLOs by throttling requests. To meet real-time demand, we support elastic scaling of compute nodes. We focus on compute nodes as they are provisioned based on the amount of tasks submitted by users, which varies from time to time. Storage provision, on the other hand, are determined by the amount of persistent data, which is relatively stable. This is in line with production systems. For example, Snowflake [14] elastically scales the number of AWS EC2 instances inside a Virtual Warehouse (VW), and charges customers based on service time and VW size.

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

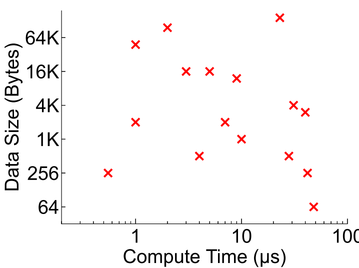


Fig. 6. Characteristics of 16 sampled tasks. Data size equals *number of data accesses* $\times$ *object size*.

To support elasticity, we wrap the α control loop inside another slower/outer loop. When requests are dropped, this loop gradually increases the amount of compute resources until all demand can be met. Otherwise, it utilizes the load variable ρ_C, ρ_S of the α control loop and searches for the minimum amount of compute resources without impacting throughput or violating SLOs. Our current prototype supports scaling at CPU core-level granularity, and can adapt to other units defined by the capacity of workers.

E. Handling Locality and Skew

Note that ρ_C and ρ_S represent the *average* load on bandwidth and storage CPU. In practice, tasks access specific shards and have locality constraints on storage nodes. Under skew, ρ_C, ρ_S may deviate from the actual load on some hotspots, which leads to suboptimal performance. For example, let there be two tasks A,B; A precedes B in the s_i/c_i ascending order. Suppose task-A instances always access the hotspot, while task-B instances may access the hotspot or some other shard. The "all-or-nothing" scheduling may choose to send none of task-B to storage nodes, as it observes high latency due to the hotspot and cannot push any more tasks to the storage pool. However, those of task-B that access the non-hotspot may still benefit from storage-side execution.

We introduce *placement groups* to address this problem. Each placement group has its own set of compute/storage nodes. Tasks (distinguished by data locality) are mapped to a single group. When there is no skew, i.e., ρ_C, ρ_S is representative of the actual load, all nodes and tasks belong to the default placement group. If a storge node's local load exceeds the average load by some threshold, it signals the scheduler to isolate it in a new placement group. The number of compute nodes in this new group can be arbitrary, as we support elastic scaling; we can configure the initial number based on the compute-intensity of tasks in this group. Each group independently performs dual loop control and elastic scaling, and schedules tasks with its own turning point. Unlike the concurrent version of Kayak (Section II), there is no interference across groups because resources are isolated.

As load shifts overtime, we can merge groups back to the default group. Theoretically, we can configure a placement group for each storage node, but the overhead of running multiple instances of dual loop control might overwhelm the scheduler. We do not investigate the impact of the maximum number of placement groups due to our limited cluster size.

TABLE II
TASK CHARACTERISTICS OF THE APPLICATION WORKLOAD

Query	Object Size	IO-intensive	Compute-intensive
Aggr-V	64KB	✓	✗
Top-k	12KB	✓	✓
Aggr-S	1KB	✗	✗
Auth	40B	✗	✓

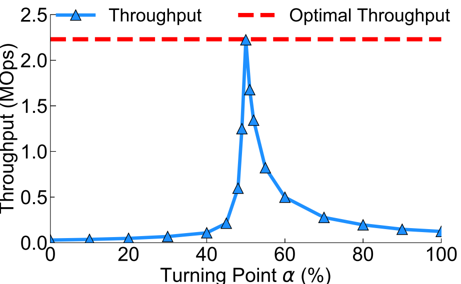


Fig. 7. $R(\alpha)$ is unimodal, and its maximum coincides with the theoretical upper bound.

V. IMPLEMENTATION

We implement a prototype of Pyxis with $\sim$4000 lines of code in Rust. We use the in-memory key-value store Splinter [18] as the data store on storage nodes.

User-defined functions. Users write tasks as Splinter extensions, which are invoked as stackless co-routines. We provide the same Get/Put API for extensions to access both local and remote data store. Users code only once, and the placement decision is transparent to extensions.

Dynamic workloads. The first time the scheduler sees a new task, it schedules that task to compute and storage nodes with a default split ratio (empirically set to 50%) to profile the runtime costs c_i, s_i, until there are enough samples to make a reliable estimation. For dynamic tasks whose characteristics change over time, the scheduler maintains a moving average of c_i, s_i and compares them with the statistics collected during the period. If there is a significant difference in either cost, the scheduler resets the moving average and falls back to the default split ratio for that task. The scheduler also periodically sorts the tasks it recently observes, which handles dynamic tasks and addition/removal of tasks.

VI. EVALUATION

Methodology. Our experiments are conducted on CloudLab [24] XL170r machines, each configured with eight CPU cores (Intel E5-2640v4 2.4 GHz), 64 GB RAM, and Mellanox ConnectX-4 25 GB NIC. Each worker uses one CPU core, and each node hosts eight workers. By default, we set up one scheduler node, eight compute nodes and one storage node. To demonstate the scalability of Pyxis, we use up to 64 compute nodes and eight storage nodes. In line with Kayak [22], we collocate a closed-loop load generator with the scheduler. We

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

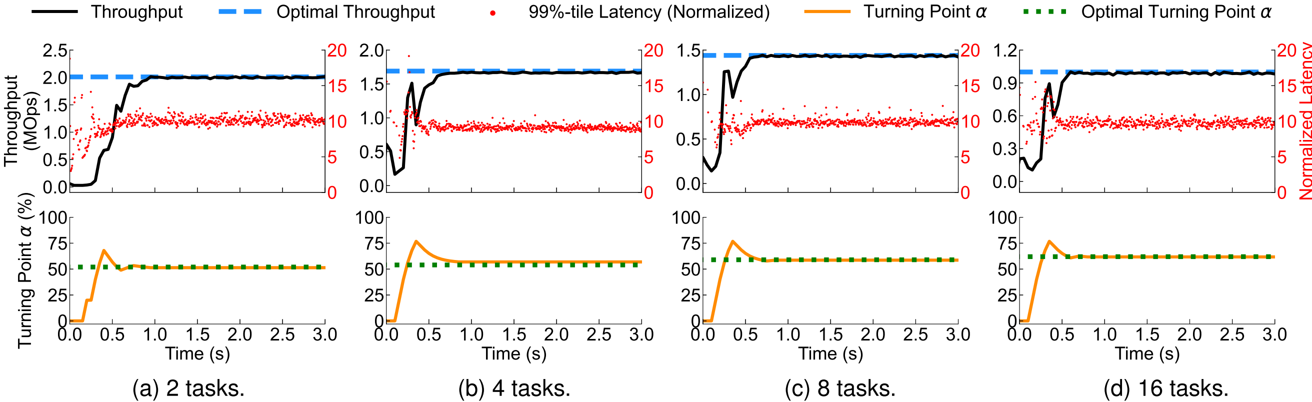


Fig. 8. Convergence under mixed workloads with different number of tasks.

use the loop frequency reported in Kayak for the dual loop control, which is 20 Hz for the outer loop (α control) and 200 Hz for the inner loop (rate control). The factor M is set to 5 for Algorithm 1. We perform sensitivity analysis over M in Section VI-G.

Workloads. Synthetic tasks are configured to issue a series of key-value operations, followed by a certain amount of computation. First, we use a workload with tasks A,B in Section II as two polarized tasks in terms of IO and compute. Second, we randomly sample 16 tasks to capture the diverse characteristics of cloud applications. Each task is represented by the tuple *(number of data accesses, object size, computation)*, and we use the parameters of the two polarized tasks as upper bound and lower bound for the three fields. The number of data accesses is sampled uniformly while the rest are sampled from log-uniform distribution. Specifically, the number of data accesses is sampled from one to four, and the object size is sampled from 64B to 48 KB. We visualize the sampled tasks in Fig. 6. By default, We use equal fractions for all tasks in a mixed workload. We use the term "fraction" to denote the number of requests targeting a specific task relative to the total number of requests. Thus "equal fractions" means that each task in the workload receives the same number of requests.

We also implement a realistic application workload that features hybrid queries over vector and scalar data. The workload includes four tasks: (i) aggregation over vector data (aggr-V); (ii) top-k query using vector distance as the metric of similarity (top-k); (iii) aggregation over scalar data (aggr-S); (iv) authentication with cryptographic hashing (auth). As shown in Table II, each task maps to a canonical category in IO-$\{$intensive,light$\}\times$compute-$\{$intensive,light$\}$.

Metrics. We define the SLO metric of a task as the 99%-tile latency normalized by the mean service time,[5] which is in line with ZygOS [50]. The SLO target of a mixed workload is defined as a constant such that the SLO metrics of all tasks do not exceed that constant. We then measure the maximum throughput under

[5]We use 5 μs for tasks with mean service time <5 μs, since it is hard to bound the tail latency of theses tasks using Splinter's extension interface.

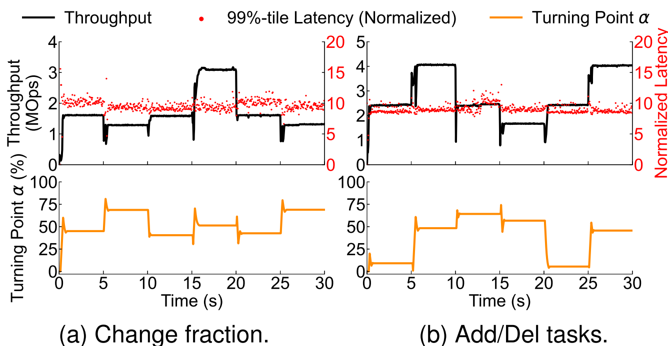


Fig. 9. Convergence under dynamic workloads: (a) varying task fractions. and (b) varying the set of tasks.

the SLO target. Unless otherwise specified, we set the SLO target to 10.

Baselines. Our primary baseline is Kayak [22], the state-of-the-art solution for task scheduling in disaggregated datacenters. In addition, we evaluate an extended version of Kayak, Kayak+, that performs sequential updates over individual split ratios, as described in Section II. Kayak+ has the same theoretical capability as Pyxis to compute the optimal solution, but is slow in convergence due to its vast solution space (Section VI-A). For reference, we also obtain the upper bound of our algorithm by performing a parameter sweep over α.

A. Convergence

Verifying Theorem 4.1 and the unimodality of $R(\alpha)$. We use the two polarized tasks A,B and perform parameter sweeps over: (i) split ratios (x_A, x_B) to obtain the theoretical optimal; (ii) turning point α to obtain the curve of $R(\alpha)$. As shown in Fig. 7, the maximum of $R(\alpha)$ coincides with the optimal, which confirms the "all-or-nothing" structure of Theorem 4.1. Moreover, $R(\alpha)$ monotonically increases in $[0, \alpha^*]$ and monotonically decreases in $[\alpha^*, 1]$, which validates the unimodality assumption of $R(\alpha)$ and supports the correctness of Algorithm 1.

Basic convergence under mixed workloads. We evaluate the convergence of our algorithm under the same setting as Fig. 4(b).

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

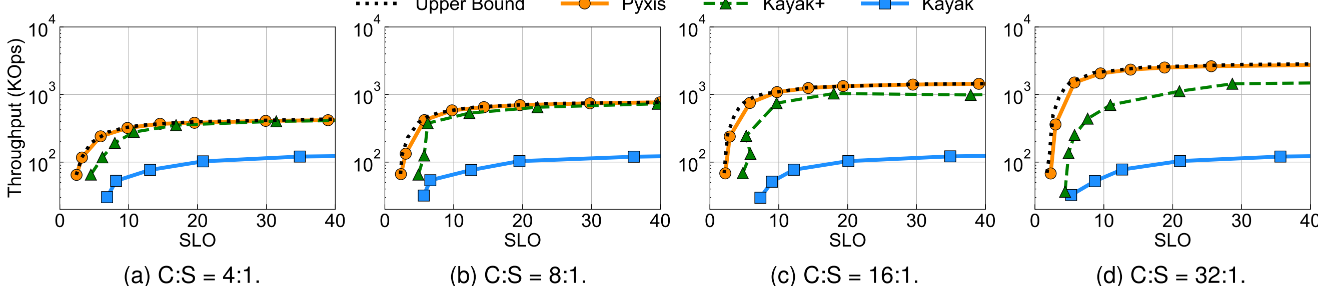


Fig. 10. Performance under different resource configurations.

Fig. 8 shows that Pyxis converges to the optimal solution in <800 ms, regardless of the number of tasks. In comparison, the time required by Kayak+ in Fig. 4(b) increases rapidly with the number of tasks. Pyxis improves the convergence speed by 5–30× over Kayak+. In addition, the latency of task instances satisfies the SLO target with minor fluctuations. We highlight a few observations.

- The α stays idle for a short period in the beginning, as the cost and order of tasks are yet unknown to the scheduler. The scheduler uses the default split ratio (50%) to profile tasks on compute/storage nodes (Section V). By the end of this stage, it sorts the tasks and starts from $\alpha = 0$.
- There may be a drop in throughput as $\alpha = 0$ does not necessarily perform better than using the default split ratio for all tasks. The throughput oscillates because our algorithm has to go across α^* before shrinking the search space. There are at most two oscillations.
- Our algorithm is stable after convergence. It is insensitive to variance in throughput, since the stopping criterion is based on $\overline{\alpha} - \underline{\alpha}$ instead of gradient. We do not re-compute α^* unless we detect a change in the workload (Section IV-C).

Dynamic workloads. We use four tasks from Fig. 6 and consider two settings: (i) the set of tasks is fixed, and the fractions of tasks changes over time; (ii) the set of tasks changes over time. In the first setting, we use four stages where each task in turn dominates with the largest fraction, and each stage lasts for five seconds. In the second setting, we add a new task for every five seconds; after all four tasks are present, we loop back to the initial stage. Fig. 9 shows the dynamics of the throughput and α. Pyxis responds quickly to the dynamic workload. It consistently achieves sub-second convergence while meeting SLOs.

B. End-to-End Performance

Performance improvement. We compare the performance of Pyxis against the baselines under different resource configurations. Let C:S be the ratio between the number of CPU cores on compute and storage nodes. We configure the storage node with four cores and vary C:S from 4:1 to 32:1. We use the two polarized tasks A,B with fractions of 20% and 80%. Fig. 10 shows the maximum throughput as a function of SLO target. Pyxis improves the saturated throughput by 3–21× over Kayak. Kayak saturates earlier because it uses a single split ratio for all tasks, and has bottleneck either in IO or compute while the entire cluster remains under-utilized. Pyxis achieves high throughput with per-task scheduling. Kayak+ performs worse than Pyxis because it needs to control every single split ratio x_i. It is hard for Kayak+ to converge exactly to the "all-or-nothing" structure of the optimal solution. The gap between Pyxis and Kayak+ is amplified as more resources are added to the pool. Moreover, Kayak+ is a convex method that uses the value of gradient, which is sensitive to fluctuations in throughput, making it even harder for Kayak+ to stick to the optimal solution (Section IV-C).

Gap between Pyxis and upper bound. We further vary the fractions of A,B in the previous experiment and compares the saturated throughput of Kayak, Pyxis and the upper bound under different combinations of workload and resource configurations. Table III shows the improvement factor of Pyxis over Kayak and the gap between Pyxis and upper bound. When task-A:task-B is 20%:80%, Pyxis's throughput increases linearly with more compute nodes. This is because task-B (compute-intensive) dominates and Pyxis schedules it to compute nodes to avoid storage-side bottlenecks. As the ratio comes to 50%:50%, Pyxis's throughput stops increasing when C:S goes beyond 16:1, since the two tasks are now even and the storage is saturated earlier. For the ratio of 80%:20%, there is no significant gain in throughput when more compute nodes are added, because the storage becomes the bottleneck. However, the throughput of Kayak does not scale with more compute nodes for all three settings, because the storage is always its bottleneck. The gap between Pyxis and the upper bound is consistently within 6%.

Multiple tasks in the workload. we select 4 to 16 tasks from Fig. 6 to evaluate the performance of Pyxis under multiple tasks. As shown in Fig. 11, Pyxis improves Kayak by at least 3×, and is close to the upper bound. The gap between Pyxis and Kayak+ increases with the number of tasks. Similar to Fig. 10, this is because Kayak+ cannot accurately and stably converge to the optimal solution, especially for $\vec{x}$ of high dimensions.

C. Realistic Application

We implement an application workload to further validate the effectiveness of Pyxis. The data model of the workload is a social graph like Facebook TAO [56], where each vertex is associated with several vector attributes (e.g., features of human faces produced by deep learning models) and some scalar attributes. This workload exemplifies emerging AI applications [57], [58], [59] and hybrid queries over both vector and scalar data [60],

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

TABLE III
SATURATED THROUGHPUT OF KAYAK, PYXIS AND UPPER BOUND UNDER DIFFERENT SETTINGS

	task-A:task-B = 20%:80%			task-A:task-B = 50%:50%			task-A:task-B = 80%:20%		
	Kayak	Pyxis	Upper bound	Kayak	Pyxis	Upper bound	Kayak	Pyxis	Upper bound
C:S = 4:1	0.134	0.440(3.3×)	0.448(+1.8%)	0.134	0.676(5.0×)	0.692(+2.3%)	0.203	1.499(7.4×)	1.514(+1.0%)
C:S = 8:1	0.135	0.781(5.8×)	0.787(+0.8%)	0.134	1.203(9.0×)	1.221(+1.5%)	0.215	1.625(7.6×)	1.718(+5.4%)
C:S = 16:1	0.135	1.483(11.0×)	1.498(+1.0%)	0.134	2.081(15.5×)	2.180(+4.5%)	0.221	1.626(7.4×)	1.718(+5.4%)
C:S = 32:1	0.136	2.874(21.1×)	2.888(+0.5%)	0.134	2.085(15.6×)	2.212(+5.7%)	0.222	1.627(7.3×)	1.718(+5.3%)

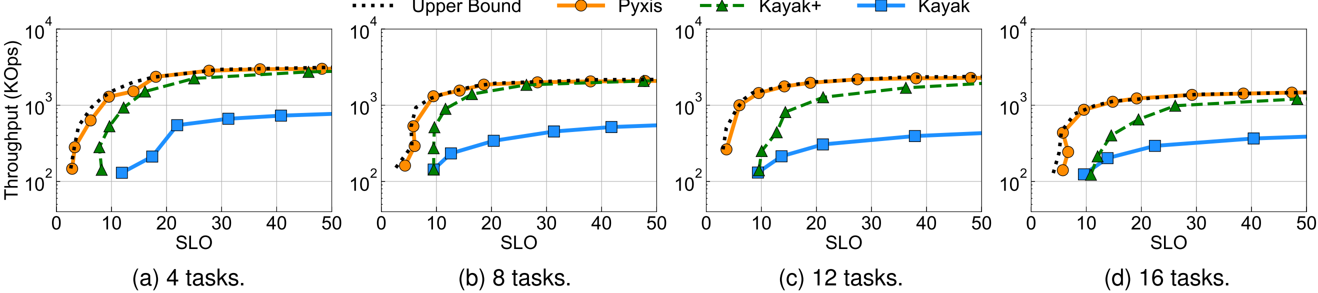


Fig. 11. Performance under different number of tasks.

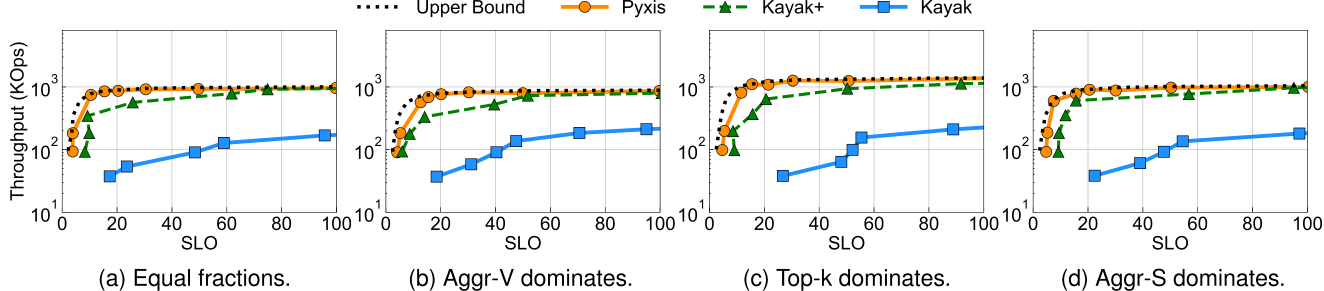


Fig. 12. Performance of a realistic application workload with four tasks.

[61], [62], [63], [64]. Specifically, the data store contains four tables, namely vector, association (edges in the graph), scalar, and authentication (cryptographic hash). Each table contains 100,000 entries We implement four tasks (Table II) as follows.

- The *vector aggregation* (aggr-V) task computes the statistics of an entry in the vector table, where each entry contains 64 256-dimensional vectors of 4B float values.
- The *top-k* task computes the nearest neighbors of a vertex. It has three stages: (i) it fetches the keys of neighbors from the association table; (ii) it accesses the vector table to get a number of vector attributes for the given vertex and all of its neighbors; (iii) it computes a score for every neighbor (e.g., using vector distance) and returns the keys of top-k neighbors with the highest scores. Each entry in the association table contains 12 keys.
- The *scalar aggregation* (aggr-S) task is the scalar version of the vector aggregation task. It accesses the scalar table where each entry is an array of 256 4B float values.
- The *authentication* (auth) task verifies user identity using scrypt [65]. Our implementation is similar to that in ASFP [23]. Each entry in the authentication table contains a 40B salted hash. This task receives a user ID and an encrypted password. It applies salted hashing to the password, and compares the result with the stored hash.

We fix the fraction of authentication tasks to 10% and explore four settings where the rest three tasks have equal fractions or one task dominates with the largest fraction. Similar to our results in Section VI-B, Pyxis outperforms the baselines and is close to the upper bound. As shown in Fig. 12, Pyxis improves the throughput by 3–4× over Kayak in all settings.

D. Scalability

We vary the number of storage nodes from one to eight and the number of compute nodes from 4 to 48. The workload is a 50%:50% mix of the two polarized tasks as it imposes a considerable load on both compute and storage. Fig. 13 shows the saturated throughput of Pyxis under different configurations. Pyxis's throughput increases with additional resources on both sides. The storage becomes the bottleneck when there is only one storage node and more than eight compute nodes. Otherwise,

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

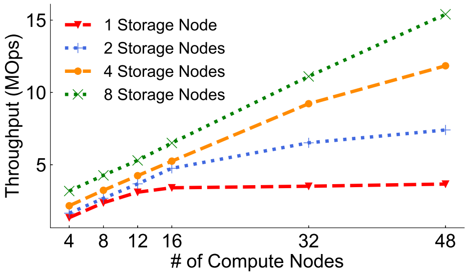


Fig. 13. Scalability.

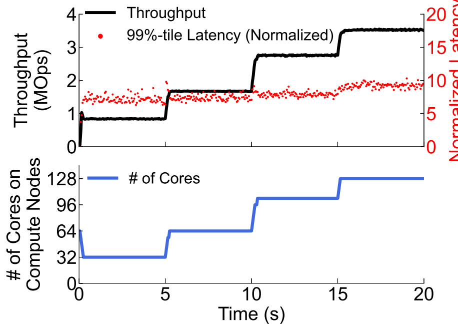


Fig. 14. Elastic scaling of compute nodes.

given enough storage nodes, Pyxis's throughput scales linearly with the number of compute nodes.

E. Elasticity

Pyxis dynamically adjusts the number of workers on compute nodes according to real-time demand. We use the two polarized tasks and set the initial demand as a quarter of the maximum throughput when 16 compute nodes (128 cores) are deployed. We increase the demand by another quarter for every five seconds. Fig. 14 shows the dynamics of throughput, latency and the number of CPU cores on compute nodes. At the start, the system is provisioned with one storage node and eight compute nodes (64 cores). However, 32 cores would be sufficient for the initial demand, and Pyxis reduces resource consumption accordingly. When the demand increases later, Pyxis quickly responds by adding more cores to the system. Pyxis achieves elasticity at sub-second time scale and has negligible impact on latency.

F. Handling Load Skew

Pyxis handles load skew by isolating hotspots in separate placement groups (Section IV-E). We use four storage nodes and up to eight compute nodes. The workload has four tasks: the first task only accesses the first storage node, the second task accesses the first two storage nodes, etc. Therefore, the load on the four storage nodes is of descending order. We vary the number of placement groups from one to four. Fig. 15(a) shows that the throughput increases with the granularity of groups. With more groups, Pyxis can make better estimation of the skewed load in the cluster. It aggressively uses higher values of α on storage

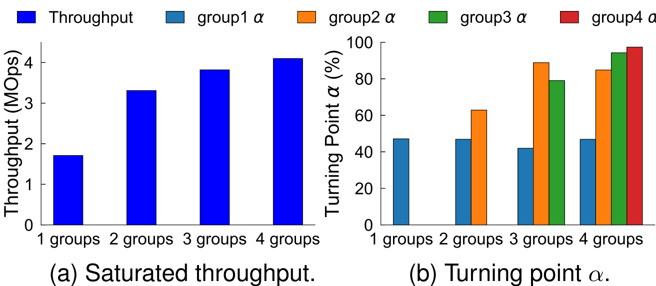


Fig. 15. Impact of the number of placement groups.

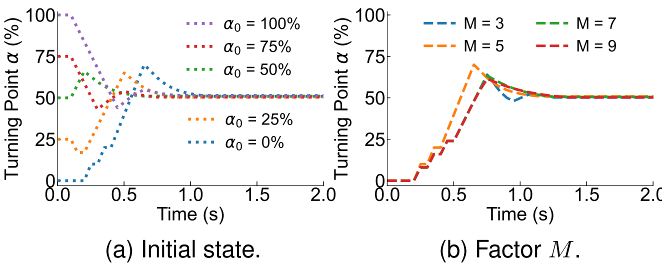


Fig. 16. Dynamics of α under different parameters.

nodes with low load, so that tasks may benefit from storage-side computation, as shown in Fig. 15(b).

G. Sensitivity Analysis

Initial state. We first evaluate Pyxis's sensitivity to its initial state. We use the two polarized tasks with equal fractions and vary the initial α from 0% to 100%. Fig. 16(a) shows that Pyxis has sub-second convergence regardless of its initial state, which confirms our analysis in Theorem 4.2.

Choice of M. Recall that Algorithm 1 equally divides $[\underline{\alpha}, \overline{\alpha}]$ into M segments and reduces the search space by $M/2$ for each iteration. Intuitively, larger M reduces the number of iterations but increases the number of steps in each iteration. To evaluate its impact, we vary M from 3 to 9 and start from $\alpha = 0$. As shown in Fig. 16(b), $M = 5$ achieves the best convergence speed, as it strikes a good balance between the number of iterations and the number of steps. Other choices of M is only slightly slower in convergence. Pyxis is not sensitive to this parameter in general.

VII. Discussion

Cost model. The runtime costs c_i, s_i present two major limitations. First, they are end-to-end costs, which does not account for different access patterns (e.g., one bulk read versus many small reads). We can use heuristics to incorporate pattern information, e.g. inflate the end-to-end cost by some factor according to the size/number of data accesses. Second, Pyxis averages c_i, s_i across instances of the same task. The average is representative of task characteristics for most cases, but may fail to cover some outliers due to intra-task variations, e.g., the dispersion in data size. This problem resembles information-agnostic scheduling with unknown task durations, where the task duration is unknown [66]. We can apply ideas like Least-Attained Service

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

(LAS) [47] to adaptively re-schedule tasks between compute and storage nodes. Alternatively, Pyxis can make better cost estimation given additional hints about task execution, like SQL query optimizers [67], [68]. It is an interesting direction for future work.

c_i and s_i represents a single resource each. To consider multiple resources, however, would escalate the problem to multi-dimensional bin packing [69], which is APX-hard [70]. Several heuristics [41], [71] have been proposed by the resource scheduling literature, and the basic idea is to collapse the cost vector into a scalar by taking a weighted sum. Pyxis can use theses heuristics to compute c_i, s_i, and its control logic is still applicable.

Profiling. The "all-or-nothing" scheduling disables the profiling of either c_i or s_i after the initial stage. However, the runtime costs may change over time for dynamic tasks. We can set a time-to-live (TTL) field for each task and re-profile periodically. Moreover, it is reasonable to assume that changes in a task would affect *both* c_i and s_i, which could be detected by mechanisms in Section V.

Optimization objective. Pyxis aims to maximize overall throughput R under SLO constraints. Under low demand it does not optimize latency as long as the SLOs are satisfied. In that case, however, more aggressively splitting tasks between compute/storage nodes could furthur improve response times. We can switch to another scheduling policy under low demand, e.g., group/split tasks with existing heuristics [41].

In practice, tasks may have different importance, and we can extend the original objective R by associating a weight w_i with task-i. The objective becomes $R \sum_i p_i w_i$, where p_i is the fraction of task-i. Note that setting $w_i = 1$ gives the original objective, and both aim to maximize R, so the analysis and algorithm in Section IV are still applicable. We assume that p_i is stable (in a period of time), since it usually takes a fixed combination of tasks to compose an application.

Multi-shard tasks. In Section IV, we assume for simplicity that a task targets a single data shard such that it accesses local data only once scheduled to storage nodes. In practice, a task may access multiple shards and cannot be factored into smaller single-sharded pieces. Pyxis can be extended to support such tasks in the following way. When scheduling a task to execute on storage nodes, the Pyxis scheduler effectively chooses one that can provide the most hit, i.e., max coverage of the task's read/write set. In case of a partial hit, missing data can be fetched from other storage nodes. Having to wait for remote data to arrive could increase the CPU consumption of a task on storage nodes, making it less favorable for storage-side computation, as reflected by its relative ranking of s_i/c_i.

Inter-task dependencies. One limitation of Pyxis is that it does not consider inter-task dependencies. Ideally if a downstream task consumes the output of an upstream task, then they should be collocated on the same node for fast local data passing. However, many existing works have explored bundling multiple tightly coupled workflow stages in a DAG (e.g., those involving huge data transfers) into a single group, and scheduling the DAG at the granularity of groups [72], [73], [74]. Pyxis can be layered on top of such grouping mechanisms, such that inter-group dependencies are weak and eligible for independent placement.

Scalability. Pyxis is horizontally scalable (Section VI-D), as its disaggregated architecture allows independent scaling of compute and storage pools. Pyxis's control loop is typically not the bottleneck, because it is off the critical path of request handling. In fact, Pyxis can be integrated in Layer 7 load balancers. It only needs to observe the throughput and latency during the past time window and periodically updates the split ratios, which allows it to run asynchronously. To increase the throughput for large clusters, one can simply add more load balancers. Thanks to the sub-second convergence (Section VI-A) of Pyxis, there is typically no need to persist the state of the scheduler.

Multi-tenancy. Pyxis is built on top of the multi-tenant data store Splinter [18], which support fairness and performance isolation by pinning tenants to CPU cores. Pyxis can reuse existing mechanisms for multi-tenancy, and the problem is orthogonal to this paper. Given a resource allocation scheme, Pyxis is able to make optimal scheduling decisions.

VIII. Related Work

Disaggregated storage. Fast kernel-bypass networking based on DPDK and RDMA enables high throughput and low latency [75], [76], [77], [78], [79] for storage systems. As a result, disaggregating compute from storage has become the paradigm for building applications in the cloud [11], [12], [13], [14], [80]. Pyxis adheres to the concept of resource disaggregation and focuses on task scheduling to improve system performance.

Storage-side computation. Pushing computation to storage is a common practice in database systems, such as SQL stored procedures [43], [44], [45] and user-defined functions (UDFs) [81], [82], [83]. Many cloud data warehouses are SQL-aware [11], [16] or optimized for in-place analytical processing [17]. A recent line of research considers pushing more fine-grained computation to storage using custom user-defined extensions [18], [84], [85], [86]. Pyxis is based on the exposed compute capability on storage and optimizes scheduling decisions for better utilization.

Task scheduling in disaggregated datacenters. Recent trend on resource disaggregation inspires several studies on workload-aware scheduling [22], [23], [87], [88]. ASFP [23] adopts a reactive approach that first schedules all tasks to storage and then pushes some of them to compute nodes when overloaded. Kayak [22] improve ASFP by proactively finding the split ratio. However, Kayak is designed for homogeneous tasks and cannot directly extend to mixed workloads. Pyxis addresses the multi-task problem by reducing the solution space to a single turning point.

Stateful Serverless Computing. The disaggregation between compute and storage also lies in the foundation of serverless computing. Provisioning separate compute and storage pools for stateless functions and storage services respectively enables the hallmark feature of fine-grained autoscaling and pay-per-use billing in serverless. However, stateful serverless functions that involve many data accesses are faced with the same problem as IO-intensive tasks in Pyxis. Several studies focus on optimizing state sharing or data passing between stateful functions [33], [89], [90], [91], [92], [93]. Cloudburst colocates a mutable cache

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

with functions for fast state accesses [35]. Faa$t proposes to place a pre-warmed autoscaling cache alongside each function [94]. Locus judiciously provisions fast in-memory storage or slow but cheap disk-based servers for efficient serverless analytics [95]. Pyxis adopts a similar setting as Locus where the cluster contains disaggregated remote storage (or cache) servers, and allows in-storage data processing by selectively scheduling a subset of tasks to storage.

IX. Conclusion

We present Pyxis, a system that provides optimal scheduling decisions for mixed workloads in disaggregated datacenters. Pyxis maximizes overall throughput while meeting latency SLOs. Our key insight is that the optimal solution has an "all-or-nothing" structure with a single turning point, which greatly simplifies the solution space. We propose an online algorithm that finds the turning point efficiently, and provide theoretical guarantees for its convergence. Our system prototype supports scalability and elasticity. Extensive experiments show that Pyxis achieves sub-second convergence and improves overall throughput by 3–21× over the state-of-the-art solution.

References

[1] J. Ros-Giralt et al., "Designing data center networks using bottleneck structures," in *Proc. ACM SIGCOMM Conf.*, 2021, pp. 319–348.
[2] Y. Zhang et al., "Aequitas: Admission control for performance-critical RPCs in datacenters," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 1–18.
[3] V. Addanki, M. Apostolaki, M. Ghobadi, S. Schmid, and L. Vanbever, "Abm: Active buffer management in datacenters," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 36–52.
[4] A. Singhvi et al., "1rma: Re-envisioning remote memory access for multi-tenant datacenters," in *Proc. ACM SIGCOMM Conf.*, 2020, pp. 708–721.
[5] S. Yan, X. Wang, X. Zheng, Y. Xia, D. Liu, and W. Deng, "ACC: Automatic ECN tuning for high-speed datacenter networks," in *Proc. ACM SIGCOMM Conf.*, 2021, pp. 384–397.
[6] G. Kumar et al., "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proc. ACM SIGCOMM Conf.*, 2020, pp. 514–528.
[7] S. S. Chen, W. Wang, C. Canel, S. Seshan, A. C. Snoeren, and P. Steenkiste, "Time-division TCP for reconfigurable data center networks," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 19–35.
[8] W. Reda, M. Canini, D. Kostić, and S. Peter, "{RDMA} is turing complete, we just did not know it yet!," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 71–85.
[9] V. Addanki, O. Michel, and S. Schmid, "PowerTCP: Pushing the performance limits of datacenter networks," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 51–70.
[10] Q. Cai, M. T. Arashloo, and R. Agarwal, "dcPIM: Near-optimal proactive datacenter transport," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 53–65.
[11] "Amazon simple storage service (S3)." Accessed: Jun. 14, 2024. [Online]. Available: https://aws.amazon.com/s3/
[12] "Microsoft azure blob storage." Accessed: Jun. 14, 2024. [Online]. Available: https://azure.microsoft.com/services/storage/blobs/
[13] "Google cloud storage." Accessed: Jun. 14, 2024. [Online]. Available: https://cloud.google.com/storage
[14] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes, "Building an elastic query engine on disaggregated storage," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation*, 2020, pp. 449–462.
[15] "Amazon S3 select command." Accessed: Jun. 14, 2024. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-glacier-select-sql-reference-select.html
[16] "Amazon redshift." Accessed: Jun. 14, 2024. [Online]. Available: https://aws.amazon.com/redshift/
[17] "Aqua (advanced query accelerator) for amazon redshift." Accessed: Jun. 14, 2024. [Online]. Available: https://aws.amazon.com/redshift/features/aqua/
[18] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman, "Splinter: Bare-metal extensions for multi-tenant low-latency storage," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 627–643.
[19] D. Kim et al., "Hyperloop: Group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems," in *Proc. ACM SIGCOMM Conf.*, 2018, pp. 297–312.
[20] Y. Zhong et al., "BPF for storage: An exokernel-inspired approach," in *Proc. Workshop Hot Topics Operating Syst.*, 2021, pp. 128–135.
[21] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter {RPCs} can be general and fast," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 1–16.
[22] J. You, J. Wu, X. Jin, and M. Chowdhury, "Ship compute or ship data? why not both?," in *Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 633–651.
[23] A. Bhardwaj, C. Kulkarni, and R. Stutsman, "Adaptive placement for in-memory storage functions," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 127–141.
[24] "Cloudlab." Accessed: Jun. 14, 2024. [Online]. Available: https://cloudlab.us/
[25] "Decomposing Twitter: Adventures in service-oriented architecture." Accessed: Jun. 14, 2024. [Online]. Available: https://www.infoq.com/presentations/twitter-soa/
[26] H. Zhou et al., "Overload control for scaling wechat microservices," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 149–161.
[27] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-based and QoS-aware resource management for cloud microservices," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 167–181.
[28] "AWS lambda." Accessed: Jun. 14, 2024. [Online]. Available: https://aws.amazon.com/lambda/
[29] "Knative." Accessed: Jun. 14, 2024. [Online]. Available: https://knative.dev/docs/
[30] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with Open-Lambda," in *Proc. 8th USENIX Workshop Hot Topics Cloud Comput.*, 2016, pp. 33–39.
[31] S. Qi, L. Monis, Z. Zeng, I.-C. Wang, and K. Ramakrishnan, "SPRIGHT: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 780–794.
[32] "Azure functions." Accessed: Jun. 14, 2024. [Online]. Available: https://azure.microsoft.com/en-us/products/functions
[33] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 427–444.
[34] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica, "Jiffy: Elastic far-memory for stateful serverless analytics," in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 697–713.
[35] V. Sreekanti et al., "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2438–2452, 2020.
[36] S. Qi, X. Liu, and X. Jin, "Halfmoon: Log-optimal fault-tolerant stateful serverless computing," in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 314–330.
[37] "Functionbench." Accessed: Jun. 14, 2024. [Online]. Available: https://github.com/kmu-bigdata/serverless-faas-workbench
[38] "Serverlessbench." Accessed: Jun. 14, 2024. [Online]. Available: https://serverlessbench.systems/en-us/
[39] "Serverless examples," Accessed: Jun. 14, 2024. [Online]. Available: https://github.com/serverless/examples
[40] "Deathstarbench." Accessed: Jun. 14, 2024. [Online]. Available: https://github.com/delimitrou/DeathStarBench/
[41] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proc. ACM SIGCOMM Conf.*, 2014, pp. 455–466.
[42] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 81–97.
[43] "Microsoft SQL server stored procedures." Accessed: Jun. 14, 2024. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine/

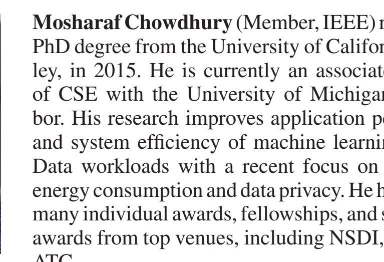
Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

[44] "MySQL stored procedures tutorial." Accessed: Jun. 14, 2024. [Online]. Available: https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx/
[45] "Oracle PL/SQL." Accessed: Jun. 14, 2024. [Online]. Available: https://www.oracle.com/database/technologies/application-development-PL/SQL.html/
[46] C. Bah–Ming and J. Raj, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Comput. Netw. ISDN Syst.*, vol. 1, 1939, Art. no. 1414.
[47] L. Kleinrock, *Queuing Systems, Volume II: Computer Applications*. Wiley, 1976.
[48] T. J. Ott, "The sojourn-time distribution in the M/G/1 queue by processor sharing," *J. Appl. Probability*, vol. 21, no. 2, pp. 360–378, 1984.
[49] A. Daglis, M. Sutherland, and B. Falsafi, "RPCValet: Ni-driven tail-aware balancing of μs-scale RPCs," in *Proc. 24th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 35–48.
[50] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving low tail latency for microsecond-scale networked tasks," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 325–341.
[51] "Pyxis: Scheduling mixed tasks in disaggregated datacenters (appendix)." Accessed: Jun. 14, 2024. [Online]. Available: https://tomquartz.github.io/files/TPDS24_Pyxis_appendix.pdf
[52] M. Doraghinejad, H. Nezamabadi-pour, A. H. Sadeghian, and M. Maghfoori, "A hybrid algorithm based on gravitational search algorithm for unimodal optimization," in *Proc. 2nd Int. eConf. Comput. Knowl. Eng.*, 2012, pp. 129–132.
[53] V. Kumar, J. K. Chhabra, and D. Kumar, "Parameter adaptive harmony search algorithm for unimodal and multimodal optimization problems," *J. Comput. Sci.*, vol. 5, no. 2, pp. 144–155, 2014.
[54] F. Merrikh-Bayat, "The runner-root algorithm: A metaheuristic for solving unimodal and multimodal optimization problems inspired by runners and roots of plants in nature," *Appl. Soft Comput.*, vol. 33, pp. 292–303, 2015.
[55] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
[56] N. Bronson et al., "{TAO }:{ Facebook's} distributed data store for the social graph," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 49–60.
[57] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2055–2063.
[58] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proc. 10th ACM Conf. Recommender Syst.*, 2016, pp. 191–198.
[59] M. Grbovic and H. Cheng, "Real-time personalization using embeddings for search ranking at airbnb," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 311–320.
[60] J. Wang et al., "Milvus: A purpose-built vector data management system," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 2614–2627.
[61] C. Wei et al., "AnalyticDB-V: A hybrid analytical engine towards query fusion for structured and unstructured data," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3152–3165, 2020.
[62] W. Yang, T. Li, G. Fang, and H. Wei, "PASE: PostgreSQL ultra-high-dimensional approximate nearest neighbor search extension," in *Proc. ACM SIGMOD Int Conf. Manage. Data*, 2020, pp. 2241–2253.
[63] J. Li et al., "The design and implementation of a real time visual search system on JD E-commerce platform," in *Proc. 19th Int. Middleware Conf. Ind.*, 2018, pp. 9–16.
[64] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Trans. Big Data*, vol. 7, no. 3, pp. 535–547, Jul. 2021.
[65] C. Percival, "Stronger key derivation via sequential memory-hard functions." Accessed: Jun. 14, 2024. [Online]. Available: http://www.tarsnap.com/scrypt/scrypt.pdf
[66] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "{Information-Agnostic} flow scheduling for commodity data centers," in *Proc. 12th USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 455–468.
[67] "PostgreSQL: Documentation: 14: 51.5. planner/optimizer." Accessed: Jun. 14, 2024. [Online]. Available: https://www.postgresql.org/docs/current/planner-optimizer.html
[68] M. Armbrust et al., "Spark SQL: Relational data processing in spark," in *Proc. Int. Conf. Manage. Data*, 2015, pp. 1383–1394.
[69] Y. Azar, I. R. Cohen, S. Kamara, and B. Shepherd, "Tight bounds for online vector bin packing," in *Proc. 45th Annu. ACM Symp. Theory Comput.*, 2013, pp. 961–970.
[70] G. J. Woeginger, "There is no asymptotic ptas for two-dimensional vector packing," *Inf. Process. Lett.*, vol. 64, no. 6, pp. 293–297, 1997.
[71] "Heuristics for vector bin packing." Accessed: Jun. 14, 2024. [Online]. Available: https://www.microsoft.com/en-us/research/publication/heuristics-for-vector-bin-packing/
[72] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs," in *Proc. 16th USENIX Symp. Operating Syst. Des. Impl.*, 2022, pp. 303–320.
[73] A. Mahgoub et al., "Wisefuse: Workload characterization and dag transformation for serverless workflows," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 2, pp. 26:1–26:28, 2022.
[74] C. Jin et al., "Ditto: Efficient serverless analytics with elastic parallelism," in *Proc. ACM SIGCOMM Conf.*, 2023, pp. 406–419.
[75] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 401–414.
[76] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 185–201.
[77] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 429–444.
[78] J. Ousterhout et al., "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 1–55, 2015.
[79] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using rdma and htm," in *Proc. 25th Symp. Operating Syst. Princ.*, 2015, pp. 87–104.
[80] "Amazon aurora serverless." Accessed: Jun. 14, 2024. [Online]. Available: https://aws.amazon.com/rds/aurora/serverless/
[81] "Microsoft SQL server user-defined functions." Accessed: Jun. 14, 2024. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions/
[82] "Postgresql: Documentation: 8.0: User-defined functions." Accessed: Jun. 14, 2024. [Online]. Available: https://www.postgresql.org/docs/8.0/xfunc.html
[83] "UDFs (user-defined functions) – snowflake documentation." Accessed: Jun. 14, 2024. [Online]. Available: https://docs.snowflake.com/en/sql-reference/user-defined-functions.html
[84] "Redis." Accessed: Jun. 14, 2024. [Online]. Available: https://redis.io/
[85] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, "Comet: An active distributed key-value store," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 323–336.
[86] M. A. Sevilla et al., "Malacology: A programmable storage system," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 175–190.
[87] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li, "Balancing CPU and network in the cell distributed B-tree store," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 451–464.
[88] S. Novakovic et al., "Storm: A fast transactional dataplane for remote data structures," in *Proc. 12th ACM Int. Conf. Syst. Storage*, 2019, pp. 97–108.
[89] Z. Li et al., "Faasflow: Enable efficient workflow execution for function-as-a-service," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 782–796.
[90] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 805–820.
[91] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful serverless computing with crucial," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, pp. 39:1–39:38, 2022.
[92] M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos, "Distributed transactions on serverless stateful functions," in *Proc. ACM Int. Conf. Distrib. Event-Based Syst.*, 2021, pp. 31–42.
[93] S. Burckhardt et al., "Netherite: Efficient execution of serverless workflows," *Proc. VLDB Endowment*, vol. 15, no. 8, pp. 1591–1604, 2022.
[94] F. Romero et al., "Faat: A transparent auto-scaling cache for serverless applications," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 122–137.
[95] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proc. 16st USENIX Symp. Netw. Sys. Des. Impl.*, 2019, pp. 193–206.

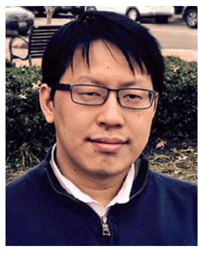
Sheng Qi received the BS degree from Peking University, in 2022. He is currently working toward the PhD degree with the School of Computer Science, Peking University. He is currently working on state management of serverless functions and microservices. His research interests include building fault-tolerant and high-performance distributed and disaggregated systems.

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.

Chao Jin received the BS degree in 2023 from Peking University, where he is currently working toward the PhD degree with the School of Computer Science. He is currently working on serverless workflow orchestration. His research interests include cloud computing and machine learning systems.



Mosharaf Chowdhury (Member, IEEE) received the PhD degree from the University of California, Berkeley, in 2015. He is currently an associate professor of CSE with the University of Michigan, Ann Arbor. His research improves application performance and system efficiency of machine learning and Big Data workloads with a recent focus on optimizing energy consumption and data privacy. He has received many individual awards, fellowships, and seven paper awards from top venues, including NSDI, OSDI, and ATC.



Zhenming Liu received the PhD degree from Harvard University, in 2012. He is currently an assistant professor in computer science with the College of William & Mary. His primary research interest is in the interplay between theory and machine learning. He has received two NSF grants and best papers from FAST 2019, SDM 2019, Infocom 2015, and PKDD 2010.

Xuanzhe Liu (Senior Member, IEEE) received the PhD degree from Peking University, in 2009. He is currently an associate professor (with Tenure) with the School of Computer Science, Peking University. He is an ACM distinguished scientist and a distinguished member of the CCF. His research interests mainly fall in service-based software engineering and systems. His efforts have been published at conferences including WWW, ICSE, FSE, SIGCOMM, NSDI, MobiCom, MobiSys, SIGMETRICS, and IMC, and in journals including *ACM Transactions on Software Engineering and Methodology*/*ACM Transactions on Information Systems*/*ACM Transactions on Internet Technology*/*ACM Transactions on the Web* and *IEEE Transactions on Software Engineering*/*IEEE Transactions on Mobile Computing*/*IEEE Transactions on Services Computing*.

Xin Jin (Senior Member, IEEE) received the PhD degree from Princeton University, in 2016. He is currently an associate professor (with Tenure) with the School of Computer Science, Peking University. His research interests include computer systems, networking, and cloud computing. He received USENIX FAST Best Paper Award (2019) and USENIX NSDI Best Paper Award (2018).

Authorized licensed use limited to: Peking University. Downloaded on July 18,2024 at 08:35:15 UTC from IEEE Xplore. Restrictions apply.