Ship Compute or Ship Data? Why Not Both?

Jie You*, Jingfeng Wu◇, Xin Jin†, and Mosharaf Chowdhury*
*University of Michigan, ◇Johns Hopkins University, †Peking University

Abstract

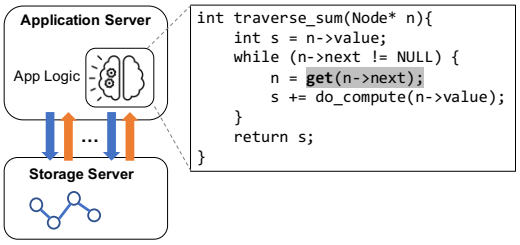
How cloud applications should interact with their data remains an active area of research. Over the last decade, many have suggested relying on a key-value (KV) interface to interact with data stored in remote storage servers, while others have vouched for the benefits of using remote procedure call (RPC). Instead of choosing one over the other, in this paper, we observe that an ideal solution must adaptively combine both of them in order to maximize throughput while meeting application latency requirements. To this end, we propose a new system called Kayak that proactively adjusts the rate of requests and the fraction of requests to be executed using RPC or KV, all in a fully decentralized and self-regulated manner. We theoretically prove that Kayak can quickly converge to the optimal parameters. We implement a system prototype of Kayak. Our evaluations show that Kayak achieves sub-second convergence and improves overall throughput by 32.5%-63.4% for compute-intensive workloads and up to 12.2% for non-compute-intensive and transactional workloads over the state-of-the-art.

1 Introduction

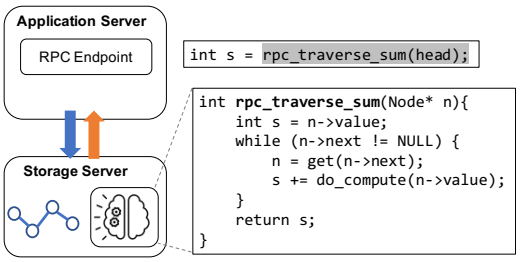
Two trends stand out amid the rapid changes in the landscape of cloud infrastructure in recent years:

- First, with cloud networks moving from 1Gbps and a few hundred μs to 100Gbps and single-digit μs [11, 20, 25], disaggregated storage has become the norm [4, 17, 21, 28, 31, 40, 44]. It decouples compute from storage, enabling flexible provisioning, elastic scaling, and higher utilization. As a result, increasingly more applications now access their storage servers over the network using a key-value (KV) interface.
- Second, the steady increase in compute granularity, from virtual machines, to containers, to microservices and serverless functions, is popularizing storage-side computation using remote procedure calls (RPCs) [23]. Many databases allow stored procedures and user-defined functions [5–8, 26, 41, 42], and some KV stores allow just-in-time or pre-compiled runtime extensions [9, 18, 29, 39].

The confluence of these two contradicting trends – the former moves data to compute, while the latter does the opposite – highlights a long-standing challenge in distributed systems: *should we ship compute to data, or ship data to compute?*



(a) *Ship data to compute.*



(b) *Ship compute to data.*

Figure 1: ***Graph traversal implemented with (a) disaggregated storage and (b) storage-side compute. The latter (1b) results in less network round-trips but exert more load on the storage server.***

The answer, in broad strokes, boils down to the ratio of computation and communication. The benefits of storage disaggregation, i.e., shipping data to compute, typically holds when most of the time of a function invocation (hereafter referred to as a request) is spent in computation. However, when a single request triggers multiple dependent accesses to the disaggregated storage, time spent in network traversals and data (un)marshalling starts to dominate [10]. Figure 1a shows an example of a simple graph traversal algorithm implemented on top of disaggregated storage, where "pointer chasing" can make network traversals the bottleneck.

In contrast, storage-side computation enables applications to offload part of their application logic to storage servers. The storage layer is customized to support application-specific RPC-style APIs [10, 12, 29], which shave off network round-trips. Figure 1b shows the previous example implemented with storage-side computing, where only one network round-trip is sufficient. However, this is not universally viable either; for compute-intensive workloads, the compute capacity of the storage servers can become the bottleneck when too much computation is offloaded to the storage.

In short, there is no one-size-fits-all solution. Existing

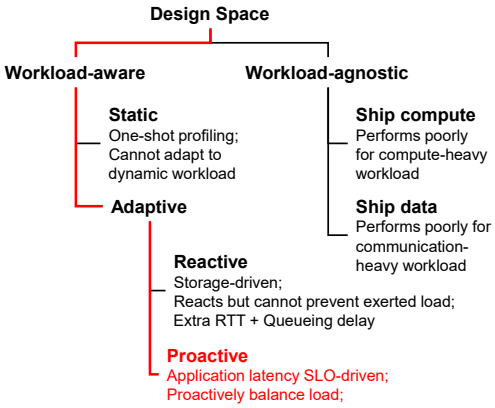


Figure 2: ***Design space of Kayak.***

works have explored different points in the design space (Figure 2). Arbitrarily forcing a disaggregated storage (i.e., ship data) or a storage-side computation architecture (i.e., ship compute) in a *workload-agnostic* manner leads to poor utilization and low throughput. Our measurements show that *workload-agnostic* solution leads to up to 58% lower throughput and 37% lower utilization when compared to the optimal. For *workload-aware* alternatives, one choice is taking a one-shot *static* approach, whereby a workload is profiled once at the beginning. However, statically choosing either KV- or RPC-based approach falls short even for a single tenant (§2.2).

The alternative, therefore, is taking an *adaptive* approach that can dynamically choose between shipping data and shipping compute. Existing adaptive solutions such as ASFP [12] take a *reactive* approach: all requests are forwarded using RPC to the storage server, which can then react by pushing some of them back. While this provides a centralized point of control, each request experiences non-zero server-side queueing delay, and more importantly, requests that are pushed back suffer from one *extra* round-trip time (RTT), which is detrimental to low-latency applications with strict latency SLOs. Moreover, throughput-driven designs cannot proactively throttle exerted load on the storage server w.r.t. tail latency service-level objectives (SLOs).

We observe that an ideal solution fundamentally calls for a *balanced* architecture that can effectively utilize the available resources and increase overall throughput while satisfying tail latency SLOs. In this paper, we present Kayak that takes a *proactive* adaptive approach to achieve these goals (§3). In order to maximize throughput without SLO violations, Kayak proactively decides between shipping compute and shipping data when executing incoming requests, and it throttles request rate in order to meet SLO requirements. Specifically, Kayak takes a latency-driven approach to optimize two parameters simultaneously: (1) the *request rate*, and (2) the *RPC fraction*, which denotes the proportion of the incoming requests to be executed using RPC.

Unfortunately, the optimal RPC fraction varies for different workloads and their SLO requirements. There is no closed-

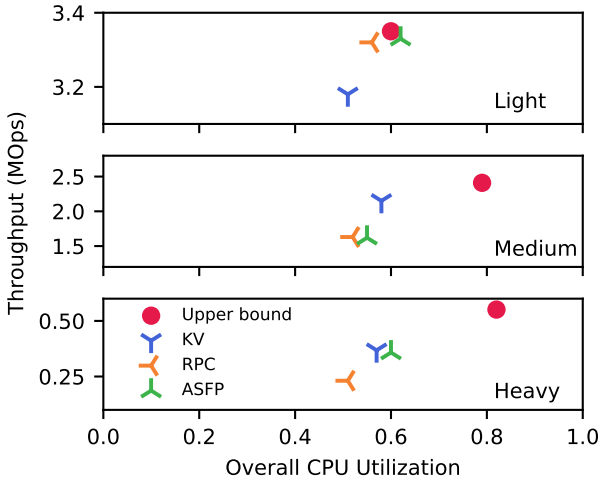


Figure 3: ***Throughput and overall request-level CPU utilization across both application and storage servers for three different workloads under different execution schemes, with 200μs SLO.***

form expression to precisely capture the relationship between RPC fraction, request rate, and tail latency either. Finally, we show that the order in which we optimize request rate and RPC fraction affects convergence of the optimization algorithm. We address these challenges by designing a dynamic optimization method using a dual loop control (§4). Kayak employs a faster control loop to optimize request rate and a slower one to optimize RPC fraction. Combined together, Kayak iteratively searches for the optimal parameters, with a provable convergence guarantee. In addition to increasing throughput in the single-tenant scenario, Kayak must also ensure fairness and work conservation of shared server resources in multi-tenant settings. Kayak pins tenants to CPU cores in a fair manner and employs work stealing to achieve work conservation.

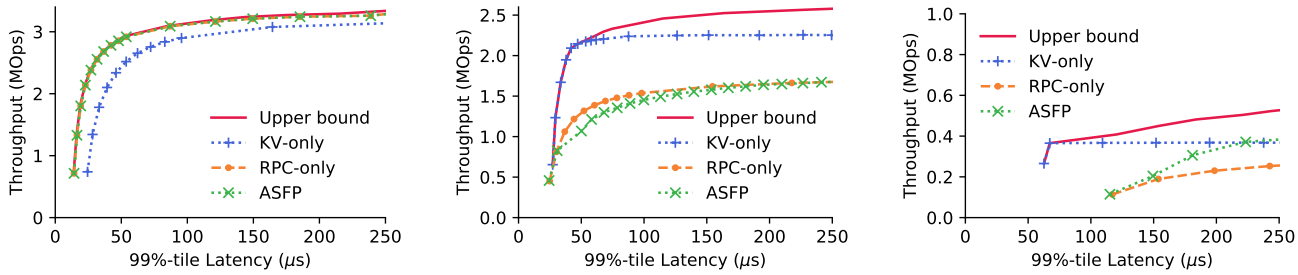
Our evaluation on a prototype of Kayak shows that: (1) Kayak achieves sub-second convergence to optimal throughput and RPC fraction regardless of workloads; (2) Kayak improves overall throughput by 32.5%-63.4% for compute-intensive workloads and up to 12.2% for non-compute-intensive and transactional workloads; and (3) in a multi-tenant setup, Kayak approximates max-min fair sharing and scales without sacrificing fairness.

2 Motivation

2.1 Limitations of Existing Designs

Existing solutions either fail to efficiently utilize the available CPU cores for a large variety of workloads or introduce additional overhead to reactively adapt to workload variations, both of which lead to lower throughput.

Workload-agnostic approaches either use KV-only design or RPC-only design. The former results in excessive network round-trips of storage access during execution, while the latter overloads the storage server CPU and leaves the CPU on the



(a) *Light (100ns/RTT)* (b) *Medium (1μs/RTT)* (c) *Heavy (10μs/RTT)*

Figure 4: ***Throughput w.r.t. SLO (99%-tile latency) for 3 different workloads under different execution schemes.***

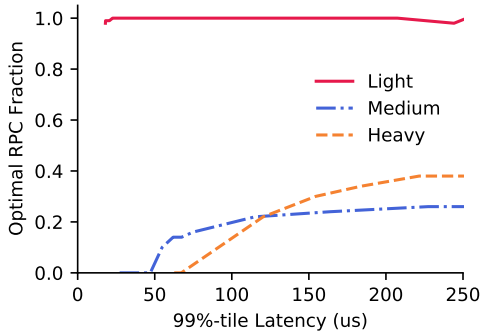


Figure 5: ***Optimal RPC fraction w.r.t. SLO (99%-tile latency) for 3 different workloads.***

application server underutilized. In either case, the overall CPU utilization is low which hinders the performance.

ASFP [12] presents an alternative to workload-agnostic solutions by taking a workload-aware approach with runtime adaptation. In that design, the requests are executed using RPC by default, and if the storage server gets overloaded by the exerted computation, a pushback mechanism is triggered to push the exerted computation back to the application server side. However, the excessive queueing on the storage server still cannot be prevented, although it can be alleviated by the pushback mechanism. Furthermore, the execution of pushed-back requests needs to restart on the application server, wasting CPU cycles on both servers.

To illustrate these issues, we perform an experiment with the graph traversal application shown in Figure 1. We configure the workload so that the each request triggers two storage accesses (i.e., two network round-trips) when executed using the KV scheme. We vary the computation after each access and refer to them as Light (100ns computation time per access), Medium (1μs per access) and Heavy (10μs per access). For reference, adding one network round-trip incurs 9.2μs more latency for each request in our testing environment. We measure the maximum achievable throughput and the CPU utilization (defined as CPU cycles spent only in executing the requests). As shown in Figure 3, using only KV or RPC leads to lower overall CPU utilization and lower throughput. Although the reactive adaptive design utilizes a higher amount of CPU on both application and storage servers, its overheads add up quickly, and its CPU usage for request-level computation does not increase significantly.

To summarize, existing designs do not efficiently utilize the CPU resource on both application and storage servers, which limits their performance. There exists an optimal RPC fraction that maximizes the throughput (calculated by a comprehensive sweep as explained below).

2.2 Need for Dynamically Finding the Optimal Fraction

A key challenge here is that this optimal fraction varies for different workloads. To highlight this, we perform another experiment with the same graph traversal workload as before. We configure the application server to handle a fraction of the requests using RPC and the rest using the KV approach. We vary the RPC fraction from 0 to 1 and measure the overall throughput and end-to-end latency of all requests. In doing so, we obtain the throughput-latency measurements for all possible execution configurations, as shown in Figure 4 and Figure 5. The *upper bound* in Figure 4 is defined by selecting the best RPC fraction for each latency SLO to maximize the throughput, and the selected fraction is plotted in Figure 5. ASFP is configured as using RPC by default with pushback enabled.

As shown in Figure 4, workload with less computation favors RPC over KV, and vice versa. For Medium and Heavy workloads, the highest throughput is achieved by combining RPC and KV together. Moreover, we observe that, (1) for different workloads, the highest throughput is achieved with different RPC fraction; and (2) for different SLO constraints, the optimal fraction for highest throughput is also different. This calls for a proactive design to search for the optimal RPC fraction. Note that we repeated the same parameter sweep with the number of storage accesses per request set to 4 and 8, and observe similar trends (please refer to Appendix A).

3 Kayak Overview

Kayak proactively decides between *shipping compute* and *shipping data* in a workload-adaptive manner. It arbitrates

incoming requests and proactively decides the optimal RPC fraction (i.e., what fraction of the compute to ship to storage servers) of executing the requests while meeting end-to-end tail latency SLO constraints.

3.1 Design Goals

Kayak aims to meet the following design goals.

- *Maximize throughput without SLO violations:* Applications have stringent tail latency SLO constraints to ensure low user-perceived latency. Kayak should maximize the throughput while not violating these SLO constraints.
- *High CPU utilization across all servers:* Kayak should balance the computation imposed by application logic across the application and storage servers.
- *Fair sharing of storage server resources:* In a multi-tenant cloud, multiple applications may be sending requests to the same storage server, which should be fairly shared.
- *Ease of deployment:* Applications should be able to use Kayak with minimal code modification.

3.2 Architectural Overview

At a high level, Kayak adaptively runs application logic on both the application and storage servers (Figure 6). However, even though it ships some computation to the storage server, Kayak's core control logic runs only on the application server and the storage server acts as an extended executor. Unlike existing reactive solutions, Kayak proactively decides the amount of computation to ship to the storage side.

Key ideas. Essentially, Kayak finds the optimal RPC fraction and maximizes throughput at runtime while meeting tail latency SLO constraints. The main design challenge is that the optimal fraction varies in accordance with different workloads and their SLO requirements (Figure 5), as well as the amount of load exerted on the storage server. In a multi-tenant cloud, all of these change dynamically. In order to keep up with the changing environment, Kayak proactively adjust the RPC fraction and the request rate according to realtime tail latency measurements.

However, the relationship between the request rate, RPC fraction and latency cannot be easily captured with a closed-form expression. Thus Kayak adopts a numerical optimization method based on a dual loop control algorithm (§4.3) to search for the optimal parameters iteratively. We notice that latency measurements in real systems exhibit large variance, which detrimentally impacts the performance of such iterative optimization algorithms. Kayak's algorithm accounts for such variance and has a provable convergence guarantee.

From an implementation perspective, Kayak faces another challenge as it has to make adjustments very quickly due to the high-throughput, low-latency nature of the environment. Kayak cannot afford to gather global information and make centralized decisions. Hence, the algorithm for Kayak is fully decentralized and runs only on the application servers.

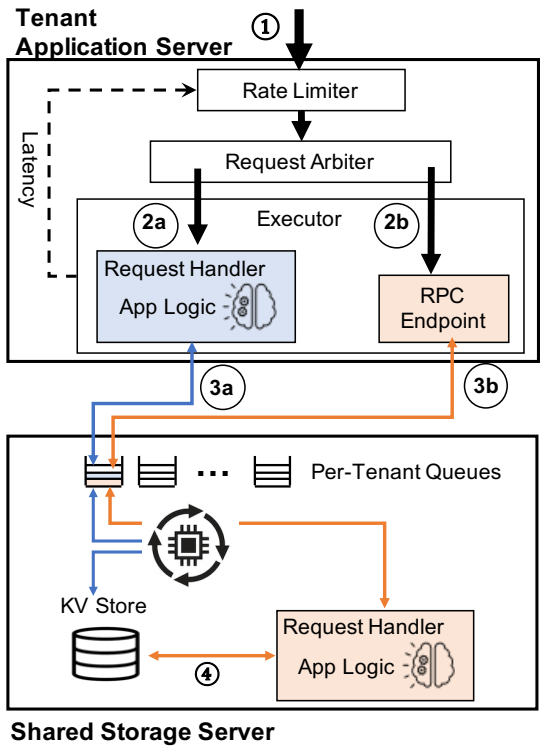


Figure 6: ***Kayak architecture. Tenant application servers interact with shared storage servers using Kayak, which proactively decides which requests to run on the application server using KV operations and which ones to ship to the storage server via RPC.***

Application server. The application server consists of three main components (Figure 6): (i) the Rate Limiter limits the rate of incoming requests from the tenant to satisfy SLO constraints; (ii) the Request Arbiter determines the optimal RPC fraction; and (iii) the Executor handles the requests according to the scheme decided by the Request Arbiter.

The Rate Limiter interacts with the tenants and ① receives incoming requests. It continuously monitors real-time tail latency of end-to-end request execution, and pushes back to signal the tenant to slow down. When the servers are overloaded and the SLO cannot be met, it drops overflowing requests. We assume that tenants implement mechanism to handle overflowing, such as resubmit dropped requests, and increase provisioning so that in the worst case scenario all requests can still be executed on all application servers within the SLO.

The Request Arbiter proactively determines the optimal RPC fraction. It selects the execution scheme for each request based on that fraction. For each request, the choice of execution scheme is determined by a Bernoulli distribution $B(1,X)$ where X is the proportion of requests processed using RPC.

The Executor handles the request in its entirety and reports the end-to-end completion time back to the Rate Limiter upon completion. It consists of two parts: (i) the Request Handler and (ii) the RPC Endpoint. If the request is to be executed using ②a the KV scheme, the Request Handler is triggered.

The Request Handler executes the application logic locally on the application server and keeps track of the states of the request. Whenever it needs to access data stored in the storage server, ③a the KV API of the storage server is subsequently called. In contrast, if the request is to be executed on the storage side using the RPC scheme, then the request is simply forwarded to ②b the RPC Endpoint on the application server. The RPC Endpoint issues an RPC request ③b to the storage server for processing the request.

Storage server. The storage server includes an additional Request Handler to handle RPC requests in addition to a KV interface. Similar to allocating CPU cores in the application server to run application code, in Kayak, computation resources in the storage server are also allocated to specific tenants at the CPU core granularity.

Each tenant has a dedicated request queue, from which its core(s) polls KV and RPC requests. Handling an incoming KV request in Kayak is the same as what happens in a traditional KV store: the request is simply forwarded to the KV store. Upon receiving an RPC request, the Request Handler is triggered and executes the application logic on the storage server. The Request Handler calls ④ the local KV API whenever data access is needed, interacting with the stored data without crossing the network.

This static *pin-request-to-core* allocation scheme of Kayak makes it easier to enforce fair computation resource sharing between tenants. However, static allocation of CPU cores cannot guarantee work conservation of the CPU cores on the storage server. Kayak uses *work stealing* to mitigate this issue: whenever a tenant's dedicated queue is empty, the corresponding CPU core *steals* requests from other queues.

4 Kayak Design

Our primary objective is to maximize the total throughput without violating the tail latency SLO. However, higher throughput inevitably leads to higher latency in a finite system [27], and there exists a fundamental tradeoff between throughput and latency. Unfortunately, the precise relationship between latency and throughput of a real system, however, is notoriously difficult to be captured by a closed-form expression. In this paper, we use an analytical model to highlight our insights and take a tail latency measurement-driven approach to design a pragmatic solution.

At the same time, as illustrated in Section 2, a reactive approach to achieve this can lead to CPU wastage. Hence, Kayak proactively decides what fraction of the requests to offload vs. which ones to run in the application server, while maximizing the total throughput within the SLO constraint. The need for optimizing both raises a natural question: *which one to optimize first?* In this section, we analyze both optimization orders and design a dual loop control algorithm with provable convergence guarantees. Detailed proofs can be found in the appendix.

Sym.	Description
R	Total request rate
X	Proportion of requests processed using RPC
τ	Random variable of request latency
t_o	Latency SLO target
$T(X,R)$	Latency SLO as a function of X, R
$R(X)$	Function implicitly defined by $T(X,R(X)) = t_0$
k	Index of iterations

Table 1: ***Key notations in problem formulation.***

4.1 Problem Formulation

We denote the proportion of requests to be executed using RPC by X, the total incoming request rate by R, and we define τ as the random variable of request latency, thus we have:

$$\tau \sim P(R,X),$$

where R and X are the parameters of distribution P. Table 1 includes the key notations used in this paper.

We denote $T(X,R)$ as our SLO statistics metric, which takes a specific statistical interpretation for the particular SLO metric. For instance, if the SLO is defined as the 99%-tile latency then T is the 99%-tile for τ. We denote t_0 as the SLO target under the same statistic metric. Thus the problem can be formulated as:

$$\max_X \quad R \tag{1}$$
$$\text{s.t.} \quad T(X,R) \le t_0 \tag{2}$$
$$R > 0, \tag{3}$$
$$X \in [0,1]. \tag{4}$$

Here constraint (2) captures the latency SLO constraint, and constraints (3) and (4) represents the boundary of R and X, respectively.

We make the following observation when solving this optimization problem:

Observation 1. *Fixing X, $F_X(R) := T(X,R)$ is monotonic increasing.*

Observation 1 captures the relationship of throughput and latency from queueing theory [27] for finite systems like Kayak.

4.2 Strawman: X-R Dual Loop Control

Optimization (1) cannot be directly solved with a closed-form solution of R and X due to the intractability of the function $T(X,R)$. Therefore, we use a numeric optimization method and try to optimize R and X independently and iteratively. To put it into our context, we need to design an iterative algorithm such that in each iteration, we first optimize either R or X, and then optimize the other. We also have to prove that this algorithm would actually converge to ensure optimality and stability of the system.

Now we are facing a question: *which one to optimize first?* In our problem, there is an asymmetry for X and R: X is the

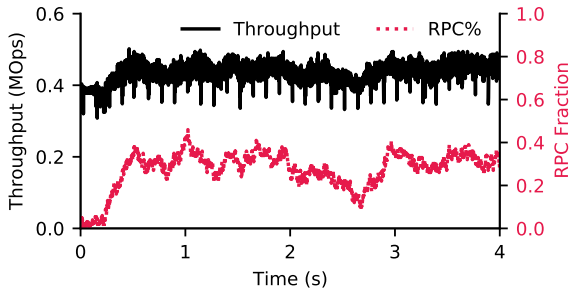


Figure 7: ***Instability of throughput and RPC fraction w.r.t time, with X-R Dual Loop Control.***

parameter in Optimization (1) where as R is the objective. A straightforward solution is to optimize X first, which leads to the following algorithm.

Algorithm I (X-R Dual Loop Control) For $k = 1, \ldots, K$, we alternatively update X_k and R_k by

1. Fix R_k, and find the RPC fraction X_k that minimizes latency, i.e., $X_k = \arg\min_X T(X, R_k)$;

2. Update R_k according to *gradient descent* so that $T \approx t_0$, i.e.,
$$R_{k+1} = R_k + \eta\left(t_0 - T(X_k, R_k)\right),$$
where $\eta > 0$ is the stepsize.

In this algorithm, the first step is to solve a convex optimization problem. Because our assumptions guarantee that X_k is unique and finite, this iteration is well defined for at least the first step (and we will show it is also good for the second step). Moreover, because $T_{R_k}(X) := T(X, R_k)$ is μ-strongly convex and L-smooth, X_k can be solved very quickly (or mathematically, in a linear rate) by iterative algorithms such as gradient descent. We have the following theorem that characterizes the convergence of this algorithm. The rigorous statements and proofs are deferred to Appendix B.2.

Theorem 1. *Fixing R, $F_R(X) := T(X, R)$ is strongly convex and smooth. Suppose for all X, $0 < \alpha \le \frac{\partial T(X,R)}{\partial R} \le \beta$. Let $0 < \eta < \frac{1}{\beta}$, then under mild additional assumptions*[1]*,*

$$|R_K - R_*| \le (1 - \eta\alpha)^K \cdot |R_0 - R_*|.$$

Here R_* denotes the optimal request throughput, and R_0 denotes the initialization. This result shows the iteration of X-R Dual Loop Control converges to the optimal requests exponentially fast, i.e., after at most $O(\log \frac{1}{\varepsilon})$ iterations, the algorithm outputs a solution that is ε-close to the optimal.

Instability of X-R dual loop control. However, while Algorithm I is theoretically sound, it is not practical to be implemented in a real system. The key obstacle is that the latency SLO metric cannot be *directly* obtained in practice. Instead, we can only measure a set of samples of latency τ, and then gather statistics to derive the SLO metric. Hence the derived SLO metric – be it average or 99%-tile – is only an *estimate* $\hat{T}$ based on *sampling*. While sampling might not be a problem for many systems by using a high sampling rate, it is indeed a problem for Kayak. In particular, because of the microsecond-scale workload and the real-time requirement of Kayak, the sample size for each estimate is limited. This leads to *large variance* in the estimated $\hat{T}$, which results in *degraded* convergence speed and quality.

To quantify the impact of this variance and show the gap between theory and practice, we conduct a verification experiment. We run Algorithm I with the Heavy workload from Section 2 under an SLO constraint of 200μs 99%-tile latency. Our experiment confirms the aforementioned issue of variance in SLO estimates. Figure 7 shows poor convergence quality of both the throughput and the RPC fraction.

4.3 Our Solution: R-X Dual Loop Control

A naive mitigation to counter the SLO variance is to simply use a metric that is more robust, such as average latency. But this limits the operators to only one viable SLO metric and compromises the generality of the system.

In order to solve the challenge of unstable SLO estimates, we must design an algorithm that is not sensitive to the variance of $\hat{T}$. Compared with the RPC fraction X, the request rate R has a more intuitive and better-studied interaction with latency T from extensive study in queueing theory. Specifically, we take inspiration from recent works [19, 30] showing that even with variance in latency measurements, one can still achieve rate control (i.e., optimization of the throughput) in a *stable* manner. From the starting point of latency-driven rate control, we design a dual loop control algorithm that first optimizes R and then optimizes X to numerically solve the optimization problem. The algorithm is shown as follows. The first part is latency-driven rate control, and the second is gradient ascent.

Algorithm II (R-X Dual Loop Control) For $k = 1, \ldots, K$, we respectively update X_k and R_k by

1. Apply rate control so that the latency approximates SLO, i.e., *R_k be such that $T(X_k, R_k) \approx t_0$;*

2. Use gradient ascent to search for the optimal X_k, i.e., *$X_{k+1} = X_k + \eta\frac{dR_k}{dX}$, where $T(X, R_k) = t_0$, and η is a positive stepsize.*

R loop: rate control. In order to satisfy the SLO requirement ($T \le t_0$), we need to carefully control the request rate R. Intuitively, too high an R leads to excessive queueing on the server side, causing SLO violations; at the same time, too low an R leads to low overall throughput and low resource utilization.

[1] For the sake of presentation, we omit the technical assumptions. For a complete description on the theorem, please refer to Appendix B.2

Input: Current throughput R, latency t, SLO target t_0
Output: Updated throughput R

/* Initialize global variables. */
1 $\mathbb{T} \leftarrow 0$; $\mathbb{R} \leftarrow 0$ ▷ Last involved latency and throughput.
/* Update R for each round. */
2 **Procedure** UpdateR ()
/* Calculate Δ_R according to Newton's method. */
3 $\Delta_R \leftarrow \frac{(R-\mathbb{R})(t_0-T)}{T-\mathbb{T}}$
/* Bounds checking, throughput should be positive. */
4 **if** $R+\Delta_R < 0$ **then** ▷ Unlikely. Violates (3).
5 $R \leftarrow \frac{R}{M}$ ▷ Discard Δ_R and divide R by half.
6 **else**
7 $R \leftarrow \Delta_R + R$

Pseudocode. 1: Dynamic search of optimal R.

Let $R(X)$ be a function implicitly determined by the boundary constraint Eq. (2), i.e.,

$$T(X, R(X)) = t_0.$$

The implicit function $R(X)$ is indeed well defined, since for any X, $T(X,R)$ is monotonically increasing,[2] implying there exists an unique request throughput $R(X)$ that satisfies the boundary constraint, i.e., the maximum throughput is achieved when the latency is equal to the SLO target.

Essentially, we have to design a dynamic algorithm that actuates $R(X)$ in real-time (via $R_k(X)$ in step 1). This problem can be solved with a root-finding algorithm such as the classic Newton's method. However, if we apply this method directly, we may encounter situations where the updated throughput R is negative, which violates constraint (3). This happens when the throughput is too high and needs to be significantly reduced. In this case, we divide R by M instead of updating it using Newton's method. This ensures that (i) the updated throughput is positive; and (ii) the updated throughput is still significantly lower than before. We note that this out-of-bound scenario does not happen frequently. For simplicity, we choose $M = 2$. Our algorithm of searching for the optimal R is shown in Pseudocode 1.

X loop: RPC fraction control. For any given RPC fraction, the rate control of Kayak essentially maximizes throughput within the allowance of SLO requirement. With rate control, we effectively get the throughput as a function of the given RPC fraction ($R(X)$). In this part, we focus on the complementary and optimize the RPC fraction to maximize $R(X)$. We use a gradient ascent algorithm to achieve that. When the updated RPC fraction falls out of the range of $[0,1]$, we apply

[2] We assume that $T(X,R)$ is continuous, and for any X, there exist R_1 and R_2 such that $T(X,R_1) \le t_0 \le T(X,R_2)$. This assumption pluses monotonicity yields the existence and uniqueness of the implicit function $R(X)$.

Input: Current throughput R, RPC propotion X,
Output: Updated RPC propotion X

/* Initialize global variables. */
1 $\mathbb{R} \leftarrow 0$; $\mathbb{X} \leftarrow 0$ ▷ Last involved throughput and RPC fraction.
/* Update X for each round. */
2 **Procedure** UpdateX ()
/* Calculate Δ_X according to Gradient Ascent. */
3 $\Delta_X \leftarrow -\eta \frac{R-\mathbb{R}}{X-\mathbb{X}}$
/* Bounds checking, X should be within constraints. */
4 **if** $X+\Delta_X \notin [0,1]$ **then** ▷ Unlikely. Violates (4).
5 $X \leftarrow \max\{\min\{X+\Delta_X, 1\}, 0\}$
6 **else**
7 $X \leftarrow \Delta_X + X$

Pseudocode. 2: Dynamic search of optimal X.

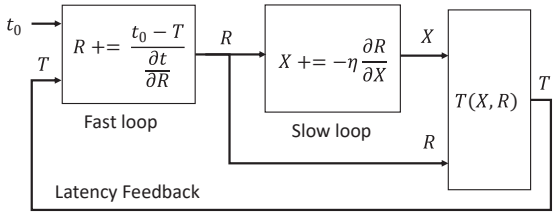


Figure 8: ***Nested control loops of Kayak.***

rounding to ensure it is within the boundary. Our algorithm of searching the RPC fraction is shown in Pseudocode 2.

Putting them together. Combing the rate control and the RPC fraction control, our algorithm (Algorithm II) naturally forms a bi-level (nested) control loops [15], with two actuators X and R and only one feedback signal t. We adopt a single control loop (the inner/fast loop), called R loop, to implement the rate control, i.e., finding the maximum throughput R while not violating the SLO t_0. The input of this control loop is the measured latency SLO metric $\widehat{T}$ and the output is request rate R which is the input for our request arbiter. We then adopt another control loop (the outer/slow loop), called X loop, to implement our request arbiter, i.e., choosing the best X that maximizes R_0.

Although this dual loop control design decouples the two actuators X and R, the resulting two feedback loops may be coupled. The coupling between two feedback loops may cause oscillation, which can be mitigated by choosing different sampling frequencies [15]. The exact two values can be tuned by the operator according to different workloads and system configurations. However, because the functioning of the second loop is dependent on the output of the first loop (R) to have converged to a stable point, it is best practice to choose a lower frequency for the second loop. Theoretically, we show that this dual loop control algorithm is guaranteed to converge in Section 4.4. Empirically, in our experiments, we let the

sampling rates of the first and second loops to be 200Hz and 20Hz respectively, and we show that the system converges fast to near optimal throughput in Section 6. We evaluate the impact of frequency selection in detail in Section 6.5.

4.4 Perfomance Guarantee

From the R loop, we obtain an estimation $R_k(x)$ at each iteration k, which approximately satisfies $T(X, R_k(X)) \approx t_0$. In the X loop, we optimize X for our request arbiter such that R_0 is maximized. This is done by *stochastic gradient ascent* (SGA, or *online gradient ascent*) on X. There is a rich literature in online learning theory for SGA when $R_k(x)$ is concave, e.g., see [36]. Applying related theoretical results to our problem, we have the following performance guarantee for our system. The proof of Theorem 2 is deferred to Appendix B.3.

Theorem 2. *Suppose for all $k = 1, \ldots, K$, $R_k(X)$ is concave, and $\|\nabla R_k(X)\|_2 \le L$. Consider the iterates of SGA, i.e.,*

$$X_{k+1} = X_k + \eta \nabla R_k(X_k).$$

Then we have the following regret bound

$$\sum_{k=1}^{K} (R_k(X_*) - R_k(X_k)) \le C \cdot \sqrt{K}, \tag{5}$$

where $C := L\|X_1 - X_*\|_2$ is a constant depends on initialization and gradient bound, and X_* can be any fixed number. Note that the regret bound holds even $R_k(X)$ is chosen adversarially based on the algorithm history.

Inteperation of Theorem 2. The sublinear regret bound implies SAG behaviors nearly optimal on average: we see this by setting $X_* = \arg\max_X \sum_{k=1}^{K} R_k(X)$, and noticing that

$$\frac{1}{K}\sum_{k=1}^{K} R_k(X_*) - \frac{1}{K}\sum_{k=1}^{K} R_k(X_k) \le O\left(\frac{1}{\sqrt{K}}\right) \to 0.$$

More concisely, in our algorithm, $\{R_k(X)\}_{k=1}^{K}$ corresponds to a sequence of inaccurate estimations to the true implicit function $R(X)$ — even so the theorem guarantees a sublinear regret bound, which implies that our algorithm behaviors nearly as good as one can ever expect under the estimations, no matter how inaccurate they could be.

Furthermore, if for each k, $R_k(X)$ is an *unbiased estimator* to the true concave function $R(X)$, i.e., $\mathbb{E}R_k(X) = R(X)$, then $\bar{X} = \frac{1}{K}\sum_{k=1}^{K} X_k$ converges to the maximal of $R(X)$ in expectation: we see this by choosing $X_* = \arg\min_X R(X)$ and noticing that

$$\begin{aligned}
\mathbb{E}\left[R(X_*) - R(\bar{X})\right] &\le \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}\left[R(X_*) - R(X_k)\right] \\
&= \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}\left[R_k(X_*) - R_k(X_k)\right] \\
&\le C \cdot \frac{1}{\sqrt{K}} \to 0.
\end{aligned}$$

The above convergence result does not require any assumptions on the randomness of R_k, as long as $R_k(X)$ is an unbiased estimator of $R(X)$. This means our algorithm can *tolerate* variance in the measured latency which causes variance in estimated R_k. The convergence is empirically validated by our experiments in Section 6.1.

4.5 Scalability and Fault Tolerance

Scalability. Kayak is fully decentralized, and its control logic (e.g., rate and RPC fraction determination) is decoupled from the request execution in the dataplane. Throughput of a tenant is limited by its total available resources in application and storage servers; one can increase throughput by adding more application servers or by ensuring more resource share in the storage servers.

Fault tolerance. Kayak does not introduce additional systems components beyond what traditional KV- or RPC-based or hybrid systems do. As such, it does not introduce novel fault tolerance challenges. The consistency and fault tolerance of the KV store is orthogonal to our problem and out of the scope of this paper.

5 Implementation

We build a prototype of Kayak with about 1500 lines of code and integrate it with the in-memory kernel-bypassing key-value store Splinter [29]. The code is available at: https://github.com/SymbioticLab/Kayak

Kayak interface. Users of Kayak provide their custom defined storage functions (App Logic in Figure 6), which are compiled with Kayak and deployed onto both the application server and storage server. At runtime, users connect to Kayak and set the desired SLO target. Users then submit request in the format of storage function invocations to Kayak.

Application server. The core control logic of Kayak is implemented in the application server. One challenge we face during implementation is to optimize the code to reduce overhead, which is especially important because of the high throughput low latency requirement. For instance, the inner control loop constantly measures request latency and calculate the 99%-tile. One naive way is to measure the quantile is using selection algorithm to calculate the k-th order statistics of n samples, with has at least $O(n)$ complexity. Instead, we apply DDSketch [32] to estimate the quantile in real time with bounded error.

Storage server. The main challenge of implementing the storage server is supporting multi-tenancy and ensuring fairness and work conservation. We pin requests from different tenants to different CPU cores to ensure fairness. And we adopt work stealing to ensure work conservation: CPU cores with no requests to process steal requests from the queues of other cores. Specifically, similar to ZygOS [38], each CPU core of Kayak steals from all other CPU cores, which is different from Splinter's work stealing from only neighboring

CPU	Intel E5-2640v4 2.4 GHz
RAM	64GB ECC Memory DDR4 2400MHz
NIC	Mellanox ConnectX-4 25 GB NIC
OS	Ubuntu 16.04, Linux 4.4.0-142

Table 2: ***Server configurations for our testbed in CloudLab.***

cores. This further improves overall CPU utilization.

6 Evaluation

In this section we empirically evaluate Kayak with a focus on: (i) verification of convergence; (ii) performance improvement against state of the art [12]; and (iii) fairness and scalability with multiple tenants. Our key results are as follows.

- Kayak achieves sub-second convergence to optimal throughput and RPC fraction regardless of workloads. It can proactively adjust to dynamic workload change as well (§6.1).
- Kayak improves overall throughput by 32.5%-63.4% for compute-intensive workloads and up to 12.2% for non-compute-intensive and transactional workloads (§6.2).
- In a multi-tenant setup, Kayak approximates max-min fair sharing, with a Jain's Fairness Index [22] of 0.9996 (§6.3) and scales without sacrificing fairness (§6.4).
- We also evaluate Kayak's sensitivity to its different parameters (§6.5).

Methodology. We run our experiments on CloudLab [3] HPE ProLiant XL170r machines (Table 2). Unless specified otherwise, we configure Kayak to use 8 CPU cores across all servers. The fast control loop algorithm is configured to run every 5ms and the slow control loop runs every 50ms. The initial RPC fraction is set at 100%, and we define SLO as the 99%-tile latency.

Workloads. We use the workload described in Section 2. Unless otherwise specified, we configure the workload with a traversal depth of two so that each request issues two data accesses to the storage. We vary the amount of computation that takes place after each access and refer to them as Light (100ns computation time per access), Medium (1μs per access) and Heavy (10μs per access). This workload emulates a variety of workloads with different computational load in a non-transactional environment.

We extend this workload and create a Bimodal workload. We denote by `Bimodal(1us, 100ns, 50%, 5s)`, a workload that consists of 50% Medium (1μs/RTT) and 50% Light (100ns/RTT) with an interval of 5 seconds.

We also run YCSB-T [14] as a transactional workload. This workload is not computationally intensive.

Unless otherwise specified, for all workloads, we set our latency SLO target as: 99%-tile request latency lower than or equal to 200μs.

Baseline. Our primary baseline is ASFP [12], which is available at https://github.com/utah-scs/splinter/releases/tag/ATC'20 and also built on top of Splinter [29].

6.1 Convergence

In this section, we validate that Kayak's fast loop can converge to a stable throughput R while satisfying SLO constraint and when running together with fast loop, the slow loop can also converge to the optimal RPC fraction.

Fast loop only. We first disable the slow loop and run Kayak with a fixed RPC fraction (100%), to show that the fast loop (rate control) can converge to optimal throughput with different workloads.

We run Light, Medium and Heavy workloads with one application server and one storage server, and measure how the throughput and 99%-tile latency changes with time. As shown in Figure 9, Kayak ramps up the throughput quickly when the measured 99%-tile request latency is below the SLO threshold of 200μs. Along with the increase of throughput, the latency also increases, as observed from the rise of red line. The entire converging process happens within 0.2 seconds.

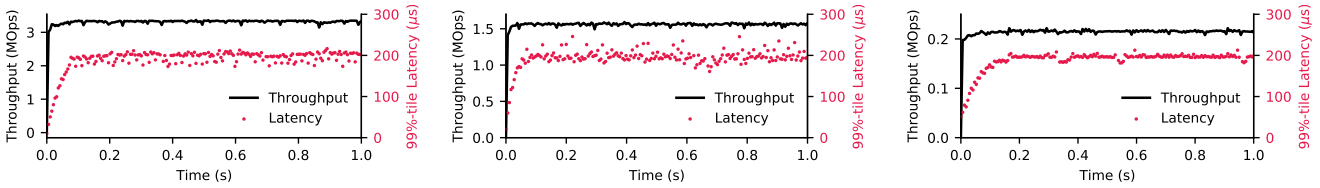
After approaching the SLO limit, both the throughput and latency remains stable with minor fluctuations, confirming the convergence of our fast loop. We note that the converged throughput are the same as the measurements of the RPC-only configuration in Figure 4. This means that our fast loop indeed converges to the optimal throughput.

Dual loop control. Now we move on to verifying the convergence of both loops combined. We repeat the previous experiments, but with both control loops enabled. Figure 10 shows the dynamics of throughput and RPC fraction and how they change with time. We highlight three observations.

- Similar to Figure 9, throughput increases rapidly within the first 0.2 seconds; this is due to the fast loop.
- With the Medium and Heavy workloads, the throughput increase slows down after 0.2 seconds. This increase comes from the slow loop, as we can see a change in RPC fraction. Note that the Light workload does not show this trend, because in this setup the initial RPC fraction (100%) is already the optimal for it.
- After 1 second since the start, the throughput converges to a stable value with only minor fluctuations.

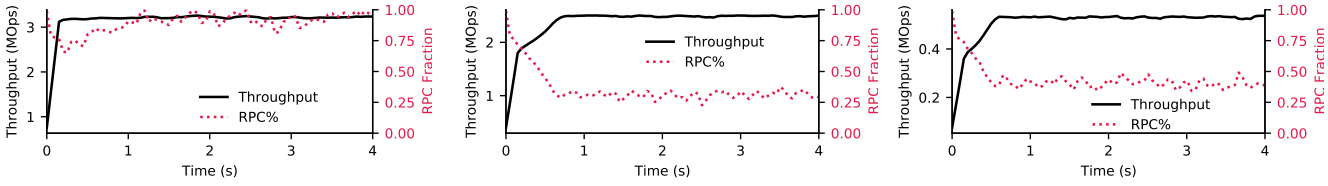
Comparing the RPC fraction in Figure 10 against Figure 5, we observe that our algorithm converges to the optimal RPC fraction. Comparing the throughput in Figure 10 against the Optimal configuration in Figure 4, we observe that the converged throughput is the optimal throughput.

Convergence under dynamic workloads. One advantage of Kayak is that it can proactively adjust to changing workload. To verify this, we run Kayak with the `Bimodal(1us, 100ns, 50%, 5s)` workload. Figure 11 shows the dynamics of throughput and RPC fraction. As we can see, Kayak adapts to the changing workload, and adjusts both the throughput and RPC fraction accordingly in a timely fashion.



(a) *Light (100ns/RTT)* (b) *Medium (1µs/RTT)* (c) *Heavy (10µs/RTT)*

Figure 9: ***Throughput and 99%-tile latency w.r.t time, with only the fast loop of Kayak and fixed RPC fraction of 100%.***



(a) *Light (100ns/RTT)* (b) *Medium (1µs/RTT)* (c) *Heavy (10µs/RTT)*

Figure 10: ***Dynamics of throughput and RPC fraction w.r.t time, with nested control loops of Kayak.***

SLO	YCSB-T		Light		Medium		Heavy		Bimodal	
	Kayak	ASFP	Kayak	ASFP	Kayak	ASFP	Kayak	ASFP	Kayak	ASFP
50μs	2.63	2.58	3.05	3.05	1.71	1.06	N/A	N/A	2.13	1.43
100μs	3.12	2.78	3.36	3.36	2.37	1.45	0.37	N/A	2.74	2.16
200μs	3.35	3.01	3.59	3.52	2.54	1.64	0.48	0.33	2.98	2.37
400μs	3.35	3.02	3.70	3.61	2.61	1.68	0.57	0.40	3.03	2.48

Table 3: ***Throughput (MOps) of Kayak and ASFP under different workloads and SLO targets. "N/A" means the SLO target is infeasible.***

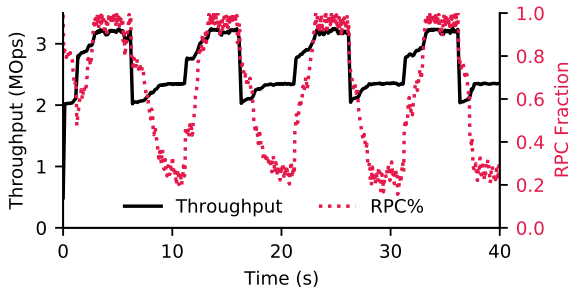


Figure 11: ***Dynamics of throughput and RPC fraction w.r.t time under Bimodal workload.***

6.2 Performance

Performance improvement. We compare the performance of Kayak against the state of the art ASFP [12]. We use all the workloads: (i) Light/Medium/Heavy computational workload; (ii) Bimodal workload; and (iii) YCSB-T workload. We use one application server and one storage server, and vary the SLO target from 50μs to 400μs. The results are shown in Table 3, which we summarize as follows.

- For non-compute-intensive workloads (Light and YCSB-T), Kayak achieves up to 12.2% throughput improvement. In this case, most of the requests are handled via RPC and Kayak's opportunity for improvement is lower.
- For compute-intensive workloads, Kayak achieves 32.5%-63.4% throughput improvement. In this case, the RPC fraction decreases, and ASFP's pushback mechanism kicks in; the overhead of pushing requests back in comparison to Kayak's proactive placement increases the gap.

Overall, Kayak outperforms ASFP because its proactive design is more efficient in using both application- and storage-side CPUs.

Workload	YCSB-T	Light	Medium	Heavy	Bimodal
Gap	5.4%	11.8%	6.7%	3.5%	10.6%

Table 4: ***Kayak's performance gap from the upper bound for different workloads.***

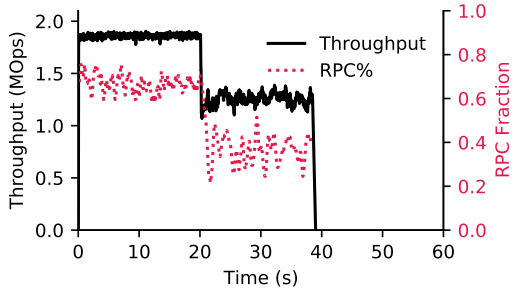
Gap between Kayak and upper bound. We set the SLO target to be 200μs and compare the achieved throughput between using Kayak and the upper bound obtained by the parameter sweep method (§2.2). We define the gap between Kayak and the upper bound as follows:

$$Gap = \frac{Throughput_{max}}{Throughput_{max,Kayak}} - 1$$

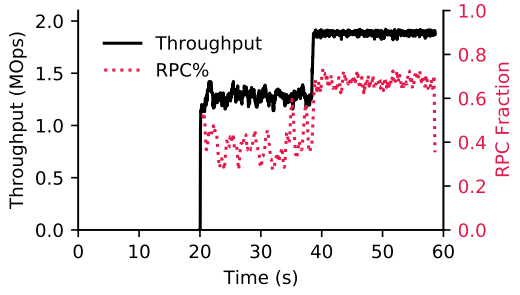
Intuitively, the lower the gap is, the closer Kayak is to the optimal. As shown in Table 4, Kayak has a slowdown of 11.8% for computationally light workload and 3.5% for computationally heavy workload.

6.3 Fairness

In this section, we show that Kayak can enforce max-min fairness when multiple tenants contend for server resources.



(a) *Tenant A*



(b) *Tenant B*

Figure 12: ***Dynamics of throughput and RPC fraction w.r.t time of tenant A and B in the multi-tenant experiment.***

We run two tenants with the same setup and configure each to use four CPU cores instead of eight, while the server is still configured with eight CPU cores. Tenant A is started first, and after 20 seconds, tenant B is started. Figure 12a shows the dynamics of tenant A. During the first 20 seconds, tenant A quickly converges to the optimal throughput of around 1.82MOps. After 20 seconds when tenant B is started, the throughput achieved by tenant A drops to around 1.21MOps. Note that in this process, the optimal RPC fraction also shifts from 60% to 40%. This is because after tenant B joins, the server has less CPU resource to process tenant A's requests. After 40 seconds, tenant A stops sending requests.

Figure 12b shows the dynamics of tenant B, and we can see that when the two tenants are running together the achieved throughput is 1.21MOps and 1.24MOps, respectively. The gap between the two tenants is only 2.4%, which indicates that the server resources are fairly shared. After 40 seconds, the throughput and RPC fraction of tenant B increases because the storage server has more available CPU resources after tenant A stops.

Then we increase to four tenants and start the tenants one after one, with ten seconds in between. Each tenant is configured with one CPU core, and runs a different workload. Tenant {1, 2, 3, 4} runs {Light,Medium,Heavy,Bimodal}, respectively. We plot the occupied CPU cycles on the storage server for each tenant in Figure 13. When the 4 tenants are running together, we measure the Jain's Fairness Index [22] to be 0.9996.

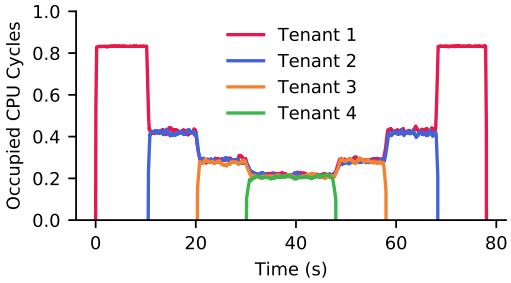
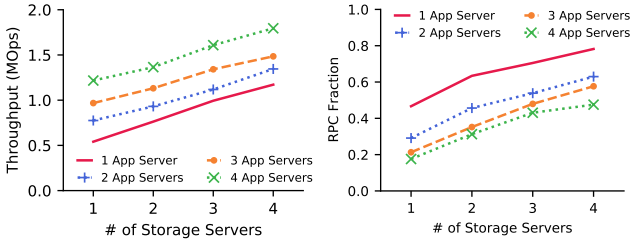


Figure 13: ***Fair-sharing of throughput for 4 applications sharing one storage server.***



(a) *Throughput* (b) *RPC Fraction*

Figure 14: ***Throughput and RPC Fraction of Kayak with different server configurations.***

6.4 Scalability

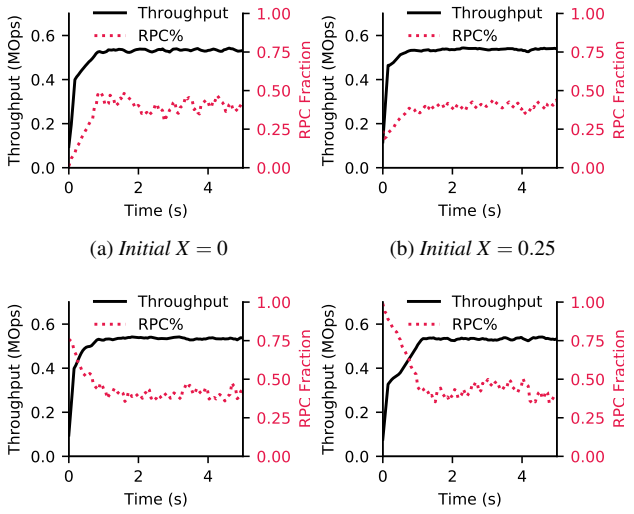
In this section, we verify that the Kayak control loops are decoupled from data plane and do not limit Kayak's scalability. To do so, we run experiments with Heavy workload, and vary the number of application and storage servers. For simplicity, we make sure a single request does not access data from more than one storage servers. We measure the total throughput and the converged average RPC fraction. As shown in Figure 14a, throughput increases with both adding a storage and an application server. This is because adding either essentially adds more CPU cores to the system. Note that the RPC fraction increases with the increment of the storage servers but decreases with the increment of the application servers (Figure 14b). This is because Kayak can judiciously arbitrate the requests and balance the load between the application and storage servers, by choosing the optimal RPC fraction.

6.5 Sensitivity Analysis

Initial state. First we evaluate whether Kayak is sensitive to its initial state. We run four experiments using the Heavy workload, and vary the starting value for RPC fraction X from {0, 0.25, 0.75, 0.100}. As shown in Figure 15, Kayak converges to the optimal request throughput and RPC fraction regardless of the initial RPC fraction in all four scenarios.

Choice of loop frequencies. In Section 4 we argue that in order for the dual loop control to work, we need to choose appropriate sampling frequencies for both loops. Here we analyze how different sampling frequencies affect the dynamics of our system.

We run two experiments using the Medium workload, and



(a) *Initial* $X = 0$ (b) *Initial* $X = 0.25$

(c) *Initial* $X = 0.75$ (d) *Initial* $X = 1$

Figure 15: ***Throughput and RPC fraction during the converging process, with different starting RPC fraction.***

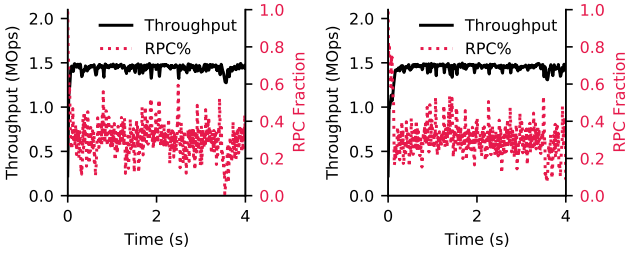
plot our results in Figure 16. In the first experiment we set the interval for both loop to be 200Hz; in the second experiment we invert the loop frequency so that the inner control loop runs slower than outer control loop. As shown in Figure 16a and 16b, the convergence quality degrades significantly in both cases. Hence, it is important to choose proper sampling frequencies to ensure that the inner control loop runs faster than the outer.

7 Discussion and Future Work

Rate Limiting. In Kayak, we consider rate limiting and admission control because simply moving up the latency curve cannot push the servers beyond their physical capacity, and we consider that if a tenant specifies a strict SLO target, requests that miss this target are essentially failed requests. As such, we assume that the tenants will implement a mechanism such as resubmitting requests which are dropped due to rate limiting, after adapting to a slower rate. At a slower timescale, tenants should also increase provisioning to avoid having to slow down, so that in the worst case scenario all requests can still be executed on all application servers within the SLO target.

Storage Function Differentiation. Kayak does not inspect individual storage functions, which reduces overhead and aligns with our design goal to be non-intrusive to applications. Therefore, Kayak can not distinguish between individual storage functions. We would like to explore how to differentiate individual storage functions without changing application code, which would allow us to implement more fine-grained control policy.

Developing Storage Functions. Currently, developers using Kayak and Splinter [29] have to code the same application logic again in the storage function which run inside the storage server. This duplication complicates the development process and increases the chance of human errors and bugs. Automatically generating server-side storage functions from code written using traditional KV API can alleviate this problem. This would be an interesting avenue for future work.



(a) *Equal loop frequency* (b) *Inverse loop frequency*

Figure 16: ***Dynamics of throughput and RPC fraction w.r.t time, with different loop frequencies: (a) both loops run at 200Hz; (b) inner loop at 20Hz, outer loop at 200Hz.***

8 Related Work

Key-Value Stores. A recent line of research on key-value store has been focusing on utilizing RDMA to boost key-value store performance. Stuedi et al. [43] have achieved a 20% reduction in GET CPU load for Memcached using soft-iWARP without Infiniband hardware. Pilaf [33] uses only one-sided RDMA read to reduce CPU overhead of key-value store. In addition, it uses a verifiable data structure to detect read-write races. FaRM [16] proposes a new main memory key-value store built on top of RDMA. FaRM comes with transaction support but still support lock-free RDMA reads. HERD [24] further improves the performance of RDMA-based key-value store by focusing on reducing network round trips while using efficient RDMA primitives. FaSST [25] generalizes HERD and used two-sided RPC to reduce the number of QPs used in symmetric settings such as distributed transaction processing, improving scalability.

While there have been many research works focusing on improving raw performance of key-value stores, few investigates real performance implications on the application. TAO [13] is an application-aware key-value store by Facebook that optimizes for social graph processing. Kayak builds on top of this concept and focuses on application-level objectives such as SLO constraint.

Storage-side computation. Storage-side computation (i.e. *shipping compute to data*) has made its way from latency-insensitive big data systems such as MapReduce [1, 37, 45] and SQL databases [5–8, 26, 41, 42] into latency-critical KV stores [9, 18, 29, 39]. Comet [18] supports sandboxed Lua extensions to allow user-defined extensions to customize the storage by enabling application-specific operations. Malacology [39] utilizes Lua extensions contributed by users of the Ceph storage system [2], allowing installing and updating new object interfaces at runtime. Splinter [29] pushes bare-metal

extension to storage server to allow RPC-like operations in addition to traditional key-value operations. These works breaks the assumption of dissagregated storage and necessitates the need for proactive arbitration provided by Kayak.

Adaptive compute placement. An emerging line of research aims at adaptively balancing between client-side processing and server-side processing. ASFP [12] extends Splinter by reactively pushing back requests to the client side if the server gets overloaded, but at the cost of wasting CPU and network resource.Instead, Kayak proactively balances the load exerted on both application and storage server. Cell [34] implements a B-tree store on RDMA supporting both client-side (RDMA-based) and server-side (RPC-based) search. Cell determines between these two schemes by tracking RDMA operation latency. This requires instrumentation into the application, which Kayak avoids by measuring end-to-end request latency instead. A recent work called Storm [35] uses a reactive-adaptive approach similar to that of ASFP [12] but with a different policy, where for each request it will try the traditional KV API first, and switch to RPC API if it detects that the application is trying to chase the pointers.

9 Conclusion

In this paper, we show that by proactively and adaptively combining RPC *and* KV together, overall throughput and CPU utilization can be improved. We propose an algorithm that dynamically adjusts the rate of requests and the RPC fraction to improve overall request throughput while meeting latency SLO requirements. We then prove that our algorithm can converge to the optimal parameters. We design and implement a system called Kayak. Our system implementation ensures work conservation and fairness across multiple tenants. Our evaluations show that Kayak achieves sub-second convergence and improves overall throughput by 32.5%-63.4% for compute-intensive workloads and up to 12.2% for non-compute-intensive and transactional workloads.

Acknowledgements

Special thanks go to the entire CloudLab team for making Kayak experiments possible. We would also like to thank the anonymous reviewers, our shepherd, Ryan Stutsman, and SymbioticLab members for their insightful feedback. This work is in part supported by NSF grants CNS-1813487, CNS-1845853, CNS-1900665, CNS-1909067, and CCF-1918757.

References

[1] Apache Hadoop. https://hadoop.apache.org/.

[2] Ceph. https://ceph.io/.

[3] CloudLab. https://cloudlab.us/.

[4] Intel Rack Scale Design. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[5] Microsoft SQL Server Stored Procedures. https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine.

[6] Microsoft SQL Server User-Defined Functions. https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions.

[7] MySQL Stored Procedures Tutorial. https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx/.

[8] Oracle PL/SQL. https://www.oracle.com/database/technologies/application-development-PL/SQL.html.

[9] Redis. https://redis.io/.

[10] Atul Adya, Daniel Myers, Henry Qin, and Robert Grandl. Fast key-value stores: An idea whose time has come and gone. In *HotOS*, 2019.

[11] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[12] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. Adaptive placement for in-memory storage functions. In *ATC*, 2020.

[13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook's distributed data store for the social graph. In *ATC*, 2013.

[14] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. Ycsb+ t: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230. IEEE, 2014.

[15] John C Doyle, Bruce A Francis, and Allen R Tannenbaum. *Feedback Control Theory*. Courier Corporation, 2013.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *NSDI*, 2014.

[17] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[18] Roxana Geambasu, Amit A Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M Levy. Comet: An active distributed key-value store. In *OSDI*, 2010.

[19] Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[20] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM*, 2016.

[21] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. Tcp≈rdma: Cpu-efficient remote storage access with i10. In *NSDI*, 2020.

[22] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.

[23] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *NSDI*, 2019.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *SIGCOMM*, 2014.

[25] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided rdma datagram rpcs. In *OSDI*, 2016.

[26] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[27] Leonard Kleinrock. Queueing systems. volume i: theory. 1975.

[28] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *EuroSys*, 2016.

[29] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *NSDI*, 2018.

[30] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[31] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *NSDI*, 2014.

[32] Charles Masson, Jee E Rim, and Homin K Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment*, 12(12).

[33] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *ATC*, 2013.

[34] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing cpu and network in the cell distributed b-tree store. In *ATC*, 2016.

[35] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, et al. Storm: a fast transactional dataplane for remote data structures. In *SYSTOR*, 2019.

[36] Francesco Orabona. A modern introduction to online learning. *arXiv preprint arXiv:1912.13213*, 2019.

[37] Dongchul Park, Jianguo Wang, and Yang-Suk Kee. In-storage computing for hadoop mapreduce framework: Challenges and possibilities. *IEEE Transactions on Computers*, 2016.

[38] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[39] Michael A Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A programmable storage system. In *EuroSys*, 2017.

[40] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *NSDI*, 2019.

[41] Michael Stonebraker and Greg Kemnitz. The postgres next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

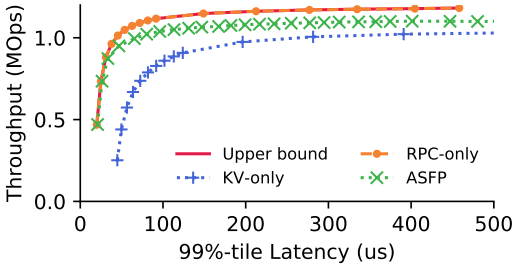
[42] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[43] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *ATC*, 2012.

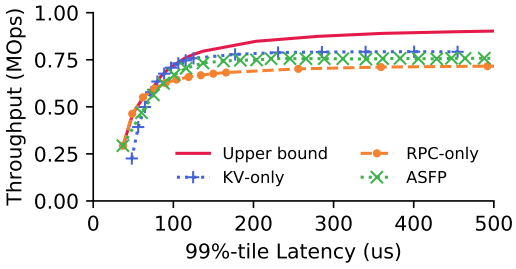
[44] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, 2020.

[45] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

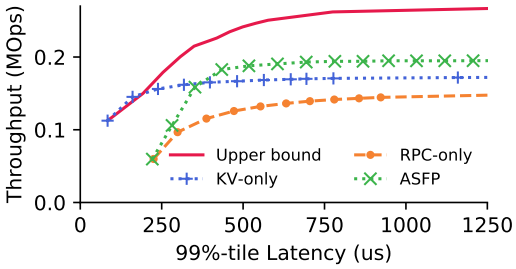
A Supplemental Measurements for Graph Traversal Workload



(a) *Light (100ns/RTT)*

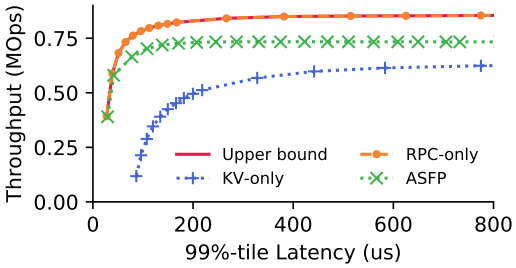


(b) *Medium (1µs/RTT)*

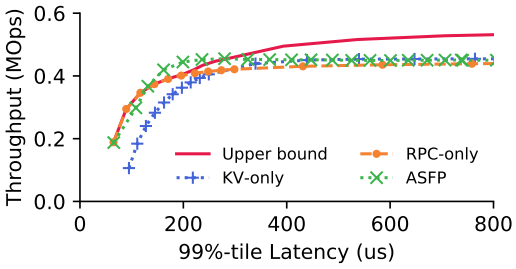


(c) *Heavy (10µs/RTT)*

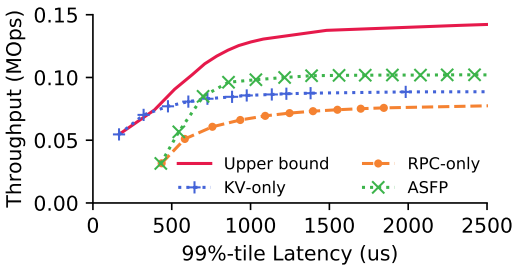
Figure 17: ***Throughput w.r.t. SLO (99%-tile latency) for graph traversal with four storage accesses per request.***



(a) *Light (100ns/RTT)*



(b) *Medium (1µs/RTT)*



(c) *Heavy (10µs/RTT)*

Figure 19: ***Throughput w.r.t. SLO (99%-tile latency) for graph traversal with eight storage accesses per request.***

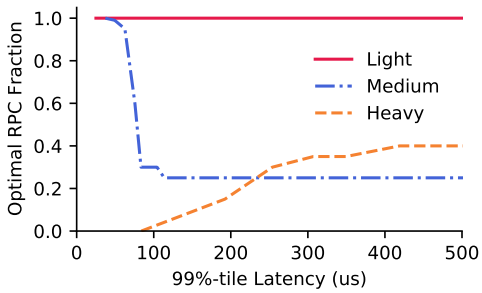


Figure 18: ***Optimal RPC fraction w.r.t. SLO (99%-tile latency) for graph traversal with four storage accesses per request.***

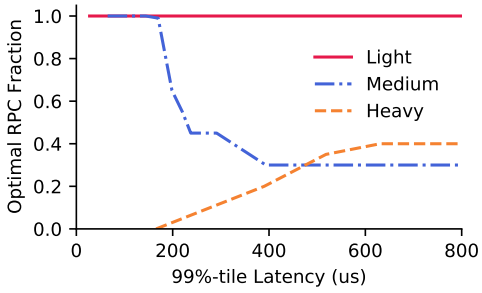


Figure 20: ***Optimal RPC fraction w.r.t. SLO (99%-tile latency) for for graph traversal with eight storage accesses per request.***

We repeat the same parameter sweep as in §2.2 but with the number of storage accesses per request set to 4 and 8. We vary the RPC fraction from 0 to 1 and measure the overall throughput and end-to-end latency of all requests. In doing so, we obtain the throughput-latency measurements for all possible execution configurations, as shown in Figure 17 and Figure 19. The optimal RPC fraction during these sweeps are shown in Figure 18 and Figure 20. We note that the observation we make in §2.2 still holds in these measurements.

B Analysis of Algorithms

B.1 Problem Formulation

We aim to solve the following optimization problem

$$\begin{aligned} \max_{X} \quad & R \\ \text{s.t.} \quad & T(X,R) \le t_0 \end{aligned} \tag{P}$$

where we assume:

Assumption 1. *Fix X, $F_X(R) := T(X,R)$ is monotonic increasing and twice differentiable.*

Assumption 2. *For any X, there exist R_1 and R_2 such that $\min_X T(X,R_1) \le t_0 \le \min_X T(X,R_2)$.*

B.2 Algorithm I: X-R Dual Loop Control

We adopt the following iterative algorithm to solve the problem:

Algorithm I (X-R Dual Loop Control) For $k = 1,\dots,K$, we alternatively update X_k and R_k by

- $X_k = \arg\min_X T(X,R_k)$;
- $R_{k+1} = R_k + \eta\,(t_0 - T(X_k,R_k))$.

We have the following theorem to characterize the convergence of the above algorithm.

Theorem 1. *Suppose in addition we have,*

1. *$T(X,R)$ is twice differentiable and lower bounded.*
2. *Fix R, $F_R(X) := T(X,R)$ is μ-strongly convex[3], L-smooth[4] and coercive.[5]*
3. *For all X, we have $0 < \alpha \le \frac{\partial T(X,R)}{\partial R} \le \beta$.*

If we set $0 < \eta < \frac{1}{\beta}$, then

$$|R_K - R_*| \le (1-\eta\alpha)^K \cdot |R_0 - R_*|\,.$$

In the following we elaborate the proof for Theorem 1. We begin with introducing a series of lemmas. Let us denote

$$H(R) = \min_X T(X,R).$$

Lemma 1. *For all $R \ne S$,*

$$0 < \alpha \le \frac{H(R)-H(S)}{R-S} \le \beta.$$

Moreover, the above inequality implies $H(R)$ is monotonic increasing and continuous.

Proof. Without loss of generality let $R > S$. Let $X = \arg\min_X T(X,R)$ and $Y = \arg\min_X T(X,S)$. Then

$$\begin{aligned} \frac{H(R)-H(S)}{R-S} &= \frac{T(X,R)-T(Y,S)}{R-S} \\ &\begin{cases} \le \frac{T(Y,R)-T(Y,S)}{R-S} = \frac{\partial T(Y,P)}{\partial R} \le \beta, \\ \ge \frac{T(X,R)-T(X,S)}{R-S} = \frac{\partial T(X,Q)}{\partial R} \ge \alpha > 0. \end{cases} \end{aligned}$$

Here $P, Q \in (S,R)$ are given by mean-value theorem. □

Lemma 2. *There exists an unique R_* such that $H(R_*) = t_0$. Moreover, this R_* gives the maximum of the original optimization problem.*

Proof. We have already shown that $H(R)$ is monotonic and continuous. Recall that there exists R_1 and R_2 such that $H(R_1) \le t_0 \le H(R_2)$, thus there exists an unique R_* such that $H(R_*) = t_0$.

For any R so that $R > R_*$, we have $H(R) > H(R_*) = t_0$ by monotonicity, thus R does not meet the constraint. Therefore R_* is the maximum of the optimization problem. □

Proof of Theorem 1. With $H(R) := \min_X T(X,R)$, we can rephrase Algorithm I as

$$R_{k+1} = R_k + \eta\,(H(R_*) - H(R_k))\,.$$

Let R_0 be the initialization. We next show the convergence of this iteration.

Case I: $R_0 < R_*$. If $R_k < R_*$, then

$$\begin{aligned} R_* - R_{k+1} &= R_* - R_k - \eta\,(H(R_*) - H(R_k)) \\ &\begin{cases} \le R_* - R_k - \eta\alpha\,(R_* - R_k) = (1-\eta\alpha)\,(R_* - R_k) \\ \ge R_* - R_k - \eta\beta\,(R_* - R_k) = (1-\eta\beta)\,(R_* - R_k) \end{cases} \end{aligned}$$

that is $0 \le (1-\eta\beta)\,(R_* - R_k) \le R_* - R_{k+1} \le (1-\eta\alpha)\,(R_* - R_k)$. Using this recursion, if $R_0 < R_*$, we have

$$0 \le (1-\eta\beta)^K\,(R_* - R_0) \le R_* - R_K \le (1-\eta\alpha)^K\,(R_* - R_0).$$

Case II: $R_0 > R_*$. If $R_k > R_*$, then

$$\begin{aligned} & R_{k+1} - R_* \\ &= R_k + \eta\,(H(R_*) - H(R_k)) - R_* \\ &= R_k - R_* - \eta\,(H(R_k) - H(R_*)) \\ &\begin{cases} \le R_k - R_* - \eta\alpha\,(R_k - R_*) = (1-\eta\alpha)\,(R_k - R_*) \\ \ge R_k - R_* - \eta\beta\,(R_k - R_*) = (1-\eta\beta)\,(R_k - R_*) \end{cases} \end{aligned}$$

that is $0 \le (1-\eta\beta)\,(R_k - R_*) \le R_{k+1} - R_* \le (1-\eta\alpha)\,(R_k - R_*)$. Using this recursion, if $R_0 > R_*$, we have

$$0 \le (1-\eta\beta)^K\,(R_0 - R_*) \le R_K - R_* \le (1-\eta\alpha)^K\,(R_0 - R_*).$$

To sum up, when $\eta < \frac{1}{\beta}$ (hence smaller than $\frac{1}{\alpha}$), we have

$$|R_K - R_*| \le O\left((1-\eta\alpha)^K\right).$$

□

[3] $f(x)$ is μ-strongly convex, if for all x and y, it holds that $f(x) \ge f(y) + \langle \nabla f(y), x-y\rangle + \frac{\mu}{2}\|y-x\|_2^2$.

[4] $f(x)$ is L-smooth, if for all x and y, it holds that $f(x) \le f(y) + \langle \nabla f(y), x-y\rangle + \frac{L}{2}\|y-x\|_2^2$. In general, if $f(x)$ is twice-differentiable and x is restricted in a bounded domain, then $f(x)$ is L-smooth in that domain, for some finite L.

[5] $f(x)$ is coercive if $f(x) \to +\infty$ as $\|x\| \to +\infty$. A strongly convex and coercive function admits an unique and finite minimum point.

B.3 Algorithm II: R-X Dual Loop Control

Let us take a closer look at the optimization problem (P) under Assumption 1 and Assumption 2. First we observe the maximal must be attended at the boundary

$$T(X,R) = t_0. \tag{6}$$

Second the boundary constraint Eq. (6) implicitly defines a function $R(X)$, where

$$T(X,R(X)) = t_0.$$

We highlight that $R(X)$ is indeed well defined, since under Assumption 1 and Assumption 2, for any X, there exists an unique $R(X)$ that satisfies the boundary constraint.

With the above observations, we may rephrase the optimization problem (P) as

$$\max_X \quad R(X) \tag{P'}$$

where $R(X)$ is implicitly defined by the boundary constraint. In the following we discussion algorithms that solve problem (P').

Our challenge it that we do not have direct access to $R(X)$; instead at each fast loop step, we have an estimation to $R(X)$, denoted as $R_k(x)$, which approximately satisfies

$$T(X,R_k(X)) \approx t_0.$$

In this set up we can perform *stochastic gradient ascent* (SGA, or *online gradient ascent*) for $R_k(X)$. We summarize the algorithm in the following.

Algorithm II (R-X Dual Loop Control) For $k = 1,\dots,K$, we respectively update X_k and R_k by

1. Apply rate control so that the latency approximates SLO, i.e.,
   R_k *be such that* $T(X_k,R_k) \approx t_0$*;*
2. Use gradient ascent to search for the optimal X_k, i.e.,
   $X_{k+1} = X_k + \eta \frac{\mathrm{d}R_k}{\mathrm{d}X}$*, where* $T(X,R_k) = t_0$*, and* η *is a positive stepsize.*

There is a rich literature for the theory of online learning when $R_k(X)$ is concave, e.g., see [36]. For completeness, we introduce the following theorem to characterize the behavior of the above algorithm.

Theorem 2. *Suppose $R_k(X)$ is concave. Consider the iterates of SGA, i.e.,*

$$X_{k+1} = X_k + \eta \nabla R_k(X_k).$$

Then we have the following bound for the regret

$$\sum_{k=1}^{K} (R_k(X_*) - R_k(X_k)) \le \frac{\|X_1 - X_*\|_2^2}{2\eta} + \frac{\eta}{2} \sum_{k=1}^{K} \|\nabla R_k(X_k)\|_2^2.$$

If in addition we assume $\|\nabla R_k(X)\|_2 \le L$, and set

$$\eta = \frac{\|X_1 - X_*\|_2}{L\sqrt{K}},$$

then

$$\sum_{k=1}^{K} (R_k(X_*) - R_k(X_k)) \le C \cdot \sqrt{K}, \tag{7}$$

where $C := L\|X_1 - X_*\|_2$ is a constant depends on initialization and gradient bound.

Remark. The sublinear regret bound implies SAG behaviors nearly optimal on average: we see this by setting $X_* = \arg\max_X \sum_{k=1}^{K} R_k(X)$, and noticing that

$$\frac{1}{K}\sum_{k=1}^{K} R_k(X_*) - \frac{1}{K}\sum_{k=1}^{K} R_k(X_k) \le O\left(\frac{1}{\sqrt{K}}\right) \to 0.$$

More concisely, in our algorithm, $\{R_k(X)\}_{k=1}^{K}$ corresponds to a sequence of inaccurate estimations to the true implicit function $R(X)$ — even so the theorem guarantees a sublinear regret bound, which implies that our algorithm behaviors nearly as good as one can ever expect under the estimations, no matter how inaccurate they could be.

Furthermore, if for each k, $R_k(X)$ is an *unbiased estimator* to the true concave function $R(X)$, i.e., $\mathbb{E}R_k(X) = R(X)$, then $\bar{X} = \frac{1}{K}\sum_{k=1}^{K} X_k$ converges to the maximal of $R(X)$ in expectation: we see this by choosing $X_* = \arg\min_X R(X)$ and noticing that

$$\begin{aligned}
\mathbb{E}\left[R(X_*) - R(\bar{X})\right] &\le \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}\left[R(X_*) - R(X_k)\right] \\
&= \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}\left[R_k(X_*) - R_k(X_k)\right] \\
&\le C \cdot \frac{1}{\sqrt{K}} \to 0.
\end{aligned}$$

Proof of Theorem 2. We first notice the following ascent lemma

$$\begin{aligned}
&\|X_{k+1} - X_*\|_2^2 \\
&= \|X_k + \eta\nabla R_k(X_k) - X_*\|_2^2 \\
&= \|X_k - X_*\|_2^2 + \eta^2\|\nabla R_k(X_k)\|_2^2 + 2\eta\langle \nabla R_k(X_k), X_k - X_*\rangle \\
&\le \|X_k - X_*\|_2^2 + \eta^2\|\nabla R_k(X_k)\|_2^2 + 2\eta\left(R_k(X_k) - R_k(X_*)\right),
\end{aligned}$$

where the last inequality is due to the assumption that $R_k(X)$ is concave. Next we re-arrange the terms and take telescope

summation,

$$
\begin{aligned}
&\sum_{k=1}^{K}\left(R_k(X_*)-R_k(X_k)\right)\\
&\leq \sum_{k=1}^{K}\frac{1}{2\eta}\left(\|X_k-X_*\|_2^2-\|X_{k+1}-X_*\|_2^2\right)+\sum_{k=1}^{K}\frac{\eta}{2}\|\nabla R_k(X_k)\|_2^2\\
&=\frac{1}{2\eta}\left(\|X_1-X_*\|_2^2-\|X_{K+1}-X_*\|_2^2\right)+\sum_{t=1}^{T}\frac{\eta}{2}\|\nabla R_k(X_k)\|_2^2\\
&\leq \frac{1}{2\eta}\|X_1-X_*\|_2^2+\sum_{k=1}^{K}\frac{\eta}{2}\|\nabla R_k(X_k)\|_2^2,
\end{aligned}
$$

which gives the first regret bound.

If further we have $\|\nabla R_k(X)\|_2 \leq L$, then

$$
\sum_{k=1}^{K}\left(R_k(X_*)-R_k(X_k)\right)\leq \frac{1}{2\eta}\|X_1-X_*\|_2^2+\frac{\eta}{2}L^2K,
$$

by setting $\eta=\frac{\|X_1-X_*\|_2}{L\sqrt{K}}$ we obtain

$$
\begin{aligned}
\sum_{k=1}^{K}\left(R_k(X_*)-R_k(X_k)\right)&\leq \frac{1}{2\eta}\|X_1-X_*\|_2^2+\frac{\eta}{2}L^2K\\
&\leq L\|X_1-X_*\|_2\cdot\sqrt{K}.
\end{aligned}
$$

□