Multi-tenant Resource Allocation For Shared Cloud Storage

David Dau Chuen Shue

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

Adviser: Professor Michael J. Freedman

June 2014

Abstract

Shared storage services enjoy wide adoption in commercial clouds. But few systems today provide performance isolation or resource allocation. Misbehaving or high-demand tenants can overload the shared service and disrupt other well-behaved tenants, leading to unpredictable performance and violating tenant service level agreements. Today's approaches for multi-tenant resource allocation are based either on per-virtual machine allocations or hard rate limits that assume uniform workloads to achieve high utilization.

In this thesis, we present a framework for achieving datacenter-wide *per-tenant* performance isolation and fairness in a shared key-value storage system we call Pisces. Pisces achieves per-tenant weighted fair shares of aggregate system resources, even when different tenants' partitions are co-located and when partition demand is skewed, time-varying, or bottlenecked by different server resources. Pisces does so by decomposing the fair sharing problem into four complementary mechanisms —partition placement, weight allocation, replica selection, and weighted fair queuing—that operate on different time-scales and combine to provide system-wide max-min fairness. Our Pisces storage prototype achieves nearly ideal (0.99 Min-Max Ratio) fair shares, strong performance isolation, and robustness to skew and shifts in tenant demand at high utilization.

Although Pisces achieves resource sharing for network-bound workloads, such guarantees have proven elusive for disk-bound workloads. Modern storage stacks often amplify the IO cost of application requests. Interference between tenant IO workloads suppresses throughput and induces variability. Even without interference, IO performance varies non-linearly with operation size. To address these challenges, we present Libra, an IO scheduling framework that provides per-tenant application-request throughput reservations while achieving high utilization for SSD-based storage. Libra tracks per-request IO consumption to provision IO resources based on tenants' dynamic usage profiles. Using an IO cost model based on *virtual IO operations* (VOP), Libra can allocate and schedule tenant IO within the limits of *provisionable* IO throughput. Our LevelDB-based prototype achieves highly accurate tenant app-request reservations and low-level VOP allocations ($>$ 0.95 Min-Max Ratio) over a range of workloads with high utilization.

Bibliographic Notes

This research presented in this thesis is based on work performed in collaboration with my adviser Michael Freedman and Anees Shaikh, formerly of IBM research and now with Google. The framework for providing per-tenant system-wide resource shares introduced in Chapter 3 with detailed model and mechanisms delineated in Chapter 4 appears in papers co-authored with Michael Freedman and Anees Shaikh [64, 65]. Extensions to the framework, described in Chapter 6, for provisioning application request throughput for disk-IO bound workloads on SSD media appears in a paper co-authored with Michael Freedman [63].

Acknowledgements

I would like to thank my adviser, Professor Michael Freedman, for his invaluable guidance and for always managing to extract out the key concepts and most salient points from the often jumbled morass of ideas in my head. His keen eye for system design and his uncanny ability to pinpoint critical issues has left its indelible imprint on my own perspective of systems research and development. I have gone from inexperienced neophyte to seasoned researcher under his expert tutelage.

My committee members — Anees Shaikh, Jen Rexford, Vivek Pai, and Margaret Martonosi — have also been instrumental in shaping both the contents of this thesis and molding my overall research agenda. I would like to especially recognize Anees Shaikh and Jen Rexford as instrumental mentors who have opened my minds eye to the broader realm of systems and networking research and have often been an encouragement to me on this academic journey.

Many thanks go to my colleagues and fellow graduate students who have struggled alongside me in the quest for academic breakthrough and helped hone my ideas through countless discussions and constructive feedback. Special thanks go to Sid Sen, Wyatt Lloyd, Erik Nordstrom, Prem Gopalan, Steve Ko, Matvey Arye and Rob Kiefer. Whether its your technical acumen, brotherly camaraderie, or just simple unbridled fun, you have been friends and colleagues of the highest order.

Most importantly, my thanks, my heart, and my life belong to my family without whom I would neither have made it through this arduous path nor found joy and relevance in the process. To my parents, thank you for your love and continual support even from afar. To Theo and Salina, you have been the most gracious, patient, and helpful in-laws anyone could ask for. Thank you so much for allowing us to invade your lives with the sounds of little footsteps and children's laughter these past 6 years. To my little girls, Hannah and Maya, thank you for your sweet smiles and innocent love for a daddy who is too often been busier than he should be and for patiently waiting until I can finally be your play buddy again. Most of all, I thank my wife and most cherished love, Christine. Without you, I would be a mere shadow of myself. Thank you for loving our kids and supporting me with incredible endurance and continual encouragement. Our journey continues on.

All things are from God and for Him, as are my being and purpose, and this thesis. He got me in, and by His grace, He got me out. May this work honor Him and benefit all who might come across its pages.

The research in this thesis was funded by National Science Foundation CAREER Award #0953197: Towards Scalable Datacenter Services with Strong Robustness Guarantees and the Intel Science and Technology Center (ISTC) for Cloud Computing.

To my precious girls Christine, Hannah, and Maya.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

An increasing number and variety of enterprises are moving workloads to cloud platforms. Whether serving external customers or internal business units, cloud platforms typically allow multiple users, or *tenants*, to share the same physical server and network infrastructure. In these infrastructure-as-a-service (IaaS) environments, tenants specify the number and type of compute resources required in the form of virtual machines (VMs) which the cloud provider then deploys across the datacenter on host servers. Tenant VMs communicate with each other over a shared, possibly virtualized, network topology. With their simple "pay-as-you-go" pricing model, IaaS cloud providers allow tenants to reduce capital expenditure and maintenance costs by leasing resources on an as-needed basis according to current workload demand.

In addition to these virtualized resources, many cloud providers also offer an array of common platform services. Examples of these shared, multi-tenant services include key-value stores [10, 26], block storage volumes [8, 57], SQL databases [6, 45], message queues [9], and notification services. These systems leverage the expertise of the cloud provider in building, scaling, and improving common services, and enable the statistical multiplexing of resources between tenants for higher utilization. Because they rely on shared infrastructure, however, these services face two key, related issues:

- ***Multi-tenant interference and unfairness:*** Tenants simultaneously accessing shared services contend for resources and degrade performance.
- ***Variable and unpredictable performance:*** Tenants often experience significant performance variations, e.g., in response time or throughput, even when they can achieve their desired mean rate [23, 75, 34, 77].

These issues limit the types of applications that can migrate to multi-tenant clouds and leverage shared services. They also inhibit cloud providers from offering differentiated service levels, in which some tenants can pay for performance isolation and predictability, while others choose standard "best-effort" behavior.

For critical workloads that require predictable performance and high Quality of Service (QoS), *provisioned* resources are essential where tenants receive guaranteed resource allocations. Provisioning low-level resources, such as disk IO operations or network bandwidth, is simpler for the provider, but it requires the often opaque and onerous task of resource estimation on the part of the tenant. Tenants only see the high-level requests they issue, not the underlying IO operations and the resources they consume. Instead, tenants prefer to specify an allocation of throughput in terms of *application-level* requests, such as GETs per second.

1.1.1 System-wide Resource Allocation

Shared back-end storage services face different challenges than sharing server resources at the VM level. These stores divide tenant workloads into disjoint partitions, which are then distributed (and replicated) across different service instances. Rather than managing individual storage partitions, cloud tenants want to treat the entire storage system as a single black box, in which *aggregate* storage capacity and application request rates can be elastically scaled on demand. Resource contention arises when tenants' partitions are co-located, and the degree of resource sharing between tenants may be significantly higher and more fluid than with VM resource allocation. Particularly, as tenants may use only a small fraction of a server's throughput and capacity, restricting nodes to a few tenants may leave them highly underutilized.

Provisioning tenant resources and isolating tenant performance at the *service* level is confounded by variable demand to different service partitions. Even if tenant objects are uniformly distributed across their partitions, per-object demand is often skewed, both in terms of request rate and size of the corresponding (read or write) operations. Moreover, different request workloads may stress different server resources (e.g., small read requests may be interrupt limited, while large reads are bottlenecked by bandwidth, and synchronous writes are disk-IO bound) In short, simply assuming that each tenant requires the same proportion of resources per partition can lead to unfairness and inefficiency.

To address these issues, we present a framework for achieving system-wide tenant throughput allocation for cloud storage systems with a high degree of resource sharing and contention. While our mechanisms may be applicable to a range of services that leverage shared-nothing architectures [69] for horizontal scalability, we focus our design and evaluation on a replicated key-value storage service, which we call *Pisces* (*P*redictable *S*hared *C*loud *S*torage). For network-bound workloads, Pisces provisions tenant throughput by targeting global *max-min fairness* with *high utilization*. Under max-min fairness, no tenant can gain an unfair advantage over another when the system is loaded, i.e., each tenant will receive its weighted fair share. Moreover, given its work-conserving nature, when some tenants use less than their full share, unconsumed resources are divided among the rest to ensure high utilization. Although fairness as an allocation metric only provides relative shares of available throughput, Pisces relies on the relatively stable provisioning of system capacity to convert tenants' desired throughput allocation into weighted proportions.

Pisces decomposes the global multi-tenant resource allocation problem into four mechanisms using the Network Utility Maximization framework [18] that correspond directly to the archetypical shared storage architecture. Operating on different timescales and with different levels of system-wide visibility, these mechanisms work in tandem to ensure max-min fairness under resource contention and variable tenant workloads.

(i) Partition Placement (re)-assigns tenant partitions to nodes to ensure a fair allocation (long timescale).

(ii) Weight Allocation distributes overall tenant fair shares across the system by adjusting local per-tenant weights at each node (medium timescale).

(iii) Replica Selection load-balances requests between partition replicas in a weight-sensitive manner (real-time).

(iv) Fair Queuing enforces performance isolation and fairness according to local tenant weights (real-time).

We first describe the overall architecture of the system in chapter 3 and dive into the specific Pisces mechanisms In chapter 4. In chapter 5, through an extensive experimental evaluation we demonstrate that Pisces significantly improves the multi-tenant fairness and isolation properties of our key-value store, built on Membase [20], across a range of tenant workloads. We also show that its replica selection and rebalancing policies optimize system performance, even as workloads shift dynamically.

1.1.2 Local Storage Node Resource Allocation

Reserving app-level request rates, rather than low-level IO resources, is more intuitive for tenants. Multiple sources of complexity in the storage stack make it difficult to provision app-level requests for disk-IO bound workloads:

- *IO operations are subject to amplification:* In modern storage engines, a single application-level request can trigger *multiple* IO operations (e.g., a 1KB PUT written once to a log and then to a data table) which can vary non-uniformly with tenant workload distribution (e.g., by the GET/PUT ratio and request size).

- *IO throughput degrades under interference:* Disk-level IO interference between reads and writes can cause severe degradation in IOP/s and IO bandwidth, which also varies unpredictably with tenant op size and workload.
- *IO cost varies non-linearly with operation size:* Even for pure read or write workloads, IOP/s and bandwidth vary non-linearly with operation size, shifting bottlenecks from the controller (IOP) to the data channel (bandwidth) as IOP sizes increase.

The local fair-queuing mechanisms in Pisces largely address the resource sharing needs of network-bound workloads with asynchronous persistence, enforcing the local policies established by the system-wide mechanisms to provide predictable throughput over the entire distributed key-value store. Tenants that require synchronous writes or induce frequent cache-misses (e.g., uniform reads or scans across a large key space), however, contend for disk-IO resources. Existing approaches typically use hard rate limits [10, 8] to provision tenant app-request reservations. While simple and effective for performance isolation, a significant portion of IO resources may lie fallow if tenant demand drops below the reserved request rates. This is untenable for cloud storage providers who must capitalize on every bit of available performance due to competitive pricing pressure.

In chapter 6 we present Libra, an IO resource scheduling framework for provisioning local app-request throughput in a multi-tenant key-value storage system backed by high-throughput, low-latency SSD media to provide predictable IO performance for disk-bound workloads, while preserving high utilization. Libra provides the crucial per-node substrate for achieving system-wide tenant throughput reservations specified in terms of size-normalized (1KB) GET and PUT requests per second. At each storage node, Libra *provisions* low-level IO resource *allocations* to satisfy each tenant's local throughput *reservation* by tracking application request cost and mediating tenant IO resource consumption.

Libra is able to provision at least half of the maximum (i.e., interference-free) SSD IO throughput to achieve the tenants' per-node application-level throughput reservations in our LevelDB-based [37] prototype storage node. Libra achieves these reservations over a wide range of workloads including intermixed reads, writes, and varying IOP sizes, with highly variable interference effects. Although up to half the IO resources may be left unprovisioned, Libra still preserves high utilization by allowing tenants to share any excess IO throughput in a work-conserving manner.

Taken together, Pisces and Libra provide a comprehensive framework for achieving predictable, provisioned performance in multi-tenant shared cloud storage systems with high utilization and arbitrary tenant workloads. In the chapter 2, we first provide the necessary background to understand the NUM optimization framework used to model and design Pisces, and the fair-queuing scheduler underpinning Libra. We then introduce the general multi-tenant storage architecture in chapter 3 followed by the Pisces framework and its evaluation in chapters 4 and 5. Chapter 6 describes the design and implementation of the Libra scheduler with evaluation in chapter 7. This thesis concludes in chapter 8 with a discussion of possible directions for future work and a summary of contributions.

Chapter 2

Background

2.1 Layering As Optimization Decomposition

A common rule of thumb in system design is to aim for simplicity and introduce complexity only when necessary. However, when this rubric is applied to system performance and optimization it often results in sub-optimal heuristic-based solutions [35]. These loose collections of parameters and thresholds, usually determined by engineering intuition and extant empirical data, can leave the system susceptible to unforeseen load dynamics and often requires extensive manual tuning [66]. Moreover, while local properties may be preserved, in increasingly distributed and multi-tier systems, it is unclear how the system as a whole will behave.

Instead, performance-minded system design should be guided by and, when possible, derived from an optimization framework [51]. Optimization modeling not only allows the designer to specify system-wide performance goals (objectives) but also understand their relationship with potential bottlenecks (constraints). By decomposing the global optimization problem into layered sub-problems, the framework can also identify implicit assumptions and trade-offs in both the "vertical" layering of control between control modules and "horizontal" distribution of computation across the architecture and suggest alternative designs [18]. Taken a step further, the decomposed sub-problems often admit simple distributed gradient-based algorithms with convergence guarantees that can adapt to real-time workload fluctuations.

In this section, we briefly explore the core concepts behind convex optimization and decomposability that we use in Pisces to achieve system-wide tenant resource allocations. We leverage both primal and dual decomposition (2.1.2) to split the global allocation problem into multiple mechanisms that are distributed across the system. The majority of the material in this section is taken from [51].

2.1.1 Global Convex Optimization

Convex optimization is a well-understood method for analyzing and optimizing a representative model of system behavior. Convexity itself is generally considered the watershed [15] for computationally tractable optimization. If a problem exhibits convexity, then it is (relatively) easy to solve. Non-convexity on the other hand, often falls into the realm of NP-hard problems. Even for convex problems, however, using a polynomial time centralized algorithm may still be prohibitive in a distributed system if control variables and input parameters (data) scale with and are spread across the system's elements. The communication delay and computational overhead of "backhaul" solutions often force system designers to find a simpler, more distributed solution. Thus, most of the heavy-lifting in convex optimization for large, networked systems involves finding the right system model that offers both vertical and horizontal decomposability and suitable algorithms to solve the various sub-problems in a distributed fashion. Before we turn to distributed solutions, however, we must first begin our discussion with the global optimization model.

A convex optimization problem can be formulated as:

$$\textbf{minimize: } f(x) \tag{2.1}$$

$$\textbf{subject to: } g_i(x) \leq 0 \text{ where } 1 \leq i \leq m \tag{2.2}$$

$$h_j(x) = 0 \text{ where } 1 \leq j \leq p \tag{2.3}$$

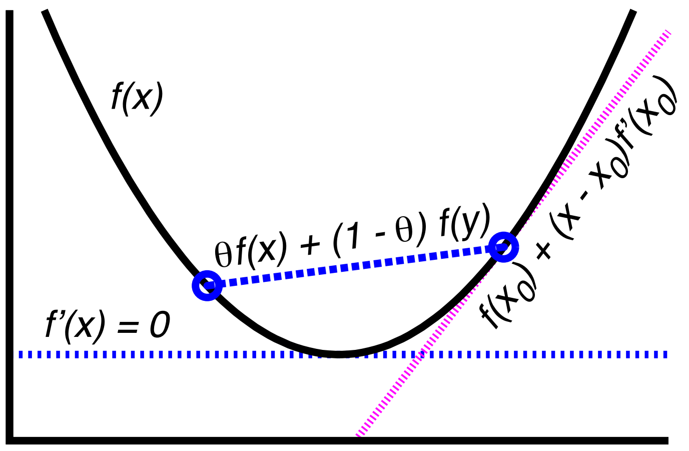


Figure 2.1: A convex function is always less than or equal to the linear interpolation between any two points on the curve. For differentiable convex functions, the gradient at any point is a global underestimate and the global minimum either exists or is $-\infty$

.

where $x \in \mathbb{R}^n$ is the (possibly multivariate) decision variable, f is the objective function to minimize, g_i are the m inequality constraint functions, and h_j are the p equality constraint functions. For the optimization problem to be *convex*, the objective (2.1) and inequality constraints (2.2) must be convex while the equality constraints (2.3) should be linear or affine. For a maximization problem, the objective must be *concave*. A point x in the domain χ of f is *feasible* if it satisfies all g_i inequality and h_j equality constraints, i.e., it lies in the convex set defined by the constraints, otherwise it is *infeasible*. A feasible optimization problem must admit at least one feasible point. The *optimal* (i.e., minimal or maximal) value f^* is achieved at the optimal point x^* where $f^* = f(x^*)$.

The basic definition of convexity for a real-valued function f is given by Jensen's inequality, illustrated in Figure 2.1

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)\ \forall x, y \in \chi \text{ and } \theta \in [0,1] \tag{2.4}$$

which states that for any point $\theta x + (1 - \theta y)$ on the line segment connecting any two points x and y in the (convex) domain χ of f, the value of f must be less than or equal to the linear interpolation between $f(x)$ and $f(y)$. In other words, much like the definition of a convex set [15], a convex function forms a partial “hull” around these interpolated “interior” points that never bulges inward. For a concave function, f must be greater than or equal to the linear interpolation. A more useful definition of convexity for optimization purposes restates the inequality in terms of first and second derivatives

$$f(x) \geq f(x_0) + (x - x_0)f'(x_0) \text{ and } f''(x) \geq 0 \ \forall x_0, x \in \chi \tag{2.5}$$

when f is a (twice) differentiable function. This definition implies that either a global minimum exists or is equal to $-\infty$, unless bounded by a constraint, since $f''(x)$ is non-negative. Moreover, because of this curvature, the gradient $f'(x)$ taken at any point x_0 is a global underestimate of f. Thus, for a feasible convex optimization problem, there must be a unique global optimum that is equivalent to the local optimum, where $(f'(x) = 0)$, which can be easily found using gradient descent. Given these properties, it is no surprise that there are numerous algorithms that can solve the global convex optimization problem in polynomial time.

The key to the power and simplicity of convex optimization for decomposing global problems into distributed solutions lies in its strong Lagrange duality. Lagrange duality transforms the original “primal” minimization problem into an often simpler “dual” maximization problem and vice versa for primal maximization problems. The premise behind the Lagrangian is to relax the primal constraints by merging them with the objective function (2.1) using *Lagrange multipliers* λ_i and ν_j.

$$L(x, \lambda, \nu) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \nu_j h_j(x) \tag{2.6}$$

$$G(\lambda, \nu) = \inf_{x} L(x, \lambda, \nu) \tag{2.7}$$

These dual variables represent the penalizing weights or *prices* imposed by their respective constraints on the primal objective function. The goal of the dual problem is to find the set of prices that optimize the original objective. The first step is to optimize the Lagrangian L over x to obtain the dual objective function G (2.7), which effectively computes the optimal x given a set of dual prices [1]. Minimizing L is much simpler in the dual formulation since the constraints have been absorbed. However, the optimization path taken by x may be infeasible, since the constraints no longer hold.

The dual maximization problem to find the optimal set of prices λ^* and ν^*

$$\begin{aligned} \textbf{maximize: } & {}_{\lambda,\nu}G(\lambda,\nu) \\ \textbf{subject to: } & \lambda \geq 0 \end{aligned} \tag{2.8}$$

yields an optimal dual value G^* that is a lower bound on the primal optimum f^*. In general, the *duality gap*: $G^* - f^* \geq 0$ between the dual and primal optima is non-negative, with positive gap indicating weak duality, i.e., the pricing strategy is too weak to fully enforce the constraints. Under mild restrictions, convex problems satisfy the KKT conditions [15], which are necessary and sufficient to achieve zero duality gap. This strong duality admits an alternative dual path to solving the primal problem. Using the dual form often simplifies the problem structure by decoupling jointly constrained variables, allowing for more independent, distributed computation of the optimal values. We explore the benefits and limitations of using the primal and dual approaches for decomposing the global optimization problem in the next section.

2.1.2 Optimization Decomposition

Decomposition entails breaking the global optimization problem into a hierarchy of subproblems with a "master" problem at the top and "slave" (or sub-master) problems fanning

[1] The dual objective G is concave even if f is not convex since it is the pointwise infimum of a family of affine functions, indexed by x, of the dual variables λ and ν

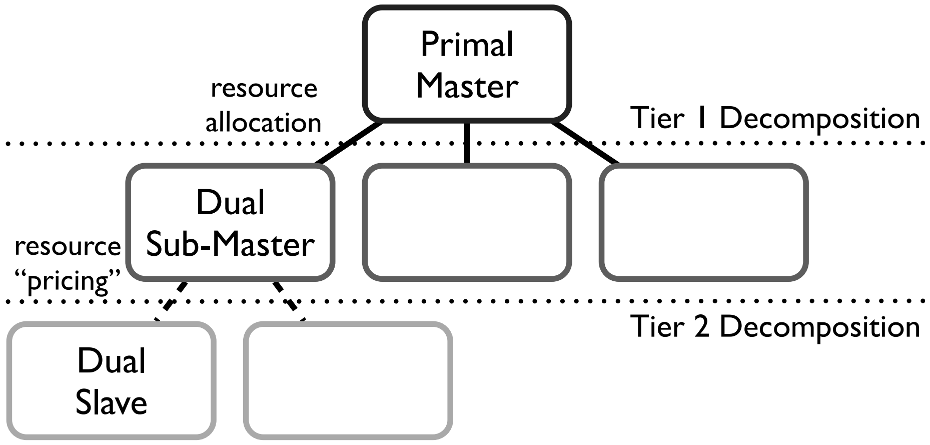


Figure 2.2: An example of a two-tier primal-dual decomposition. Vertical edges denote coordination channels to disseminate optimal sub-problem solutions. Horizontal decoupling implies independent distributed computation. Primal decomposition generally requires explicit signaling of master resource allocations. Dual decomposition, however, can often exploit implicit (dashed lines) measurements of sub-problem demand and master "pricing". Lower tiers must complete sub-problem optimization at faster timescales.

out underneath. The master-slave relationship denotes a vertical division of functionality where the master optimizes higher-level control variables that parameterize the operation of the lower-level slave mechanisms. Figure 2.2 depicts an archetypical two-tier decomposition. Splitting a problem across multiple sibling slaves corresponds to a horizontal distribution of the computation where the sub-problems optimize their variables at a faster timescale, which then become input to the coordinating master problem. Master-slave connections reveal the (implicit or explicit) coordination channels needed to convey optimization results up and down the hierarchy. Sibling slaves, on the other hand, can compute their sub-problems independently of each other without the need for interaction. The two main techniques for decomposing a convex optimization problem involve breaking apart either the primal problem (primal decomposition) or the dual problem (dual decomposition).

In primal decomposition, the master problem *directly* allocates resources to the sub-problems according to their needs. In contrast, the dual decomposition master problem sets a resource "price" to *indirectly* affect resource consumption by penalizing the sub-problems that over-spend on a bottleneck resource while incentivizing others to consume under-utilized capacity. Although the dual approach often yields highly decoupled sub-problems amenable to distributed computation, it also requires some flexibility in the physical system to allow for transient infeasibility, i.e., inequality constraint (2.2) violations. For example, in a network setting links have output buffers to absorb occasional bandwidth overruns. On the other hand, primal techniques tend to require more master-slave coordination, but they ensure that all intermediate steps taken to reach the optimum are feasible in the physical system.

Dual decomposition:

Generally, dual decomposition is best suited for relaxing inequality constraints that couple together otherwise independent decision variables. Take the canonical NUM problem [18] (i.e., TCP) of maximizing send rates in a communication network

$$\begin{aligned} \textbf{maximize:}\ & \sum_{s} U_s(x_s) \\ \textbf{subject to:}\ & \sum_{s:l\in L(s)} x_s \le c_l \ \forall\, l \in L \end{aligned} \tag{2.9}$$

where S sources send data at rate x_s over L links with capacity c_l according to a fixed routing with path p_s. The utilities U_s for the source send rates are twice-differentiable, increasing, and strictly concave functions which we can generalize as $U_s(x_s) = \int f_s(x_s)\, \mathrm{d}x_s$ where f_s is differentiable and invertible. Many typical utilities (e.g., log) fall in this category. Clearly, without the link capacity constraints (2.9), x_s would decouple and each source node could optimize their send rate independently to arrive at the global optimum.

The goal of dual decomposition is to split the dual problem into two levels where both the master and slave computations can be distributed across the network elements. At the top level, the master problem sets link congestion "prices" λ_l according to the load it sees at each link. These prices force the slave sub-problems at each source node to adjust their send rates with respect to the capacity constraints along their path. This decomposition follows directly from the dual problem by first maximizing the Lagrangian with respect to

x_s (slave sub-problems) to obtain the dual objective in (2.7) and then minimizing over the dual variables (master problem) in (2.8) to find the global optimum. To decouple the slave sub-problems, we first form the Lagrangian:

$$
\begin{aligned}
L(x, \lambda) &= \sum_s U_s(x_s) + \sum_l \lambda_l \Big(c_l - \sum_{s:l \in L(s)} x_s \Big) \\
&= \sum_s L_s(x_s, \lambda^s) + \sum_l c_l \lambda_l
\end{aligned}
$$

By relaxing the link capacity constraints, the Lagrangian allows us to group the terms by source send rate x_s into partial Lagrangians $L_s = U_s(x_s) - \lambda^s x_s$, where $\lambda^s = \sum_{l:l \in p_s} \lambda_l$ is the path congestion price for s. Given λ^s, these slave sub-problems can then independently optimize their send rates at the source nodes.

$$
x_s^* = f^{-1}(\lambda_s) \text{ when } \frac{\partial L_s}{\partial x_s} = 0 \ \forall \ x_s \tag{2.10}
$$

$$
x_s(t+1) = x_s(t) + \alpha(f(x) - \lambda_s) \tag{2.11}
$$

In most cases, sources compute the optimal send rate using the gradient ascent update in (2.11) with step size α rather than in the single shot optimization (2.10). Note how λ_s is the "price" source s has to pay for each byte of data it sends, balanced against its utility. For a utility function like log with diminishing returns, the optimal send rate $x_s^* = \frac{1}{\lambda_s}$ results in proportional bottleneck fairness.

Once the slave sub-problems have computed the optimal send rates for the current congestion prices, the master problem readjusts the prices by minimizing the dual objective

$$
\begin{aligned}
&\textbf{minimize: } G(\lambda) = \sum_s U_s(x_s^*) - \lambda^s x_s^* + \sum_l c_l \lambda_l \\
&\textbf{subject to: } \lambda_l \geq 0 \ \forall \ l \in L
\end{aligned} \tag{2.12}
$$

To distribute and solve the master dual problem at each link l, we take the partial derivative of G with respect to each link price λ_l

$$\frac{\partial G}{\partial \lambda_l} = c_l + \sum_{s:l\in L(s)} f(x_s^*) f_s^{-1'}(\lambda_s) - x_s^* - \lambda_s f_s^{-1'}(\lambda_s) = c_l - \sum_{s:l\in L(s)} x_s^*$$

$$\lambda_l(t+1) = \lambda_l(t) - \alpha\Big(c_l - \sum_{s:l\in L(s)} x^* s\Big) \tag{2.13}$$

Thus, given the optimal send rates x_s^*, each link's dual optimization of λ_l can be independently computed using the gradient descent update in (2.13) . The true beauty of this decomposition lies in the interpretation of the optimal primal x_s^* and dual λ_l^* variables as physical phenomena that can be implicitly measured rather than explicitly signaled by the master and slave sub-problems. Each link measures the optimal send rates as the aggregate traffic traversing the channel while nodes detect the optimal link prices as queuing delay along its path. If sources attempt to overload link capacity, queuing delay quickly rises to quell the surge, while underutilized links allow traffic to increase unpenalized.

Primal decomposition:

Instead of using dual variable prices to incentivize behavior, the master problem in primal decomposition directly allocates primal variables to decouple and distribute slave sub-problem computation. This allocation can be viewed as "virtualizing" or "slicing" resources between principals (tenants). To accomplish this, primal decomposition often involves introducing auxiliary variables to fully decouple computations. Consider the NUM problem in (2.9) but with additional min and max QoS bandwidth constraints for Q classes.

$$\begin{aligned} \textbf{maximize:} \ & \sum_s U_s(x_s) \\ \textbf{subject to:} \ & \sum_{s:l\in L(s)} x_s \le c_l \ \forall\, l \in L \\ & q^i_{\min} \le \sum_{s:s\in S(i)} x_s \le q^i_{\max} \ \forall\, i \in Q \end{aligned} \tag{2.14}$$

Here, the send rates x_s are not only coupled by shared link capacity constraints, but also by the QoS class bandwidth bounds ($q^i_{\min}$, $q^i_{\max}$). To decouple the problem, we add auxiliary variables y^i_l to represent the per-class aggregate bandwidth at each link.

$$\begin{aligned}
\textbf{maximize:}\ & \sum_s U_s(x_s) \\
\textbf{subject to:}\ & \sum_{s:l\in L(s)\wedge s\in S(i)} x_s \le y^i_l\ \forall\ i \in Q \text{ and } l \in L \\
& \sum_i y^i_l \le c_l\ \forall\ l \in L \\
& q^i_{\min} \le y^i_l \le q^i_{\max}\ \forall\ i \in Q \text{ and } l \in L
\end{aligned} \tag{2.15}$$

Now we can apply primal decomposition to break the problem into a primal master problem that directly allocates per-class link bandwidth y^i_l and a dual slave problem that solves the decoupled send rate NUM sub-problem for each class i

$$\begin{aligned}
\textbf{maximize:}\ & \sum_{s:s\in S(i)} U_s(x_s) \\
\textbf{subject to:}\ & \sum_{s:l\in L(s)\wedge s\in S(i)} x_s \le y^i_l\ \forall\ l \in L
\end{aligned} \tag{2.16}$$

by applying the dual decomposition in (2.1.2) given the current class bandwidth allocations.

The master problem, on the other hand, optimizes the per-class bandwidth allocations in response to these optimal send rates.

$$\begin{aligned}
\textbf{maximize:}\ & \sum_i U^*_s(y^i) \\
\textbf{subject to:}\ & \sum_i y^i_l \le c_l\ \forall\ l \in L \\
& q^i_{\min} \le y^i_l \le q^i_{\max}\ \forall\ i \in Q \text{ and } l \in L
\end{aligned}$$

Since the utilities U_s^* only depend indirectly on y^i via the per-class bandwidth constraint in (2.16), the optimal per-class allocations must be solved with the help of the dual per-class prices λ_l^i. Taking the partial derivative of the dual sub-problem objective in (2.12) with respect to y_l^i gives independent *subgradients* $\frac{\partial G}{\partial y_l^i} = \lambda_l^{i*}$ that can be used in an iterative update

$$y_l^i(t+1) = [y_l^i(t) + \alpha \lambda_l^{i*}(y_l^i(t))]_y \tag{2.17}$$

where α is the step size and $[\dots]_y$ refers to the projection of the update onto the set of feasible y_l^i. In other words, if the link congestion price for class i is high under the current allocation $y_l^i(t)$, then increase it. Otherwise, if it is not being fully used (small λ_l^{i*}), then decrease it. In many cases, primal decomposition requires more explicit signaling between system components to measure congestion and set the resource allocations. In this case, however, the allocations are all link-local and can be computed independently without additional coordination using the same implicit measurements as in (2.1.2).

2.1.3 Distributed Algorithms and Convergence

Applying decompositions in a single tier (2.1.2) or hierarchically (2.1.2), breaks the original global optimization into distributed sub-problems that run on different timescales. Sub-problems at the same level run concurrently with little, if any, coordination between them. Between levels, slave sub-problems must first complete their optimizations before the master problem can optimize its allocations or prices in response. These two computation schedules, parallel across a level and sequential between, correspond to the two main distributed optimization techniques, Jacobi and Gauss-Seidel.

The nonlinear Jacobi algorithm [15] iteratively optimizes a subset K of decision variables out of n total variables in parallel while keeping the rest fixed

$$x_i^{t+1} = \operatorname*{argmin}_{x_i \in \mathcal{X}_i} f\Big(x_1^t, \dots, x_{i-1}^t, x_i, x_{i+1}^t, \dots, x_n^t\Big) \; \forall\, i \in K \tag{2.18}$$

where f is the objective function and χ_i represents the convex constraint set for x_i. Note that the values of all other variables $x_{j \neq i}$ are fixed at time step t. On the other hand, the nonlinear Gauss-Seidel algorithm [15] (block coordinate descent) iteratively optimizes a subset K of decision variables in sequential (circular) fashion while keeping the rest fixed

$$x_i^{t+1} = \underset{x_i \in \chi_i}{\operatorname{argmin}} \, f\Big(x_1^{t+1}, \ldots, x_{i-1}^{t+1}, x_i, x_{i+1}^t, \ldots, x_n^t\Big) \;\forall\; i \in K \tag{2.19}$$

Here, variables x_1 through x_{i-1} have already been optimized at the new time step $t + 1$.

For convex optimization problems, both the Jacobi (2.18) and Gauss-Seidel (2.19) algorithms converge to the global optimum of f over χ as long as they fully minimize the variables x_i at each step t in the iterative sequence [15]. This is why timescale separation is essential between master and slave sub-problems to allow the slaves to fully optimize their variables before the master step in Gauss-Seidel. In a single iteration, Jacobi and Gauss-Seidel are free to use single-shot, gradient (2.11), or subgradient (2.17) update algorithms to compute the optimal x_i^{t+1}. Convergence proofs for these algorithms can be found in [15]. Every decomposition has its own performance characteristics, communication overhead, and convergence time. These properties depend on the vertical and horizontal decomposition, implicit or explicit signaling, and per-subproblem optimization techniques. Where one decomposition architecture falls short in efficiency, another may unlock hidden savings. In chapter 4 we introduce the global optimization model of system-wide per-tenant max-min fairness and the particular decomposition used in Pisces to separate functionality and distribute the optimization across the storage system and their merits.

2.2 Resource Scheduling

For as long as there have been multi-user operating systems, shared network topologies, and interleaved user-space disk IO, resource scheduling has been essential for mitigating resource contention and achieving predictable performance. In Pisces and Libra, resource scheduling plays a crucial role in enforcing per-tenant allocations at a local storage node.

Taken together, these local allocations achieve system-wide tenant shares. In this section, we first define resource sharing in high utilization settings in terms of max-min fairness and then discuss the idealized fluid-model resource scheduler and its approximations.

Statistically multiplexing a shared resource among multiple users increases utilization and overall efficiency. Unused resources left fallow by one user can be consumed by another. However, left unchecked this resource sharing can be "unfair", i.e., one user's workload may violate the performance guarantees of another. Although static rate-limiting (hard allocations) provides strong performance isolation and is simple to implement, it reduces the effective service rate of the underlying resource, which can be prohibitive in high utilization environments. Instead, the resource scheduler should preserve user shares in a work-conserving manner. As long as there are eligible tasks available, the scheduler should service them. The key, then, is to schedule user tasks in such a manner that each user receives its desired resource rate over a given interval.

For a fixed rate resource (e.g., link bandwidth), resource allocation can be achieved by proportional shares, or equivalently, weighted *max-min* fair shares. Under max-min fairness, each user (or class) $i \in Q$ with user demand u_i receives service rate r_i in proportion to its weight w_i and resource capacity c

$$\begin{aligned} \textbf{maximize:} \quad & \sum_i \min(u_i, \pi_i v) \\ \textbf{subject to:} \quad & \sum_i \min(u_i, \pi_i v) \le c \end{aligned} \tag{2.20}$$

where $\pi_i = \frac{w_i}{\sum_k w_k}$ is the normalized user weight, v is the max-min fair share, and $r_i = \min(u_i, \pi_i v)$ is the computed service rate. If user demand, user weight, and resource capacity are fixed in the scheduling interval, then this maximization is convex and admits a simple "water-filling" solution. Imagine a container with $|Q|$ user sections where each section has base area proportional to the user's normalized weight and total volume equal to the user's demand. If we "pour" in the resource, up to c, from the tallest section, it will fill up each section equally up to its max volume. Excess “fluid” continues to fill up the larger sections until the container either overflows ($\sum_i u_i < c$), or we deplete the resource ($\sum_i u_i \geq c$). The “fill-line” corresponds to the optimal fair-share v. In other words, under max-min fairness, each user will receive at least its weighted share of the resource, if it has sufficient demand, and a proportional share of any unused excess. This provides both the desired user shares $\pi_i c$ and performance isolation since excessive demand from mis-behaving users cannot eat into other users’ shares.

2.2.1 Generalized Processor Sharing

Generalized processor sharing (GPS) is an idealized scheduler first introduced in the context of network resource sharing [53] to formulate and analyze the behavior of a fluid-model max-min fair scheduler. A GPS scheduler services competing users (e.g., flows) by selecting a user task (e.g., packet) from the set of eligible (non-empty) user queues and servicing it in proportion to the users’ weights. For network resource sharing, this is equivalent to scheduling flows on a bit-by-bit basis, i.e., pouring bits into the flow containers. Say flows A, B, and C should receive proportional shares of $1/2$, $1/3$, and $1/6$ of the link bandwidth respectively. When all flows are backlogged (i.e. have unsatisfied demand), the GPS scheduler should interleave 3 bits from A, 2 bits from B, and 1 bit from C in round-robin fashion to ensure that each flow receives its fair share over all time intervals.

More formally, let $S_i(\tau, t)$ be the service rate (amount of data served) for user i over time interval $(\tau, t]$ in a system with capacity c and N users. A user i is backlogged at time t if there is work available in its queue. A GPS scheduler is one that provides the following max-min fair service time over interval $(\tau, t]$ for each user i

$$S_i(\tau, t) = \int_{\tau}^{t} f_i(t)dt$$

$$f_i(t) = \frac{w_i}{\sum_{j \in B(t)} w_j} \text{ when } i \text{ is backlogged, otherwise } f_i(t) = 0$$

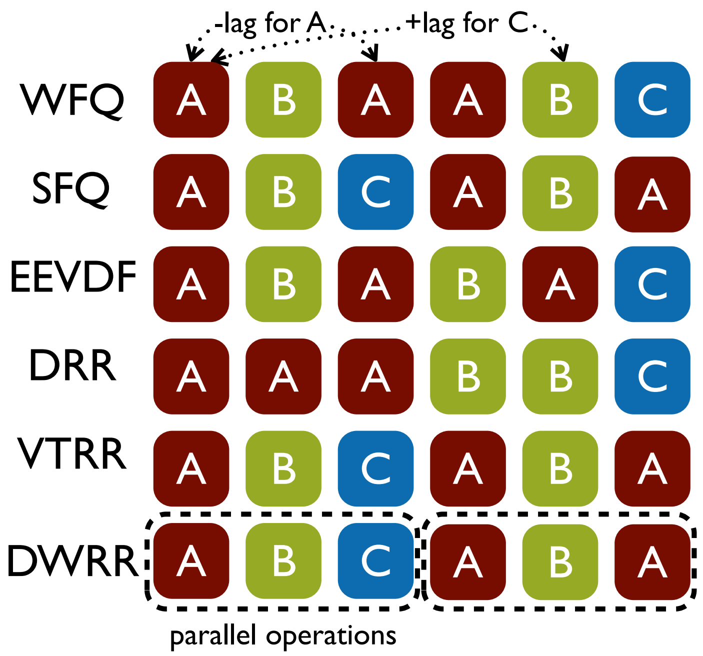


Figure 2.3: GPS approximations achieve proportional shares with alternate scheduling orders, resulting in different *lag* bounds. Tenants A, B, and C have weights 3, 2, and 1 respectively and issue unit cost requests commensurate with their shares. Note how VTRR behaves much like SFQ in this instance and EEVDF has the most consistent lag. DWRR performs operations in parallel and achieves bounded lag over each scheduling round.

where $f_i(t)$ is the instantaneous fair share of user i which depends on the set of backlogged users $B(t)$ at time t. Under the GPS model, the underlying resource is assumed to be fluid where user tasks can be subdivided and serviced in infinitesimally small chunks (e.g., bits).

In real systems, however, resource schedulers can only approximate the ideal GPS behavior. Due to efficiency concerns or physical constraints schedulers generally service user tasks using fixed (e.g., cpu time slice) or lower-bounded (e.g., logical disk block size) resource quanta or they are required to treat user tasks as indivisible operations (e.g., datagram packet). The accuracy of these approximations, in terms of fairness, can be measured as the *lag* between the ideal service time $S_i(\tau, t)$ and the actual service time $s_i(\tau, t)$ for user i provided by the resource scheduler over the interval $(\tau, t]$

$$\text{lag}_i(t) = S_i(\tau, t) - s_i(\tau, t) \tag{2.21}$$

Scheduler Parameters			
c	resource capacity	$l^{\max}$	maximum length or size of any task
w_i	weight of user i	q	size of minimum resource quanta
Scheduler Input			
Q	total number of users in the system	p_i^n	nth task for user i
$B(t)$	number of backlogged users at time t	l_i^n	length or size of the nth task
Scheduler Variables			
$S_i(t_0, t)$	max-min fair service time for user i	S_i^n	start time of the nth task for user i
$s_i(t_0, t)$	actual service time for user i	F_i^n	finish time of the nth task for user i
r_i	max-min fair rate for user i	$v(t)$	virtual time at real time t

Table 2.1: Max-min fair scheduler model

	Virtual Time	**Start Time**	**Ordering**	**Lag** $(+, -)$	**Overhead**
WFQ	$\frac{dv}{dt} = \frac{c}{\sum_{i \in B(t)} r_i}$	$\max(v(t), F_i^{n-1})$	$\min_i F_i^{\text{next}}$	$\frac{l^{\max}}{c}$, $O(Q)$	$O(\log Q) + O(Q)$
SFQ	$S_{\text{current}}(t)$ if busy $F_{\text{last}}(t)$ if idle	$\max(v(t), F_i^{n-1})$	$\min_i S_i^{\text{next}}$	$2\frac{l^{\max}}{c}$,$O(Q)$	$O(\log Q)$
EEVDF	$v(t)+\frac{S_i(t_0^i,t)-s_i(t_0^i,t)}{\sum_{j \in B(t)} w_j}$	$v(t_0^i) + \frac{s_i(t_0^i,t)}{w_i}$	$\min\limits_{i:S_i<t} F_i$	q,$l^{\max}$	$O(\log Q)$

Table 2.2: Virtual time based scheduler approximations of the ideal max-min fair GPS scheduler.

If the scheduler gives user i less than its fair share, then lag will be positive. Otherwise, if user i is able to consume more than its fair share, lag trends negative. Zero lag indicates exact max-min fair allocation. In the next two sections we examine the two basic types of GPS approximations, those based on virtual time and those using resource quanta, and several representative schedulers of each type. Figure 2.3 illustrates how these different approximation techniques achieve the same overall fairness over a set of tenant requests, but with differing lag bounds over the scheduling interval. As we shall see, schedulers often have to make a trade-off between computational efficiency and strict (minimal lag) allocation fairness.

2.2.2 Virtual Time-based Approximations

Virtual time (VT) based schedulers approximate GPS behavior by scheduling user tasks in the same order (i.e., at the same virtual time) as the GPS scheduler using an estimate of their virtual *start* and *finish* times. Table 2.1 defines the key parameters, inputs, and variables used by VT schedulers. If the n^{th} task (e.g., packet) p_i^n of user i (e.g., flow) i arrives at time t in the user's service queue, then a VT scheduler computes the task's virtual start S_i^n and finish F_i^n times as

$$\begin{aligned} S_i^n &= g(v(t), F_i^{n-1}) \; \forall\, n \geq 1 \text{ where } S_i^0 = 0, \text{ and } g(a, b) \geq a, b \\ F_i^n &= S_i^n + h(p_i^n, r_i) \; \forall\, n \geq 1 \text{ where } F_i^0 = 0 \end{aligned} \tag{2.22}$$

In other words, each user task should start at a virtual time S_i^n, defined by the function g, which must be greater than or equal to the current virtual time and the finish time of the user's previous task. Similarly, each task should finish its service cost h at the same virtual time F_i^n as in the GPS scheduler, at the user's max-min fair service rate r_i. Using these virtual times,the VT scheduler decides which task to service next based on the earliest virtual start time [27], finish time [21], or some combination of the two [67]. Note that since the VT scheduler must find the user task with the minimum virtual time, each scheduling operation takes at least $O(\log Q)$ time, i.e., find-min in a priority queue over all Q users, which can be expensive for high-throughput systems.

How virtual time $v(t)$ is computed has a large impact on scheduling efficiency, while the functional form of start time $g(v, F)$ can affect fairness. Table 2.2 summarizes the functional and behavioral differences between the three representative VT schedulers we examined. Weighted Fair Queuing [21] (WFQ) was the first bounded-error VT approximation to GPS for network resource sharing. WFQ updates virtual time by emulating round progress in the GPS bit-by-bit scheduler

$$\frac{dv}{dt} = \frac{c}{\sum_{i \in B(t)} r_i} \tag{2.23}$$

Since it is work-conserving, the GPS scheduler must advance a round if all active flows (users) $i \in B(t)$ have been serviced and at least one backlogged flow exists. Thus round duration, and hence virtual time, contracts when fewer flows are active and dilates when more flows are backlogged. Using this definition of virtual time, WFQ computes the virtual start time of a packet as the max of the current virtual time and the finish time of the flow's previous packet $g = \max(v(t), F_i^{n-1})$. This avoids penalizing other flows for consuming bandwidth while the current flow was idle. WFQ uses a simple packet transmission time cost model $h = l_i^n / r_i$, which depends solely on packet length l_i^n and the flow's allocated rate r_i, to compute virtual finish time and mimic the service rate of the GPS scheduler. Thus, by scheduling packets in ascending virtual finish time order, WFQ services flows in nearly the same sequence as GPS and achieves bounded positive lag. Negative lag, however, can be linear in the number of backlogged users [53] since packets scheduled to finish at the same time in GPS must be issued in some particular order in WFQ, with the earliest such packet inducing the worst lag. The main drawback of the WFQ scheduler is the overhead of computing virtual time in terms of GPS flow events. To compute (2.23), WFQ needs to maintain an event queue that tracks the arrival and departure of backlogged flows from $B(t)$. In the worst case, WFQ may need to fully process the entire $O(Q)$ event queue on each packet transmission.

Start-time fair queuing (SFQ) minimizes this computational burden by relaxing the virtual time approximation. As shown in Table 2.2, SFQ sets virtual time to be the start time of the current packet being serviced $S_{\text{current}}(t)$ during a busy period, or the maximum finish time $F_{\text{last}}(t)$ of any packets that have completed by time t if the resource is idle. By estimating $v(t)$ in this way, SFQ reduces the overhead to simply tracking the current start and last finish times. This estimate also decouples virtual time from the resource capacity c, enabling SFQ to provide max-min fair (but not absolute) shares even if the capacity fluctuates over time, whereas the WFQ virtual time emulation assumes a fixed capacity [27]. As its name suggests, SFQ services packets by earliest virtual start time rather than finish time, which gives it a delay bias towards low-throughput flows and bounded positive lag similar to WFQ. However, like WFQ, SFQ still exhibits worst-case $O(n)$ delay (negative lag).

The earliest eligible virtual deadline first (EEVDF) scheduler changes both $v(t)$ and start time g to improve the lag bounds while maintaining low overhead. EEVDF defines the start time for user task p_i^n in terms of the virtual time at the start of user i's busy period $v(t_0^i)$ and the actual service time $s_i(t_0^i, t)$ the user received up to the current time t, as shown in Table 2.2. If the service lag for user i is positive, then the start time of the task will be less than the virtual time $v(t)$ and the task is *eligible* for execution. Otherwise, negative lag delays the start time after $v(t)$, making it ineligible until some time in the future. EEVDF uses this notion of eligibility to service eligible users tasks in ascending finish time order, which achieves bounded positive and negative lag. Since EEVDF is a CPU scheduler, it also needs to handle the case where a task uses less than its allocated service time due to early termination, preemption, or IO wait. EEVDF captures this in the cost model $h = \frac{u_i^n}{r_i}$ used for (2.22), where u_i^n is the *actual* service time consumed. In general, this post-execution computation of F_i^n is necessary when resources can be partially consumed or when the total demand is initially unknown. The Pisces fair-queuing scheduler (Section 4.3.4) uses this approach since it does not know the actual demand for network IO until a socket operation returns. Virtual time in EEVDF advances in proportion to positive service lag whenever a user quiesces and is no longer backlogged in the real system. Users with negative lag are virtually delayed by a dummy request to bring their rate of consumption in line with the virtual GPS round. Since virtual time updates on events in the real system, not in the GPS model, computation overhead is low compared to WFQ. The Linux CFS ([46]) scheduler is similar in construct and operation as EEVDF.

2.2.3 Quanta-based Approximations

Round robin schedulers present a simpler and more efficient alternative to virtual time emulation of GPS. For high-throughput resources that require low overhead, round robin schedulers are the most viable option especially if the user count is high. Like the bit-by-bit GPS network scheduler, a round robin scheduler simply iterates across the current set of active (backlogged) users, executing a number of queued tasks in proportion to each user's weight (one task for the minimal weight user). While efficient — list iteration takes $O(1)$ time to select the next user task versus $O(\log Q)$ for VT schedulers — the basic weighted round robin (WRR) algorithm suffers from severe unfairness since it does not take task size into account. To rectify this problem, deficit round robin [62] (DRR) introduces the notion of per-user resource deficits. In each scheduling round t, DRR refills user i's allocation of resource quanta r_i in proportion to her weighted max-min fair share

$$q_i^t = q_i^{t-1} + r_i \tag{2.24}$$

where q^{t-1} is the deficit quanta remaining from the previous round. As the scheduler scans across the user work queues, it dequeues and services a task p_i^n for user i if the task cost h fits within the user's current deficit q_i^t: $h(p_i^n) \leq q_i^t$. DRR decrements q_i^t accordingly on task completion. If p_i^n is too expensive to execute in the current round, DRR saves the remaining deficit quanta q_i^t for the next round and continues on to the next user. Otherwise, if a user runs out of work, DRR resets his deficit quanta to zero to avoid accruing excess resources for the idle user.

In DRR, round duration contracts and dilates depending on the number of concurrent backlogged users, just as in the GPS model. The total service time $s(t)$ that DRR can render in each round depends on the active users and their current deficits: $s(t) = \sum_{i \in B(t)} q_i^t$. When fewer users are active (smaller $B(t)$), total service time decreases resulting in a shorter scheduling round. If all users are active and consume their entire quanta allocation in a given round, then the round duration equals the max (canonical) round length $s_{\max}(t) = \sum_{i \in Q} q_i^t$. Round duration also adjusts to the current resource capacity. If the resource capacity (e.g., bandwidth) shrinks, rounds take longer to complete and vice versa for increased capacity. Thus, DRR can allocate and consume resources on behalf of user tasks according to their max-min fair shares of *available* resource capacity. DRR provides bounded positive lag ($3\ l_{\max}$) and $O(n)$ negative lag. However, because DRR services each user to completion before moving on to the next in each round, negative lag (and hence delay) also scales with the max/min share ratio $\frac{r_{\max}}{r_{\min}}$.

To address this increased delay, the virtual time round robin [48] (VTRR) scheduler interleaves user tasks within a round according to their resource shares to reduce lag. It uses three key techniques to achieve this goal (i) user queues are ordered by descending weight (ii) each user is serviced for a single quanta at a time rather than to completion (iii) a user task can run only if it does not violate the user's service time share at the current virtual time. Like DRR, VTRR gives each user i a max-min fair quanta allocation $q_i^t = r_i$ in each round t. Starting from the highest weight user i, VTTR executes a task p_i^n for a resource quantum and decrements q_i^t by one. Note that this only works for divisible tasks or properly sized quantum $q > l_{\max}$. Additionally, VTTR advances both the user's virtual finish time $F_i^n = F_i^{n-1} + \frac{q}{r_i}$ and the resource virtual time $v^n = v^{n-1} + \frac{q}{\sum_i r_i}$ according to the GPS model. VTTR then proceeds to the next user j, executing a task if $q_j^t > q_i^t$ or the user's virtual finish time is less than the post-execution virtual time $F_j^{n+1} < v^{n+1}$. Otherwise, the scheduler resumes execution at the head of the user list and repeats the process until either all user quanta has been exhausted or all work has completed. In this way, VTRR retains the efficiency advantage of round-robin scheduling by selecting the next task sequentially and updating queue order only when user weights change, while improving fairness accuracy by emulating GPS more closely.

The last representative scheduler we examine is distributed weighted round robin [38] (DWRR). DWRR extends the basic DRR algorithm over multiple cores for more efficient parallel scheduling. To accomplish this, DWRR decouples and distributes the round scheduling over the cores. Each core k runs a DRR scheduler with a local round counter t_k to service user tasks. Since there may be a per-user load imbalance across the schedulers, at the end of a local scheduling round, each scheduler attempts to steal and execute eligible tasks, i.e., tasks for users with remaining quanta, from the other cores. If no eligible tasks remain, the scheduler increments its local scheduling round t_k and updates the *global* round counter $T = \max_k t_k$. This global round counter synchronizes the scheduling and ensures that each user receives its proper service time in each global round since the local scheduler only renews the user quanta allocation if the local and global rounds match. Since DWRR's synchronization steps only occur at the end of a round, the overhead is much lower than for other VT and WRR schedulers that require locking a centralized queue on every task execution. Moreover, both work stealing and global counter update can be implemented in a non-blocking, atomic fashion without expensive locking. Despite the loose round synchronization, DWRR achieves bounded positive ($2w_{\max}q$) and negative ($-3w_{\max}q$) lag if the maximum user weight $w_{\max}$ is bounded (e.g., 100), which is a reasonable assumption.

Enforcing local per-tenant resource allocations is essential for Pisces to achieve system-wide tenant resource shares. Pisces relies on the local storage node resource scheduler to provide these designated max-min weighted fair shares. Since Pisces operates in a high-throughput storage environment, the resource scheduler is based on DWRR for maximum efficiency with extensions to handle multi-tenancy (Chapter 4), resource vectors, and complex resource cost models (Chapter 6).

Chapter 3

Multi-tenant Shared Cloud Storage

In this thesis we consider multi-tenant cloud services with partitioned workloads (i.e., data sets), where each partition is disjoint but may be replicated on different service nodes. Client requests are routed to the appropriate node based on the partition mapping and replica selection policy in use. Ultimately, request arbitration for resource allocation and performance isolation between tenants occurs at the service nodes.

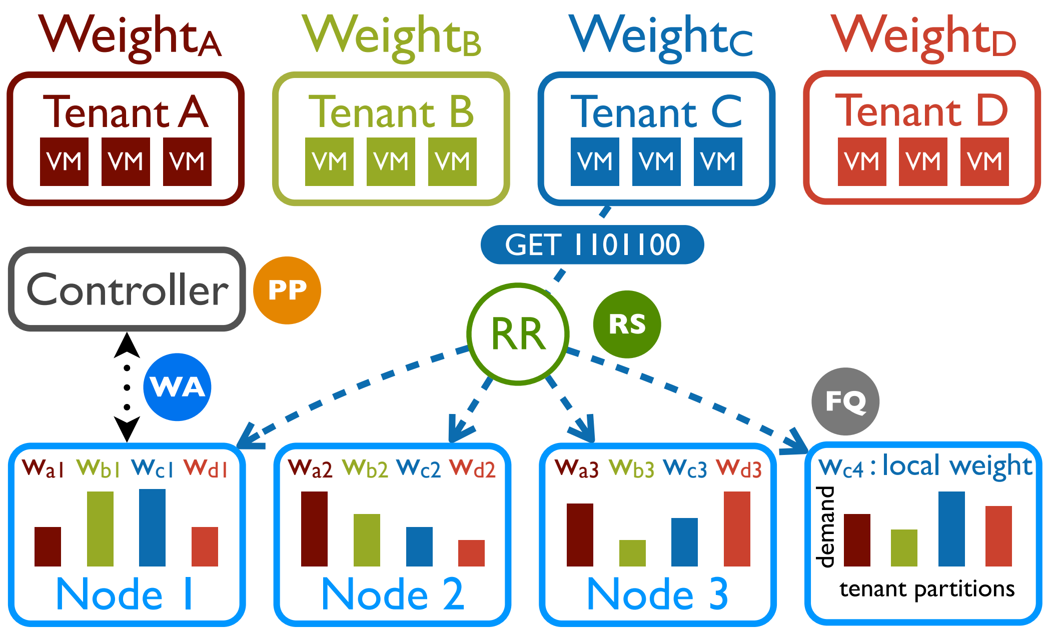


Figure 3.1: Pisces multi-tenant storage architecture.

Figure 3.1 shows the high-level architecture of Pisces, a key-value storage service that provides system-wide, per-tenant fairness and isolation. Pisces provides the semantics of a persistent map between opaque keys (bit-strings) and unstructured data values (binary blobs) and supports simple key lookups (GET), modifications (PUT), and removals (DEL). To partition the workload, the keys are first hashed into a fixed-size key space, which is then subdivided into disjoint segments. While key-value storage systems can support additional data operations (e.g., scans) and structured key spaces (e.g., lexical ordering), we focus on the most common operations and key distribution scheme.

Pisces enforces per-tenant fairness at the system-wide level. As shown in Figure 3.1, each tenant t is given a single, global weight w_t that determines its share of overall system resources (i.e., application throughput). These weights are generally set according to the tenant's service-level objective (SLO). To support service models with rate guarantees, the provider can convert a specified request rate into a corresponding resource proportion (weight) given the current system capacity. If system capacity is fixed or lower bounded, then the proportional shares given by Pisces will achieve the desired rate allocation. The service provider can also adjust the tenant weights as needed, e.g., in response to new tenant requirements or changes in system capacity.

Pisces allows a cloud service provider to offer a flexible service model, in which customers pay for their consumed storage capacity, with an optional additional tiered charge for an "assured rate" service. Assured service users can reserve a minimum service throughput—which, when normalized, translates to a minimum fair share of the global service throughput—with the price dependent upon this rate. The system ensures this minimum rate, while allowing users to exploit unused capacity potentially for an additional "overage" fee. Such multi-tiered charging is common in many Internet contexts, e.g., network transit and CDNs often use burstable billing and charge differently for rate commitments and overages. This is in contrast to maximum rate reservations as found in

Amazon's DynamoDB which charges for an assured rate but disallows over consumption at the cost of lower utilization and forfeited revenue.

3.1 System Assumptions

While Pisces's general mechanisms should extend to other shared storage systems, our current prototype makes some simplifying assumptions. It does not support more advanced queries, such as scans over keys. It is designed primarily to serve keys out of an in-memory cache for high throughput, and only asynchronously writes data to disk (much like Masstree [39] and the MyISAM storage engine in MySQL). For tenants that require durability guarantees we address the need for synchronous persistence in Libra (Chapter 6). We do not focus on consistency issues, and assume that a separate protocol keeps the partition replicas in sync.[1] Further, we assume a well-provisioned network (e.g., one with full bisection bandwidth [5]); we do not deal explicitly with in-network resource sharing and contention, which has been considered by complementary work [58, 60, 56]. Finally, we assume a reasonably stable tenant workload distribution. While Pisces can support highly-skewed demand distributions across partitions and can handle short-term fluctuations in demand for particular keys, the relative popularity of entire partitions should shift relatively slowly (e.g., on the order of minutes). This provides the system with sufficient time to rebalance partition weights when needed.

3.2 Multi-tenancy Model

As shown in Figure 3.2, storage services can adopt a variety of architectures to support multi-tenancy. These models range from no-sharing between dedicated per-tenant servers (Cluster-Per-Tenant) to full-sharing where data isolation exists only in-memory on a per-object basis (Object-Per-Tenant). Given the design goals and system architecture, the primary model of multi-tenancy we support in Pisces is Keyspace-Per-Tenant. In this model,

[1]Our implementation is built on Membase (Couchbase Server) [20], which asynchronously replicates from a partition's primary copy to its backup(s), and by default reads only from the primary for strong consistency.

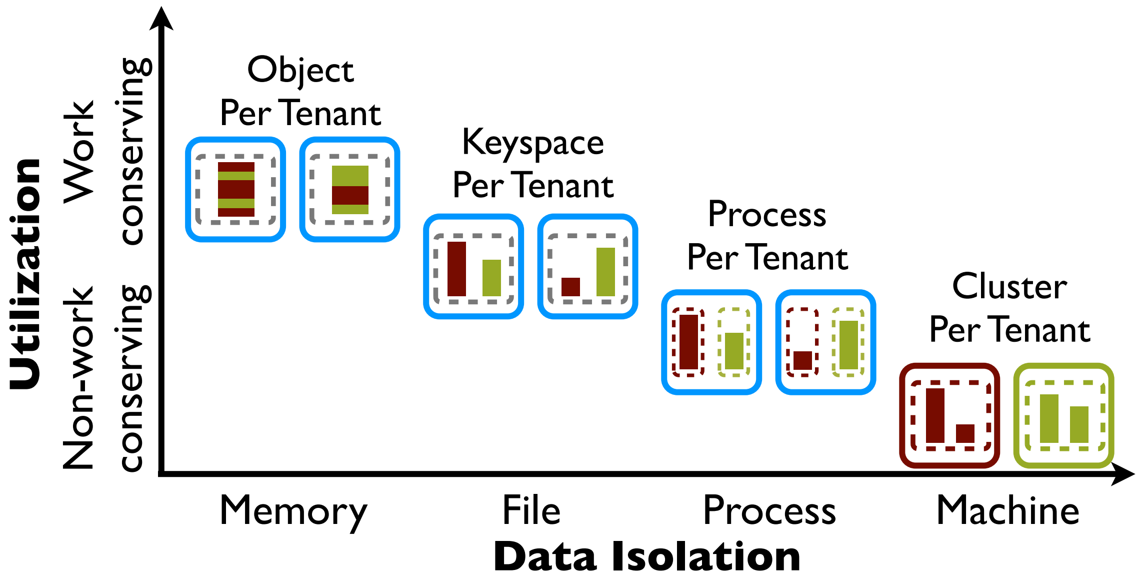


Figure 3.2: Multi-tenancy models in a distributed storage service.

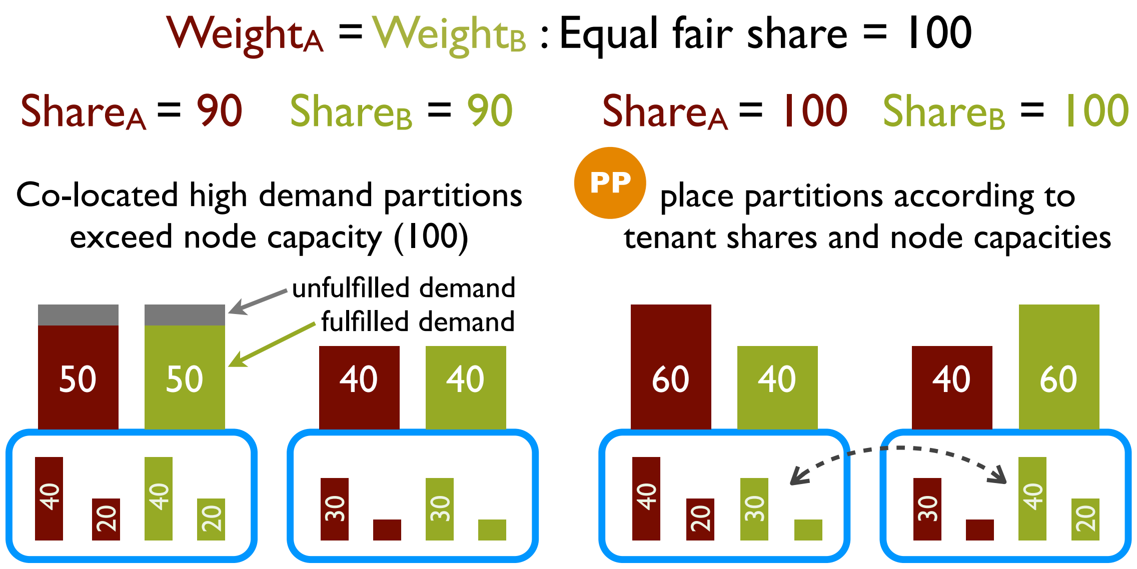
partitions belonging to different tenants can be co-located on the same storage node but with independent key spaces. Multiple threads within a single storage service process requests on behalf of any locally resident tenants. Tenant data, however, is persisted in separate on-disk locations to reduce the chance of inter-tenant data corruption and information leakage. The server process itself is responsible for in-memory data isolation. Although this potentially leaves multiple tenants' data at risk of compromise in the event of a security breach, we expect the storage servers to be dedicated (i.e., not virtualized), hardened machines.

Compared to other models, Keyspace-Per-Tenant trades a bit of data isolation for better performance and throughput. Cluster-Per-Tenant sacrifices too much multiplexed performance by running tenants on their own dedicated machines. Process-Per-Tenant co-mingles tenant partitions on the same set of servers, but retains strong isolation by serving each tenant in their own storage process. However, for high throughput services, each process has to maintain a full set of processing threads, exacerbating the overhead of context switching (e.g., page table and TLB flush). Object-Per-Tenant, on the other hand, errs on the side of minimal isolation by writing data from different tenants into the same on-disk locations (e.g., shared write-ahead logs and even shared data files). This can lead to inter-tenant data corruption and data leakage via the filesystem.

3.3 Life of a Pisces Request

Before a tenant can read (GET) and write (PUT) data in the system, a central controller first performs *partition placement (PP)* to assign its data partitions (and their replicas) to service nodes. The controller then disseminates the partition mapping information to each of the request routers. These request routers can be implemented in client libraries running on the tenants' (virtual) machines, or deployed on intermediate proxies (as illustrated in Figure 3.1). The controller also subdivides each tenant's global fair-share into resource shares at individual storage nodes through *weight allocation (WA)*. When a client tenant, C, issues a request to Pisces, the request router dispatches the request based on its key (e.g., 1101100) and partition location to an appropriate server. If enabled, *replica selection (RS)* allows the request router to flexibly choose which replica to use (e.g., Nodes 1 or 4). Otherwise, the router directs the request to the primary partition. Once the replica has been selected, the router adds the request to a windowed queue of outstanding operations—one queue per server, per tenant.

Since partitions from multiple tenants may reside on this node, the tenants' requests will contend for local resources. To enforce fairness and isolation, the service node applies *fair queuing (FQ)* to schedule tenant requests according to each tenant's local weight ($w_{c,4}$). When a request reaches the server, the server adds the request to a queue specific to that tenant (C). On every scheduling "round", the server allocates resource quanta to each tenant according to its local weight ($w_{c,4}$), which it then consumes when processing requests from the tenant queues. As in DRR, described in Section 2.2.3, if the request consumes more than the allocated resources, it must complete on a subsequent round after tenant C's quanta have been refilled. This guarantees that each tenant will receive its local fair share in a



WeightA = WeightB : Equal fair share = 100
ShareA = 90
ShareB = 90
ShareA = 100
ShareB = 100
Co-located high demand partitions exceed node capacity (100)
PP place partitions according to tenant shares and node capacities
unfulfilled demand
fulfilled demand
50
50
40
40
60
40
40
60
40
20
40
20
30
30
40
20
30
30
40
20

Figure 3.3: Random partition placement can lead to infeasibility. Left-hand figures correspond to settings lacking Pisces's mechanisms; right-hand figures apply its techniques.

given round of work, if multiple tenants are active. Otherwise, tenants can consume excess resources left idle by the others without penalty.

3.4 System-wide Fair Sharing: Example

The challenge of achieving system-wide fairness can be illustrated with a few scenarios. From these, we derive design lessons for how Pisces should (1) place partitions to enable a fair allocation of resources, (2) allocate local weights to maintain global fairness, (3) select replicas to achieve high utilization, and (4) queue requests to enforce fairness. In the examples, two tenants (*A* and *B*) with equal global shares, where shares refer to there normalized resource allocations, access two Pisces nodes with equal capacity (100 kreq/s). Each tenant should receive the same aggregate share of 100 kreq/s.

3.4.1 Place Partitions With Respect To Node Capacity Constraints

For per-tenant fairness to be *feasible*, there must exist some assignment of tenant partitions that can satisfy the global tenant shares without violating node capacities. Not all placements lead to a feasible solution, however, as shown in Figure 3.3. Here, each tenant has

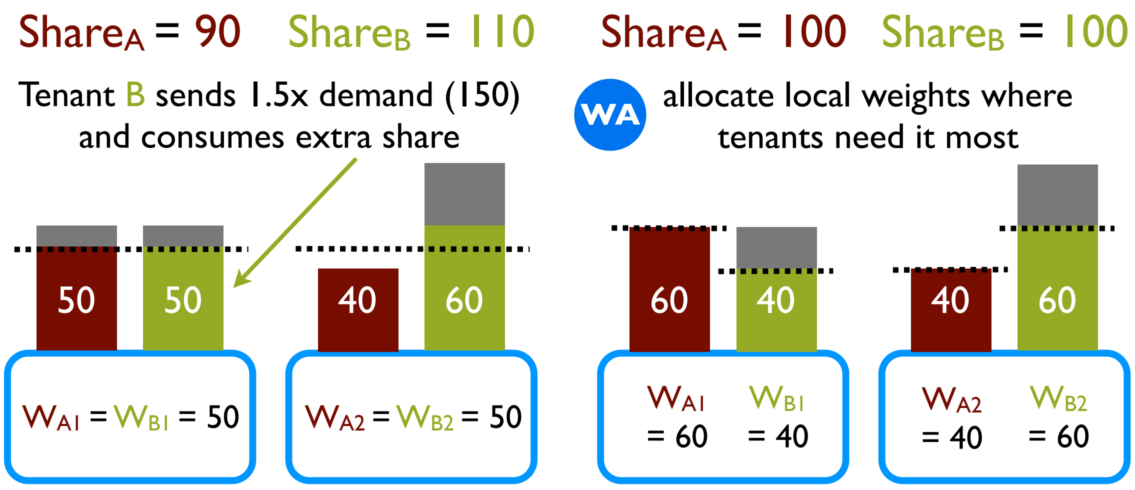


Figure 3.4: Mirrored local weight allocation can impair fairness and performance.

the same skewed distribution of partition demand: 40, 30, 20, and 10. Arbitrary partition assignment can easily lead to capacity overflow: with *A* and *B* both demanding 60 kreq/s for the partitions on the first node, each tenant receives 10 kreq/s less than their global share. If we take partition demand into account and shuffle tenant *B*'s partitions between the nodes, then we can achieve a fair and feasible placement. Although the skew in this example may be extreme, Internet workloads often exhibit a power-law (or Zipf) distribution across keys, which can induce this type of skewed partition demand.

3.4.2 Allocate Throughput Where Tenants Need It Most

Even with a feasible fair partition placement, if the local weights simply mirror the global (uniform: 50 each) weights, as in Figure 3.4, global fairness may still suffer. Fair queuing allows tenant *B* to consume more than its local share ($60 > 50$) at the first node when *A* consumes less ($40 < 50$) despite the skewed partition demand. However if tenant *A* increases its overall demand, it will be able to consume its remaining local allocation at node 1. This will increase its global share and eat into tenant *B*'s global share. However, if we adjust each tenant's local weights to match their demand (60/40 and 40/60), we can preserve fairness even under excess load.

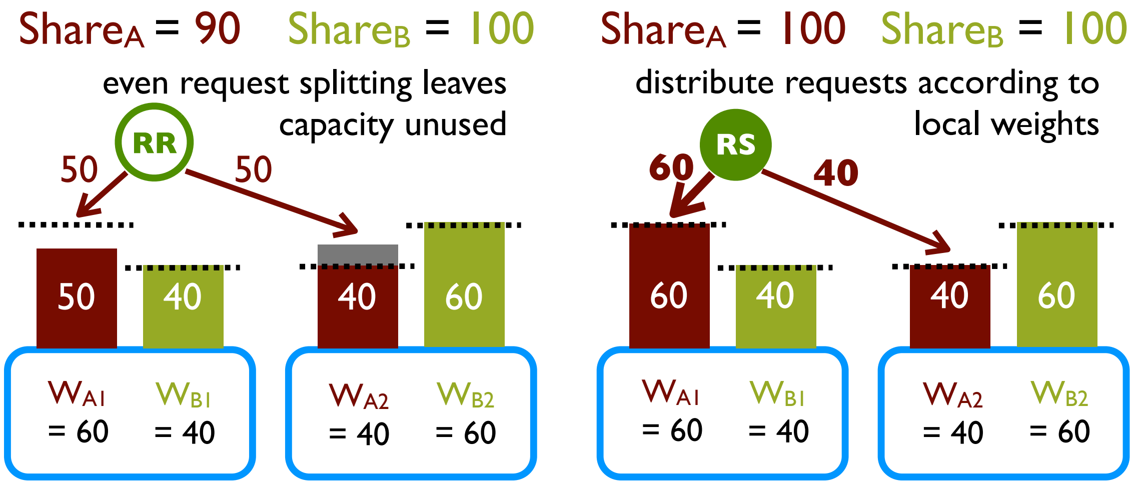


Figure 3.5: Equal split replica selection can degrade fairness and performance.

3.4.3 Select Replicas In A Weight-sensitive Manner

When replica selection is enabled, the fairness problem becomes easier in general, since each replica only receives a *fraction* of the original demand. By spreading partition demand across multiple servers, replica selection produces smoother distributions that makes partitions easier to place. Further, once placed, the reduced per-replica demand is easier to match with local weights. However, the replica selection policy must be carefully tuned, otherwise fairness and utilization may still diverge from the system-wide goals. In Figure 3.5, tenant *A* can send requests to replicas on either node. Since its local weights are skewed to match the partition demand, simple replica-selection policies are insufficient to exploit the variation between the provisioned weights. For example, equal-split round robin would lead to a 10 kreq/s drop in *A*'s global share. Instead, by adjusting per-tenant replica selection proportions according to the local weights, we can fully exploit replicated reads to improve performance and facilitate fairness.

3.4.4 Enforce Dominant Resource Fairness At The Local Node

Up to this point, our examples have illustrated multiple tenants with identical weights competing for identical request rates, which implicitly assumes that all requests have equivalent cost. In practice, requests may be of different input or output sizes, and can activate differ-

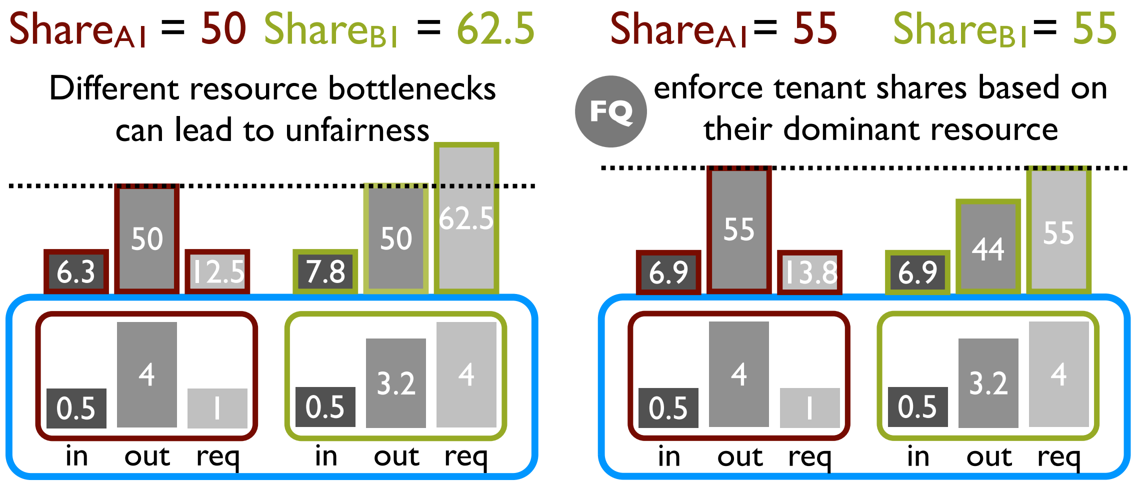


Figure 3.6: Single-resource queuing can violate fairness.

ent bottlenecks in the system (e.g., small requests may be bottlenecked by server interrupts, while large requests may be bottlenecked by network bandwidth). Thus, each tenant's workload may vary accordingly across the different resources, as seen in Figure 3.6.

Each tenant's resource profile shows the relative proportion of each resource—bytes in, bytes out, and number of requests—that the tenant consumes. These resources are the likely limiting factor for large writes, large reads, and small requests (read or writes), respectively. Tenant *A* is read bandwidth bound, consuming 4% of the out bandwidth for every 1% of request capacity it uses. Tenant *B*, on the other hand, is interrupt-bound for its smaller reads, consuming more request resources (4%) than out bandwidth (3.2%). Applying fair queuing to a single resource (bytes out) gives each tenant a fair share of 50%, but also allows tenant *B* to receive a larger share (62.5%) of its dominant resource (requests). Instead, using *dominant resource fairness* (DRF) [25] ensures that each tenant will receive a fair share of its dominant resource relative to other tenants: tenant *A* receives 55% of out bytes and tenant *B* receives 55% of requests, while out bytes remains the bottleneck.

Chapter 4

Pisces

Pisces makes a number of contributions to address the challenges of providing system-wide per-tenant allocations of application-level resources.

- **Global fairness:** To our knowledge, Pisces is the first system to provide per-tenant fair resource sharing across all service instances. Further, as total system capacity is allocated to tenants based on their normalized weights, such a max-min fair system can also provide minimal throughput guarantees given sufficient provisioning. In comparison, recent commercial systems that offer request rate guarantees (i.e., Amazon DynamoDB [1]) do not provide fairness, assume uniform load distributions across tenant partitions, and are not work conserving.

- **Novel mechanism decomposition:** Pisces introduces a clean decomposition of the global fairness problem into four mechanisms. Operating on different timescales and with different levels of system-wide visibility, these mechanisms complement one another to ensure fairness under resource contention and variable demand.

  (i) Partition Placement ensures a feasible fair allocation by centrally assigning (or remapping) tenant partitions to nodes according to per-partition demand while respecting per-node resource constraints (long timescale).

(ii) Weight Allocation distributes overall tenant fair shares across the system where most needed, i.e., skewed to the popular partitions, by adjusting local per-tenant weights at each node (medium timescale).

(iii) Replica Selection improves both fairness and performance by directing tenant requests to service nodes in a local weight-sensitive manner (real-time).

(iv) Weighted Fair Queuing at service nodes enforces performance isolation and fairness according to the local tenant weights and workload characteristics (real-time).

- **Novel algorithms:** We introduce several novel algorithms to implement Pisces's mechanisms. These include a reciprocal swapping algorithm for weight allocation that shifts weights when tenant demand for local resources exceed their local share, while maintaining global fairness. We use a novel application of optimization-inspired congestion control for replica selection, which complements weight allocation by distributing load over partition replicas in response to per-node latencies. Finally, to manage different resource bottlenecks on contended nodes, we enforce *dominant resource fairness* [25] between tenants at the node level, while providing max-min fairness at the service level. To do so, we extend traditional deficit-weighted round robin queuing to handle per-tenant multi-resource scheduling.
- **Low overhead and high utilization:** Pisces is designed to support high server utilization, both high request rates (100,000s requests per second, per server) and full bandwidth usage (Gbps per server). To do so, its mechanisms must apply at real time without inducing significant throughput degradation. Through careful system design, our prototype achieves $<3\%$ overhead for 1KB requests and actually outperforms the unmodified, non-fair version for small requests. Commercial systems like DynamoDB, on the other hand, typically target lower rates (e.g., more than 10,000 reqs/s requires special arrangements [2]).

System Parameters	
w^t	global weight for tenant t
z^t	global max-min weighted fair share for t: $z^t = \frac{w^t \sum_n c_n}{\sum_u w^u}$
f_p^t	t's fair share demand for partition p: $\sum_p f_p^t = z^t$
ρ^t	replicas per partition for tenant t
c_n	resource capacity for service node n
P^t	set of partitions for tenant t
T, N	set of all tenants and service nodes
Decision Variables	
r_n^t	local resource share for t at n
w_n^t	local weight t at n: $w_n^t = \frac{r_n^t}{\sum_u r_n^u}$
Q^t	$\lvert N\rvert \times \lvert P^t\rvert$ partition replica selection matrix

Table 4.1: Pisces system model

Global Fair-Sharing Optimization

$$\begin{aligned}
\textbf{maximize: } & \Lambda(r_{n\in N}^{t\in T}, Q^{t\in T}) : \text{throughput utility function} && (4.1)\\
\textbf{subject to: } & \sum_n r_n^t \le z^t : \text{fair share constraint} && (4.2)\\
& \sum_t r_n^t \le c_n : \text{node capacity constraint} && (4.3)\\
& \sum_p Q_{n,p}^t \le r_n^t : \text{local share constraint} && (4.4)\\
& \sum_n Q_{n,p}^t > 0 = \rho^t : \text{replication constraint} && (4.5)\\
& \sum_n Q_{n,p}^t \le f_p^t : \text{partition demand constraint} && (4.6)
\end{aligned}$$

Table 4.2: Pisces global fair-sharing optimization problem

4.1 Pisces System Model

To motivate the design of Pisces' mechanisms and show how they work in tandem to provides per-tenant fairness at the system-wide level, we model the system as a global optimization problem, shown in Table 4.1. As described in Section 2.1, the optimization model not only provides a framework for understanding and improving system-wide performance, but it also sheds light on viable architectural decompositions that turn the (heuristic) design lessons described in Section 3.4 into well-formulated (optimal) mechanisms. At a high level, each tenant t has a single, global weight w^t that determines its fair share of aggregate system resources (i.e., throughput): z^t. As mentioned, these weights are generally set according to the tenant's service-level objective and can be adjusted at any time.

To ensure each tenant receives its fair share, Pisces has to make several key decisions. Since tenant partitions p are distributed across the service nodes n and can have (varying) demand f_p^t, Pisces needs to determine (i) where each partition's ρ^t replicas should reside (*partition placement (PP)*), (ii) how much demand to send to each one (*replica selection (RS)*), and (iii) what share of local resources to give each tenant at the nodes hosting its partitions (*weight allocation (WA)*). Lastly, to enforce fairness and provide performance isolation, service nodes should schedule requests with some form of *fair queuing (FQ)*. Note that in Pisces, the aggregate resource shares are defined in terms of the backlogged max-min fair share described in Section 2.2. While Pisces allows tenants to consume any unused capacity across the system, it only guarantees an *exact* proportional share of the excess according to the local tenants weights at individual storage nodes, not system-wide.

Although the global objective could explicitly target per-tenant fairness, the real goal is to find a fair allocation that also achieves high performance. In other words, within the set of feasible solutions defined by the constraints where each tenant receives its fair share (4.2) across the system, no node is over-burdened (4.3), all local allocations are enforced (4.4), and per-partition tenant demand is satisfied (4.6), we want to find the partition mapping, i.e., where $I(Q^t) > 0$, local resource allocation r_n^t, and replica selection policy Q^t that maximizes overall throughput (4.1). While this global formulation gives a bird's-eye view of the system objective and fairness constraints, it serves solely as an orienting framework. Gathering the necessary measurements and updating the appropriate components (request routers and service nodes) to solve the global problem in a centralized fashion at sufficient frequency to adapt to dynamic workloads would be prohibitive. Instead, PP, WA, and RS

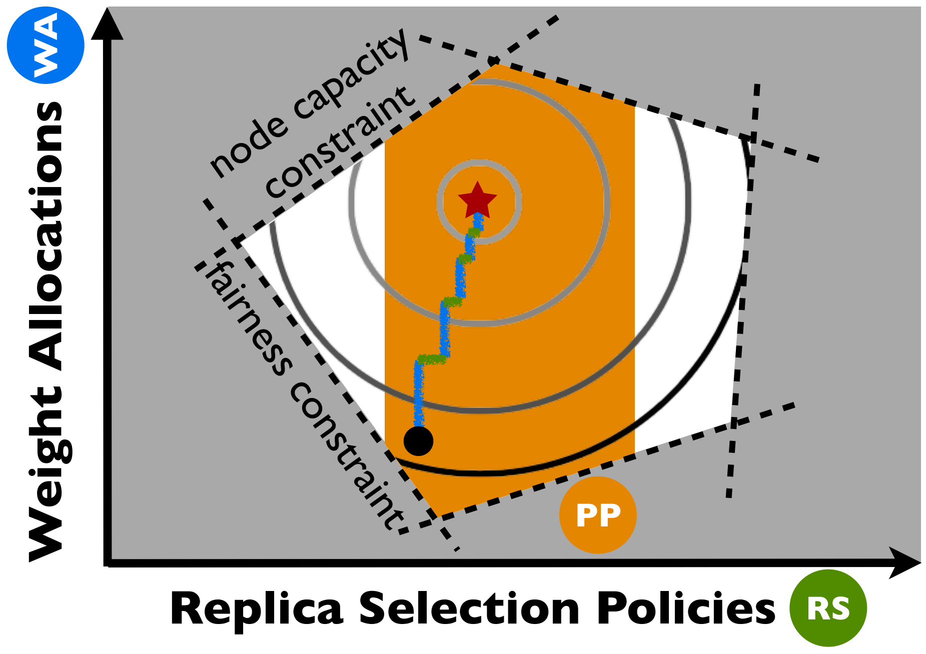


Figure 4.1: Partition Placement (PP) ensures *feasibility* by fitting each tenant's per-partition fair-share demand within the node capacity constraints (shaded region within the constraint set). Weight Allocation (WA) and Replica Selection (RS) then iteratively search for the optimal solution (star) starting from an initial configuration (dot) via coordinate gradient ascent, climbing through regions of progressively higher throughput (isoclines).

(FQ simply enforces the shares determined by WA) should compute their own policies in a dynamic and decoupled fashion which we describe in the next section.

4.2 Pisces Mechanisms

We use optimization decomposition (Section 2.1), to break the global problem down into the more tractable and adaptive pieces shown in Table 4.3. In this section we discuss the sub-problem each mechanism solves and how they fit together to achieve "optimal" global fairness, illustrated visually in Figure 4.1.

4.2.1 Partition Placement

The main goal of partition placement (PP) is to find a *feasible* configuration of Q^t i.e., fixing its non-zero entries which correspond to the assignment of partition p to node n for tenant t. In other words, PP boils down to placing partitions such that each tenant's fair-share partition demand (4.10) fits within the node capacities (4.8). As the top-level master

Partition Placement (long timescale: min/hrs)

$$\textbf{minimize: } \Upsilon(Q^{t\in T}, c_{n\in N}) = \max_n \frac{\sum_{t,p} Q^t_{n,p}}{c_n} \tag{4.7}$$

$$\textbf{subject to: } \sum_{t,p} Q^t_{n,p} \leq c_n : \text{node capacity constraint} \tag{4.8}$$

$$\sum_n Q^t_{n,p} > 0 = \rho_t : \text{replication constraint} \tag{4.9}$$

$$\sum_n Q^t_{n,p} \geq f^t_p : \text{demand constraint} \tag{4.10}$$

Weight Allocation (medium timescale: sec)

$$\textbf{maximize: } \Lambda(r^{t\in T}_{n\in N}, Q^{*t\in T}) = \sum_{t,n} \log Q^{*t}_n \tag{4.11}$$

$$\textbf{subject to: } \sum_n r^t_n \leq z^t : \text{fair share constraint} \tag{4.12}$$

$$\sum_t r^t_n \leq c_n : \text{node capacity constraint} \tag{4.13}$$

$$\textbf{parameters: } Q^{*t}_n \text{ (fixed by RS)}$$

Replica Selection (real-time: ms)

$$\textbf{maximize: } \Lambda(r^{t\in T}_{n\in N}, Q^{*t\in T}) = \sum_{t,n} \log Q^{*t}_n$$

$$\textbf{subject to: } Q^t_n \leq r^{*t}_n : \text{local share constraint} \tag{4.14}$$

$$\textbf{parameters: } r^{*t}_n \text{ (fixed by WA)}, I(Q^t) > 0 \text{ (fixed by PP)}$$

Table 4.3: Pisces mechanism decomposition

problem (Section 2.1.2) in the Pisces decomposition, PP ensures that the other mechanisms search within the set of globally fair solutions, as shown in Figure 4.1. In this way, PP factors out the non-convex bin-packing demand constraints, which renders the remaining optimization sub-problems solvable via convex optimization.

Since PP requires global information, it runs on the controller which gathers per-partition demand statistics from each node to determine the fair-share demand f^t_p and optimizes the PP sub-problem shown in Table 4.3. The optimal partition assignment $I(Q^t_{n,p}) > 0$ minimizes the max node utilization (4.7), which in turn optimizes the original objective (4.1) via a primal subgradient allocation to give WA and RS greater headroom for optimizing throughput. Note that the PP sub-problem omits the local shares r^t_n since it only needs to ensure that the total node demand conforms to the capacity constraint (4.8).

4.2.2 Weight Allocation

Given a feasible partition placement, weight allocation divides each tenant's global share into local shares r^t_n where the tenant needs it most, i.e., where the per-node tenant demand determined by Q^t is highest. Although we could optimize both of these variables together as a single slave sub-problem at the controller and disseminate the policies to the appropriate components, the computation and update overhead would limit adaptivity and prevent the system from handling short-term demand variations. Thus, as shown in Table 4.3, we decompose the problem further into a "master" weight allocation (WA) and "slave" replica selection (RS) sub-problems to minimize coordination and achieve near real-time adaptivity. These two mechanisms work in tandem to optimize system throughput by iteratively adapting r^t_n to match tenant demand then adjusting Q^t to leverage the larger local shares, as depicted in Figure 4.1.

Using the primal approach (Section 2.1.2), WA observes per-node tenant request latencies, which acts as a proxy for tenant demand, and increases local shares r^t_n to match tenant demand. On each iteration, WA collects an estimate of the per-node tenant demand $x^t_n = \sum_p Q^{*t}_{n,p}$ generated by RS. Using this estimate, WA approximates the RS Lagrangian w.r.t r^t_n (i.e., throughput subgradient) : $\frac{\partial L(Q,\lambda)}{\partial r^t_n} = \lambda^t_n$, with a latency-based cost function: $l^t_n = 1 \,/\, (r^t_n - x^t_n)$. Minimizing this cost function maximizes throughput (Table 4.3, eq. 4.11) since λ^t_n is a "congestion" price corresponding to the request queuing delay experienced by tenant t at node n. Since WA requires global information (max latency), WA also runs on the controller.

4.2.3 Replica Selection

When enabled, replica selection (RS) not only improves performance by load-balancing read requests, but it also relaxes the fair-share demand constraint (4.10). RS accomplishes this by smoothing the demand distribution across replicas and alleviating node hotspots. This expands the set of feasible solutions since tenant partition demand is now easier to fit within node capacities. Given the local shares r_n^{*t} computed by WA, the RS optimization ensures that the replica-selection policy Q^t sends more demand to replicas on nodes with greater local resource shares. Note that these local shares are in fact, auxiliary primal variables used to decouple the individual tenant RS policies $Q_{n,p}^t$. Without them, one tenant's RS policy would affect the operation, and hence throughput, of another.

In RS, we apply dual decomposition (Section 2.1.2) to minimize coordination overhead by distributing the RS optimization across request routers. Since the local rate allocation r_n^{*t} isolates tenant demand from each other on each node, RS can compute $Q_n^t = \sum_p Q_{n,p}^t$independently for each tenant. This allows each request router to not only compute the tenant's RS policy on a per-node rather than per-node per-partition basis, but also ignore the demand, and hence congestion pressure, generated by other tenants. RS uses a modified FAST-TCP [76] gradient update to optimize the tenant policies for multiple storage nodes and partition replicas.

4.2.4 Fair Queuing

Although not explicitly involved in the optimization computation, fair queuing (FQ) plays the crucial role of implementing and enforcing the fairness and performance bounds established by the local rate allocations. Moreover, these local shares acts as a coordination point between WA and RS, eliminating the need for direct communication. RS implicitly detects the local shares r_n^{*t} through latency estimates, while WA infers the current replica selection policy Q_n^{*t} by measuring the actual per-node request rate x_n^t at n. In this way, FQ implicitly computes a "congestion" latency price that forces RS to back off on over utilized nodes and informs WA to increase the local allocation.

4.3 Pisces Algorithms

Pisces implements four complementary mechanisms according to the decomposition model described in the previous section. Each mechanism operates on different parts of the system at different timescales. Together, they deliver system-wide fair service allocations with minimal interference to each tenant.

4.3.1 Partition Placement

Algorithm 1 Partition Placement: long timescale (m)

```
$T$: tenants, $N$: nodes, $W$: global tenant weights, $P$: partitions, $D$: tenant demand,
$F$: tenant fair-share demand, $M$: partition mapping
function CONTROLLER.PLACEPARTITIONS($T, P, N$)
    for $t \in T$ do
        $D \leftarrow collect\_partition\_rates(N, T, P)$
        $F \leftarrow compute\_fair\_share\_demand(W, D, C)$
    $M \leftarrow solver.bin\_pack\_partitions(D, F, C)$
    $reassign\_partitions(M)$
```

The partition placement mechanism ensures the feasibility of the system-wide fair shares. It assigns partitions so that the load on each node (the aggregate of the tenants' per-partition fair-share demands) does not exceed the node's rate capacity. Since our prototype currently implements a simple greedy placement scheme, we only outline the algorithm here, shown in Algorithm 1, without providing details. The centralized controller first collects the request rate for each tenant partition, as measured at each server. It then computes the partition demand distribution by normalizing the rates. Scaling each tenant's global fair share by this distribution yields the per-partition demand share that each tenant should receive. Next, the controller supplies the demand and node capacity constraints into a bin-packing solver to compute a new partition assignment relative to the existing one. Finally, the controller migrates any newly (re)assigned partitions and updates the mapping tables in its request router(s).

Partition placement typically runs on a long timescale (every few minutes or hours), since we assume that tenant demand distributions (proportions) are relatively stable, though demand intensity (load) can fluctuate more frequently. However, it can also be executed in response to large-scale demand shifts, severe fairness violations, or the addition or removal of tenants or service nodes. While the bin-packing problem is NP-hard, simple greedy heuristics may suffice in many settings, albeit not achieve as high a utilization. After all, to achieve fairness, we only need to find a feasible solution, not necessarily an optimal one. Further, many efficient approximation techniques can find near-optimal solutions [61, 47]. We intend to further explore partition placement in future work.

4.3.2 Weight Allocation

Algorithm 2 Weight Allocation: medium timescale (s)

```
$T$: tenants, $N$: nodes, $W$: global tenant weights, $R$: local tenant resource share
function CONTROLLER.ALLOCATEWEIGHTS($T, N, W, R$)
    $D \leftarrow monitor\_node\_rates(T, N)$
    $C \leftarrow compute\_cost\_estimates(D, R)$
    $R \leftarrow compute\_weight\_swap(\max(C))$
    $reassign\_weight\_allocations(R)$
```

Once the tenant partitions are properly placed, weight allocation iteratively adjusts the local tenant shares (R) to match the demand distributions. As sketched in Algorithm 2, the weight allocation algorithm on the controller (i) detects tenant demand-share mismatch and (ii) decides which tenant share(s) (weights) to adjust and by what amount, even while it (iii) maintains the global fair share. A key insight is that any adjustment to tenant shares requires a *reciprocal swap*: if tenant t takes some rate capacity from a tenant u on some server, t must give the same capacity back to u on a different server. This swap allows the tenants to maintain the same aggregate shares even while local shares change. Each tenant's local weight is initially set to its global weight, w_t, and then adapts over time. To adapt to distribution shifts and still allow replica selection to adjust to the current allocation, weight allocation runs at the medium timescale (seconds).

Detecting mismatch: While it is difficult to directly measure tenant demand, the central controller can monitor the rate each tenant receives at the service nodes. Weight allocation uses this information (D), together with the allocated shares (R), to approximate the congestion-based subgradient of the throughput objective, i.e., demand-share mismatch, that tenants experience at each node. We tried both a latency-based cost function using the M/M/1 queuing model of request latency—i.e., $l_n^t = 1 / (R_n^t - D_n^t)$ for tenant t on node n—as well as a more direct rate-based cost. Unfortunately, in a work-conserving system, it can be difficult to determine how much rate R was actually allocated to t, as it can vary depending on other tenants' demand as shown in Section 2.2. Instead, by using the difference between the measured rate and the configured local share, $e_n^t = \left| D_n^t - \hat{R}_n^t \right|$, we can largely ignore the variable allocation and instead focus on the tenant's desired rate under full load (i.e., $\hat{R}$). Fortunately, the allocation algorithm can easily accommodate any convex cost function to approximate demand mismatch.

Determining swap: Since the primary goal of the Pisces model is to maximize tenant performance, weight allocation seeks to minimize the *maximum* demand-share mismatch (or cost) which in turns optimizes tenant throughput via subgradient update. However, giving additional rate capacity to the tenant t that suffers maximal latency necessarily means taking away capacity from another tenant u at the same node n. If too large, this rate (weight) *swap* may cause u's cost to exceed the original maximum. To ensure a valid rate swap, the algorithm uses the linear bisection for latency, $\text{take}(t, u, n) = ((R_n^u - D_n^u) - (R_n^t - D_n^t)) / 2$, or the min of the differences for rate: $\min(e_n^t, e_n^u)$.

Maintaining fair share: Before committing to a final swap, weight allocation must first find a reciprocal swap to maintain the global fair share: if tenant t takes from tenant u at node n, then it must reciprocate at a different node. Given a reciprocal node m, the controller computes the rate swap as the minimum of the take and give swaps, swap $= \min(\text{take}(t, u, n), \text{give}(u, t, m))$, and translates them into local weight settings. While this bilateral exchange between two tenants may suffice, a multilateral exchange may optimize

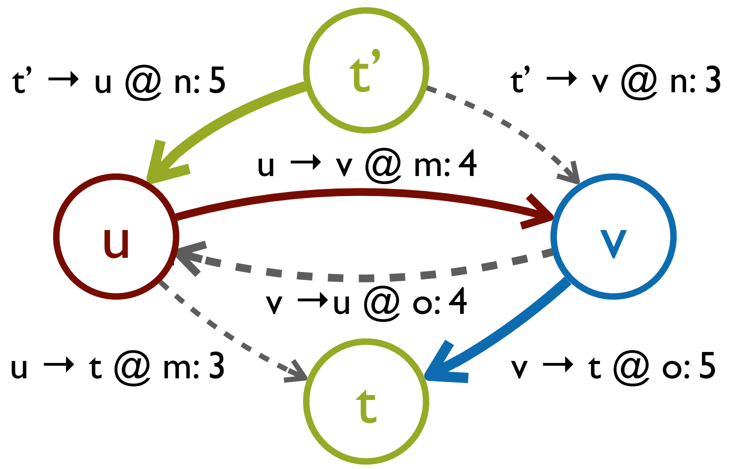


Figure 4.2: Reciprocal weight exchange modeled as a Maximum Bottleneck Flow problem.

local shares even further. We model this exchange as a path through a flow graph, as shown in Figure 4.2. Nodes in the graph represent tenants and each directed edge represents a possible swap with capacity equal to the swap rate (e.g., tenant u can take rate 4 from tenant v at server m). Swap rates are computed as the (positive) maximum over all server nodes where the edge's tenants have co-located partitions.

The max latency tenant t is modeled as both the source t' and sink of the flow graph, as it must first take rate to minimize its cost (latency or rate-distance) and ultimately reciprocate it to maintain the global fair-share invariant. In the example, both bilateral exchanges ($t' \rightarrow u \rightarrow t$ and $t' \rightarrow v \rightarrow t$) are bottlenecked by the smallest capacity edge (i.e., 3). Instead, weight allocation should choose the multi-hop path ($t' \rightarrow u \rightarrow v \rightarrow t$) with bottleneck 4 that corresponds to the Maximum Bottleneck Flow (MBF). The MBF not only minimizes the max cost for t by the greatest extent, but also reduces the cost for u and v. In general, the longer the swap path, the greater the number of tenants who benefit.

On each iteration of the weight allocation loop in Algorithm 2, the controller constructs the flow graph from the collected node latency data and solves the MBF problem using a variant of Dijkstra's shortest path algorithm. Then, just as with bilateral swaps, the algorithm converts the rates into weights and sets the local tenant shares accordingly. Since the WA-RS decomposition is an iterative optimization algorithm, we rely on the general properties of convex optimization to ensure convergence and stability. The latency cost function exhibits convexity over the operating regime where $r_n^t > x_n^t$, and each gradient descent step (reciprocal exchange) shrinks the latency variation across the max latency tenant t's nodes, which reduces the next weight swap (step size) involving t. The timescale separation between WA (seconds) and RS (real-time) allows the RS sub-problem to converge to an optimal Q^{*t}, i.e., request window lengths, within each WA iteration. Taken together, these properties ensure that WA and RS will converge to an optimal fair-share weight allocation [15] (< 20 iterations in our experiments). To avoid oscillations around the optimal point, only swaps that exceed a minimal threshold ϵ are executed.

4.3.3 Replica Selection

Algorithm 3 Replica Selection: real-time (ms)

```
  $j$: request/response, $t$: tenant, $M$: partition mappings
  $n$: node, $q^t$: per-tenant request queue
  function RequestRouter.SendRequest($j, t, M$)
      $p \leftarrow get\_partition\_of(j)$
      for $n \in M[t, p]$ do
          if window $w_n^t >$ outstanding $s_n^t$ then
              $send\_request(j, n)$
              $s_n^t \leftarrow s_n^t + 1$ return
      if not sent then $queue\_request(j, q^t)$
  function RequestRouter.RecvResponse($j, t, n$)
      $l_{\text{resp},n} \leftarrow latency\_of\_response(j)$
      $s_n^t \leftarrow s_n^t - 1$
      $update\_window(w_n^t, l_{\text{resp},n})$
      SendRequest($dequeue\_request(q^t), t, M$)
```

To maintain fairness while balancing load for optimal throughput, request routers distribute tenant requests to partition replicas in proportion to their local shares (i.e., normalized weights). While the controller (or alternatively, each server) could disseminate the local share information to each request router, Pisces avoids the need for explicit updates by exploiting implicit feedback. Explicit updates could be prohibitively expensive for a system with tens of thousands of tenants, request routers, and service nodes.

As delineated in Algorithm 3, when a request router (or client) sends a request, it round-robins between partition replicas (nodes), consuming slots from their respective request windows. Once the windows fill up, the request router locally queues requests until server responses free additional window slots. Due to per-tenant fair queuing at the nodes, requests sent to nodes with a larger tenant share experience lower queuing delay than nodes with a smaller share. The request router uses the relative response latency between replicas as a proxy for the dual "congestion" price imposed by the storage node in response to excessive demand. Thus, the request router adjusts its per-node request windows according to the FAST-TCP [76] update:

$$w(m+1)_n^t \;=\; (1-\alpha)\cdot w(m)_n^t \;+\; \alpha \cdot \left(\frac{l_{\text{base}}}{l_{\text{est}}}\right)$$

Each iteration of the algorithm adjusts the window based on how close the tenant demand is to its local rate allocation, which is represented by the ratio of a desired average request latency, l_{base}, to the smoothed (EWMA) latency estimate, l_{est}. The α parameter limits the window step size. In this way, each request router makes proper adjustments in a fully decentralized fashion: it only uses local request latency measurements to compute the replica proportions. The convergence and stability guarantees for the replica selection follow directly from those given by FAST-TCP.

4.3.4 Fair Queuing

Ultimately, system-wide fairness and isolation comes down to mediating resource contention between tenants at the individual storage nodes. As stated in Section 2.2.3, Pisces implements local fair queuing using a form of distributed weighted round robin [38] (DWRR). We chose to extend this quanta-based scheduling discipline because of its simplicity, low time complexity, multi-core efficiency, and bounded deviation from the ideal max-min fair Generalized Processor Sharing model for finite (< 1000 in Pisces) weights. Pisces adds support for multi-tenancy to enforce the per-tenant allocations r_n^t and resource vectors to allocate and track resource consumption.

Algorithm 4 Fair Queuing: real-time (μs)

```
$j$: request, $t$: tenant, $r$: round, $g$: global round, $R$: local tenant share
Executes in each scheduler thread $s \in S$

function ServiceNode.QueueRequests($R$)
    $r \leftarrow 0, g \leftarrow 0$ (set initial round to 0)
    while request_queue_not_empty() do
        while $j, t \leftarrow$ dequeue_request() do
            if $t$.state = inactive and $r = g$ then
                $t$.state $\leftarrow$ active
                allocate_tenant_quanta($t, R$)
            if $t$ has quanta then
                consume_request_resources($j$, $t$)
                if $j$ unfinished then
                    queue_request($j$)
                else if $j$ has leftover resources then
                    refund_unused_resources($j$,$t$)
            else
                queue_exhausted_request($j, t$)
                $t$.state $\leftarrow$ exhausted
        steal_work_and_quanta($S$)
    if $\exists\, t \in T$ such that $t$.state = exhausted then
        $r \leftarrow r + 1$ (increment round)
        if $r > g$ then
            $g \leftarrow r$ (advance global round)
            for $t \in T$ do
                if $t$.state = exhausted then
                    $t$.state $\leftarrow$ active
                    allocate_tenant_quanta($t, R$)
                else if $t$.state = active then
                    $t$.state $\leftarrow$ inactive (clears unused quanta)
```

The basic unit of work in the Pisces scheduler is called a quantum, which represents the resources required to handle a single normalized (i.e., 1 KB GET or PUT) request. However, unlike other quanta based *single* resource schedulers, Pisces schedules over *multiple* resources using dominant resource fair queuing (DFQ), as mentioned in Section 3.4. DFQ ensures that each tenant receives a proportional (max-min) share of the resource it consumes most, e.g., bytes in for PUT-heavy workloads, bytes out for GET-heavy workloads, and requests for small interrupt-bound requests. If all tenants have the same dominant resource, then DFQ degenerates to single resource max-min fair queuing. Thus, each quantum corresponds to a vector of tenant resources.

To translate a quantum into a tenant-specific resource vector, DFQ tracks each tenant's resource consumption profile k^t over all requests and periodically (every half second in our prototype) recomputes the per-tenant dominant resource fair (DRF) shares r_i^t

$$u_i^t = \frac{k_i^t}{c_i^{\text{round}} u_{\text{dom}}^t} \text{ where } u_{\text{dom}}^t = \max_i \frac{k_i^t}{c_i^{\text{round}}} \;\forall\; t \in T, i \in I \tag{4.15}$$

$$\tag{4.16}$$

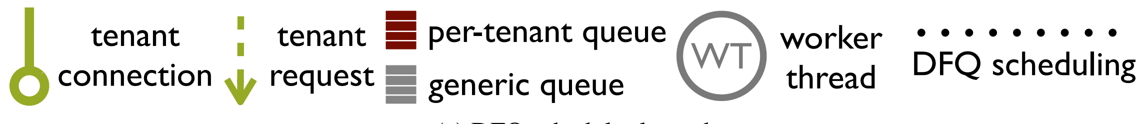
$$r_i^t = r_{\text{DRF}} w^t u_i^t c_i^{\text{round}} \text{ where } r_{\text{DRF}} = \min_i \frac{1}{\sum_t w^t u_i^t} \;\forall\; t \in T, i \in I \tag{4.17}$$

where I is the set of resources and c_i^{round} is the throughput of resource i over a fully backlogged round. In Pisces, the DFQ scheduler accounts for the number of bytes received, bytes sent, and requests (i.e., packets). We examine disk IO in Libra. First, DFQ determines each tenant's dominant (i.e., max until) resource and then normalizes each tenant's resource utilization u_i^t by its dominant resource (4.15). To compute the DRF shares, DFQ computes the minimum over the inverse sums of the (normalized and weighted) tenant utilizations to find the overall DRF allocation. This allocation corresponds to the share that the minimum weight tenant should receive of its dominant resource. Using the DRF allocation, DFQ computes each tenant's per-round resource allocation r_i^t as (4.17). Any excess resources not allocated with DRF are distributed equally among all tenants.

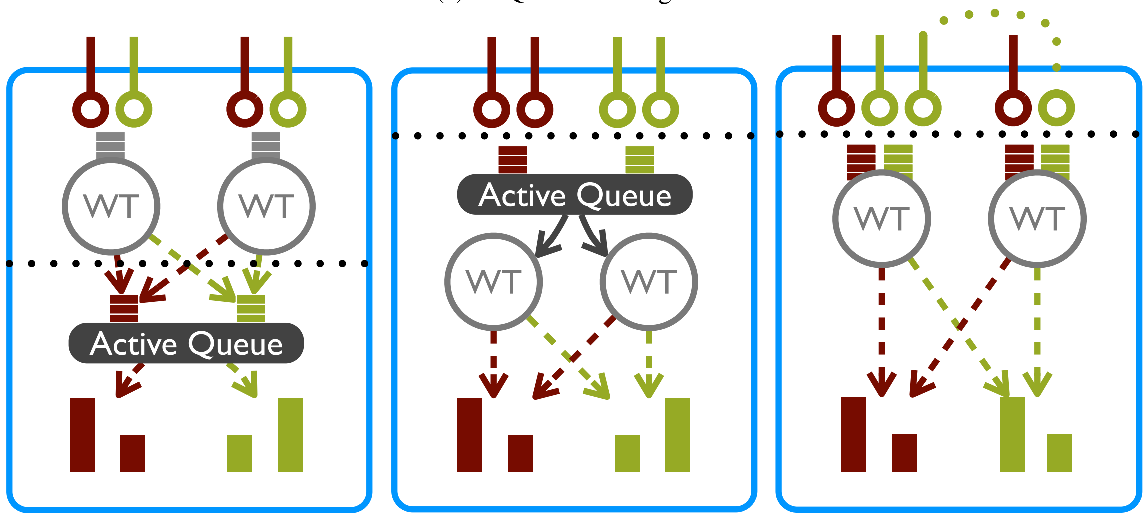
Although DFQ could use these per-round tenant resource allocation vectors directly to service tenant requests, it quantizes them for efficiency. Since DFQ, like DWRR, runs an independent scheduler on each core (thread), it must share the per-tenant allocations across the schedulers. Synchronizing access to the tenant resource vectors via locks would be prohibitively expensive. By quantizing the resource vectors into nominally sized allocations, the DFQ threads need only increment (on round advance) and decrement (on request handling) per-tenant quanta deficit counters, which can be efficiently implemented with atomic primitives. This also allows DFQ to quantize each tenant separately according to the size of their typical requests. Note that multiple quanta may be needed to serve a large request, or a single quantum's resources may span several small requests.

Request processing in DFQ proceeds in rounds. Per Algorithm 4, on each global scheduling round, DFQ allocates the designated per-round quanta to each active tenant (those with queued requests). Recall that the quanta allocations correspond to each tenant's weighted dominate fair share. Each DFQ scheduler thread consumes resources by sub-allocating quanta from the tenants' global pool and converting it into a corresponding tenant resource allocation for each request. This differs from DWRR, where each task (request) is treated as a separate resource principal (tenant) and can be renewed independently by its current scheduler. If a *divisible* request, i.e., one that can be partially completed, requires additional resources, the scheduler adds it to the back of the request queue. This mitigates head-of-line blocking and reduces request delay, similar to VTRR's quanta slicing approach (Section 2.2.3). Otherwise, if the request completes, the scheduler *refunds* any unused resources back to the tenant in the form of accrued quanta. If a tenant runs out of quanta allocation, the scheduler adds its outstanding requests to an exhausted queue.

As in DWRR, each DFQ scheduler thread must perform work stealing at the end of its local scheduling round to ensure that all eligible work has been completed. However, in DFQ this amounts to stealing both eligible tenant requests, i.e., requests that have already been allocated resources, and tenant quanta. If the scheduler manages to steal work, then it resumes request processing. Otherwise, it increments its local round and updates the global round as needed. This ensures that the global scheduling round advances only when every tenant is either inactive (no requests) or exhausted (requests but no quanta) and there is work to do in the next round. Thus, at the end of a global round DFQ guarantees each tenant will receive its weighted dominant fair share of resources. As in GPS, if a subset of tenants are active (backlogged) in the global round, then the round advances faster, since DFQ is work conserving. When all tenants are active, the global round spans its maximum



(a) DFQ scheduler legend



(b) Per-request DRR (c) Per-connection DRR (d) Non-blocking DWRR

Figure 4.3: Scheduling per-connection (c) instead of per-request (b) achieves fairness. Decoupling threads and work-stealing (d) optimizes performance.

duration. By limiting the resources available in each global round c^{round}, DFQ can bound the maximum round duration, and hence achieve fairness with bounded lag (Section 2.2).

Despite the fact that DFQ only enforces DRF on a per-node level, it still provides global max-min fair shares for each tenant. When two tenants have different dominant resources at a node, DRF allocates each tenant a larger local share than it would have received if the tenants had contended for the same resource. In other words, each tenant's share of its dominant resource is lower bounded by the max-min fair share of a single common resource. Thus, even if a tenant's dominant resource varies from node to node, its aggregate share will still equal or exceed its max-min fair share (proportion) of total system resources. This allows Pisces to use a single weight to represent the per-tenant global and local shares, rather than a more complicated weight vector.

4.3.5 Getting Fair Queuing Right

Enforce queuing at the appropriate software layer. Implementing the Pisces resource scheduler may seem straightforward at first, but doing so with low overhead is significant engineering challenge. The most natural approach is simply to place tenant request queues right after request processing in the application, as in Figure 4.3b, and use a simple DRR or virtual time based scheduler. The problem, however, is that resources have already been consumed (to receive bytes and parse the request), which prevents the scheduler from enforcing fairness and isolation for certain network I/O bound workloads. Thus, the Pisces scheduler instead operates prior to application request handling, as in Figure 4.3c, and mediates between *connections* before any resources are consumed. Now, however, the scheduler no longer knows how much work remains in each connection queue. To handle this uncertainty, the Pisces scheduler allocates a fixed number of quanta from the tenant's quanta pool when a connection is dequeued for processing. Any unused quanta and resources are *refunded* back to the tenant.

Avoid queue locking at all costs. Even with the scheduler in the right place, we still need to worry about the scheduler implementation and efficiency. Maintaining a centralized active queue—per-connection DRR in Figure 4.3c—is a natural design point for fair-queuing. A single thread enqueues connections, and separate worker threads pull tenant connections off this queue one-at-a-time, servicing them by consuming quanta resources. While simple and fair, this design is flawed: whenever the active queue is empty, the worker threads must wait on a conditional lock. We found that the overhead of this conditional waiting and waking can reduce the processing of small requests by over 30%. To combat this overhead, Pisces uses a combination of non-blocking per-tenant connection queues [44] and the modified DWRR [38] algorithm described in Algorithm 4.

Eliminate intermediary queuing effects. While the non-blocking design (Figure 4.3d) is highly efficient, it faces one final barrier to achieving max-min fairness. Our measurements showed good fairness for small requests, which are bottlenecked by server interrupts, but poor fairness for bandwidth-bound workloads. This arises for two reasons. First, if the scheduler does not properly wait for write-blocked connections, i.e. those experiencing EAGAIN on `send`, to finish consuming resources before advancing the round, then high-weight, bandwidth-bound tenants could see their remaining quanta wiped out prematurely if additional request are in-flight but have not yet arrived. In other words, any outstanding IO must first compete before the scheduler checks for remaining work to accurately determine whether tenants are have truly quiesced. Second, over-sized TCP send buffers (128 KB) mask network back pressure. On a 1Gbps link with sub-ms round-trip time, the bandwidth delay product is on the order of tens of kilobytes. When multiple tenant connections (>64) contend for output bandwidth, writes can succeed even when the outbound link is congested. By masking the backpressure signal when additional demand is in-flight, the oversized buffer again causes the scheduler to advance the round too soon because it thinks that the tenants have no more work to do. In response, the Pisces DFQ scheduler uses small connection send buffers (6 KB), and waits for I/O-bound connections to finish consuming resources before advancing the round. To prevent worker threads idling excessively due to non-local network congestion, the scheduler uses a short timeout on the order of the maximum round duration (1 ms) that wakes all I/O quiescent threads and allows the round to advance.

4.4 Prototype on Membase

We implemented Pisces on top of the open-source Membase [20] key-value storage system (part of the Couchbase suite). Built around the popular memcached in-memory caching engine, Membase adds object persistence, data replication, and multi-tenancy. Membase relies heavily on the in-memory key-value cache to serve requests and dispatches disk-bound requests to a background thread. Key-value put or delete operations are committed first in memory and later asynchronously written to disk. Membase creates even-sized explicit partitions and directly maps the partitions to server nodes. It can replicate and migrate partitions for fault tolerance, and synchronizes primary and secondary replicas in an eventually consistent manner. For evaluation purposes, we replaced Membase's uniform partition-placement mechanism with one based on a simple greedy heuristic which we use to pre-compute a feasible fair placement based on known (oracle) tenant demand distributions.

We integrated Pisces's fairness and isolation mechanisms into Membase using a mix of languages. Implementing the optimized multi-tenant, DFQ scheduler in the core server codebase required an extensive overhaul of the connection threading model in addition to adding the scheduling and queuing code in C (3000 LOC). Replica selection was implemented in Java (1300 LOC) and integrated directly in the spymemcached [3] client library. Our centralized controller, which implemented both weight allocation and partition placement, comprised approximately 5000 LOC of Python.

4.5 Pisces Related Work

Sharing the network. Most work on sharing datacenter networks has used either static allocation or VM-level fairness. The static allocations by SecondNet [31] and Oktopus [14] guarantee bandwidth but can leave the network underutilized. While more throughput efficient, Gatekeeper's ingress and egress scheduling [58], Seawall's congestion-controlled VM tunneling [60], and FairCloud's per-endpoint sharing [56], respectively provide fairness on a per-VM, VM-pair, or communicating-VM-group basis, rather than per-tenant.

NetShare [36] and DaVinci [32] take a per-tenant network-wide perspective, but the former allocates local per-link weights statically, while the latter requires non-work-conserving link queues and does not consider fairness. In contrast, Pisces achieves per-tenant fairness by leveraging replica selection and adapting local weights according to demand, while maintaining high utilization.

Sharing services. Recent work on cloud service resource sharing has focused mainly on single-tenant scenarios. Parda [28] applies FAST-TCP congestion control to provide per-

VM fair access to a black-box storage system. Pisces modifies the FAST-TCP algorithm to balance per-tenant load across replicated service nodes. mClock [30] adds limits and reservations to the hypervisor's virtual time based proportional share IO scheduler. mClock mediates per-VM IO requests to a shared storage volume, while Pisces's DFQ scheduler operates on a per-tenant level. Argon [73] uses cache management and time-sliced disk scheduling for performance insulation between applications on a single shared file server. Pisces could adapt these techniques for memory and disk I/O resources. Stout [40] exploits batch processing to minimize request latency, but does not address fairness.

Several other systems focused on course-grained allocation. Autocontrol [50] and Mesos [33] allocate per-node CPU and memory to schedule batch jobs and VMs, using utility-driven optimization and dominant resource fairness, respectively. They operate on a coarse per-task or per-VM level, however, rather than on per-application requests. In [24], the authors apply dominant resource fairness to fine-grained multi-resource allocation, specifically to enforce per-flow fairness in middleboxes. However, their DRF queuing algorithm relies on virtual time and $O(n)$ time since it schedules packets by scanning all n per-flow packet queues for the lowest virtual start time.

Chapter 5

Pisces Evaluation

In our evaluation, we consider how the mechanisms in Pisces build on each other to provide fairness and performance isolation by answering the following questions:

- Are each of the four mechanisms (PP, WA, FQ and RS) necessary to achieve fairness and isolation?
- Can Pisces provide global weighted shares and local dominant resource fairness?
- How well does Pisces handle dynamic workloads?

We quantify fairness as the Min-Max Ratio (MMR) of the dominant resource across all tenants, $\frac{x^{\min}}{x^{\max}}$. This corresponds directly to a max-min notion of fairness.

5.1 Experimental Setup

To evaluate Pisces's fairness properties, we setup a testbed comprised of 8 clients, 8 servers, and 1 controller host. Each machine has two 2.4 GHz Intel E5620 quad-core CPUs with GigE interfaces, 12GB of memory, and run Ubuntu 11.04. Pisces server instances are configured with 8 threads and two replicas per partition. All machines are connected directly to a single 1 Gbps top-of-rack switch to provide full bisection bandwidth and avoid network contention effects. Similarly, replica selection and request routing are handled directly in a client-side library to minimize proxy bottleneck effects.

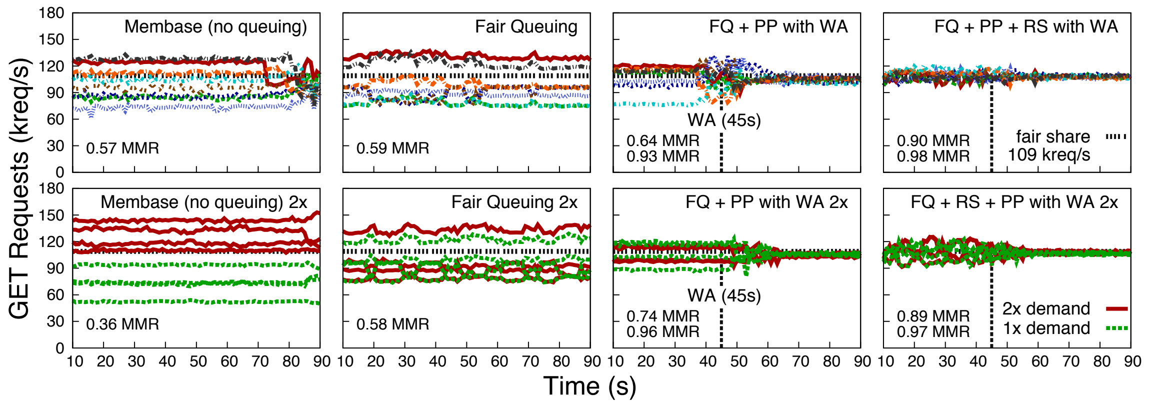
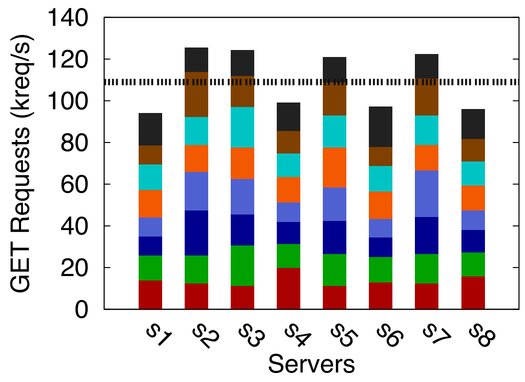


Figure 5.1: System-wide throughput fairness (top) and performance isolation (bottom) with Pisces mechanisms. For experiments involving weight allocation (columns 3 and 4), WA is activated at time 45s.

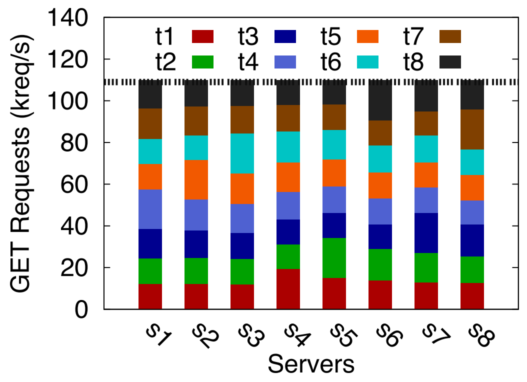
On the clients, we use the Yahoo Cloud Storage Benchmark (YCSB) [19] to generate a Zipf distributed key-value request workload ($\alpha = 0.99$). Each client machine runs multiple YCSB instances, one for each tenant, to mimic a virtualized environment while avoiding the overhead of virtualized networking. Each tenant is pre-loaded with a fully cached data set of 100,000 objects which are hashed over its key space and divided into 1024 partitions. Object sizes are set to 1kB unless otherwise noted. Tenant request workloads include read-only (all GET), read-heavy (90% GET, 10% PUT) and write-heavy (50% GET, 50% PUT). All clients send their requests over TCP.

5.2 Achieving Fairness and Isolation

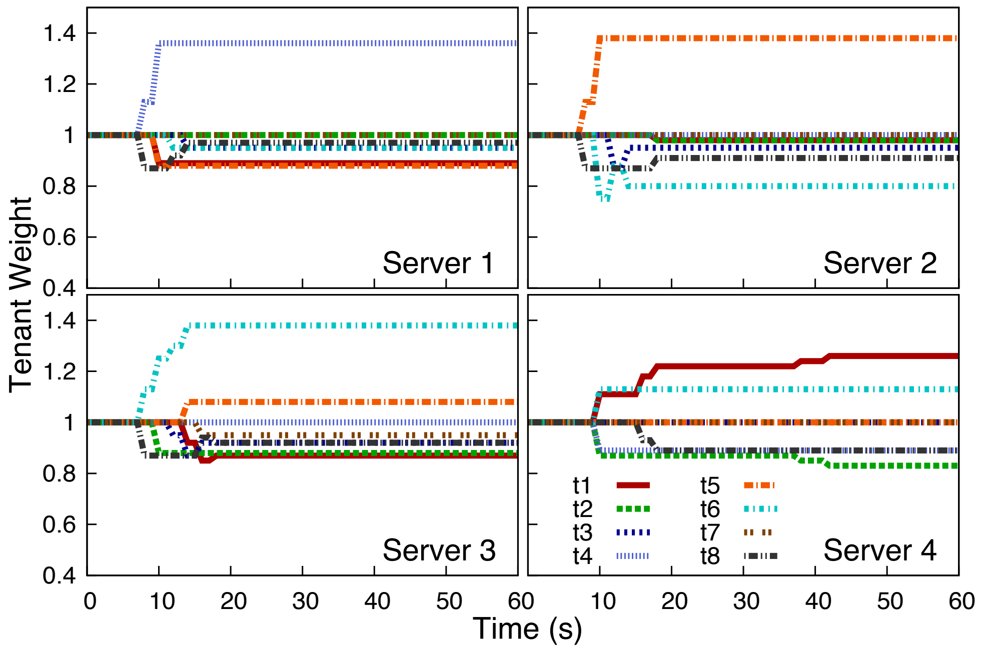
To understand how Pisces's mechanisms affect fairness and isolation, we evaluated each mechanism in turn and in combination, as shown in Figure 5.1. Starting with an unmodified base system (Membase) without fair queuing, we add in fair queuing (FQ), followed by partition placement (PP) and weight allocation (WA). Lastly, we enable replica selection (RS). In the top row of experiments, 8 tenants with equal global weights attempt to access the 8-node system with the same demand. In the bottom row, half of the (equal weight) tenants issue twice the demand of the others to evaluate performance isolation. For illustrative purposes, we first present results for a simple GET workload, and summarize results



(a) Uniform partition placement



(b) Demand-aware placement



(c) Local tenant weights evolving with weight allocation

Figure 5.2: Skewed demand can lead to infeasible partition mappings (a). Placing partitions according to tenant demand and node capacities ensures feasibility (b). Weight allocation, in turn, adjusts the per-node tenant weights to the demand (c) (4 of 8 shown).

for more complex workloads in Table 5.1. For the 1kB GET workload, the fair share (1.0 MMR) is bandwidth-limited at 109 kreq/s per node.

5.2.1 Unmodified Membase

The unmodified system provides poor throughput fairness (0.57 MMR) between tenants. This is largely due to the inherent skew in the tenant demand distribution which, per Section 3.4, can lead to an infeasible partition mapping. Figure 5.2a shows tenants 3 and 7 contending for hot partitions at server 2, while tenants 2, 4, and 6 collide on server 3 under the default, demand-oblivious placement. In contrast, Figure 5.2b shows how packing the partitions according to demand resolves the node capacity violations to ensure feasibility.

5.2.2 Multi-tenant Weighted Fair Queuing

Unsurprisingly, fair queuing alone barely improves fairness (0.59 MMR) due to over-contention for node resources under the uniform partition placement. FQ can only enforce (not change) the policies computed by higher-level mechanisms, whether they are feasible or not. However, fair queuing proves essential for performance isolation. Where Membase falls flat under excess demand from the 2x tenants (bottom row), fair queuing achieves the same degree of fairness under all conditions. Infeasible placement and unmatched weights may allow tenants to consume more than their aggregate fair share, but they cannot consume more than their local allocations.

5.2.3 Fair Queuing and Partition Placement

Despite starting with the pre-computed feasible partition placement shown in Figure 5.2b, fairness only improves marginally (0.64 MMR). Although the tenant demand should fit within node capacity, local hotspots still remain. The mismatch between the hotspots and the initially fixed (uniform) local weights allows tenants with an unduly large share at a given node, i.e., the local share exceeds the fair partition demand at the node, to ”over” consume and exceed their global fair share. Once weight allocation starts (45s), the local shares converge within 10 seconds (5 iterations) to their optimal fair values (0.93 MMR).

5.2.4 Fair Queuing, Partition Placement, and Replica Selection

When enabled, replica selection improves fairness (0.90 MMR) by smoothing out hotspots in the demand distribution. However, under this particular partition mapping, RS is unable to eliminate all demand skew on its own. With weight allocation running (after 45s), RS is

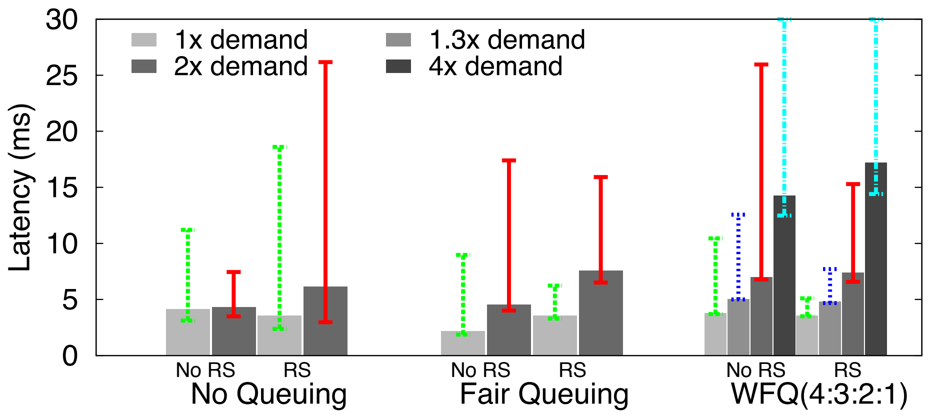


Figure 5.3: Fair queuing protects request latencies for even and weighted tenant shares.

able to adjust the selection policy in tandem with WA to find the optimal fair solution, as described in Section 4.2.2, and achieve near ideal fairness (0.98 MMR). Using a different feasible partition mapping (not shown), RS is able to achieve $>$ 0.99 MMR even without WA, due to the more efficient placement and work-conserving local shares.

5.2.5 Performance Isolation

In the bottom row of Figure 5.1, half of the (equal weight) tenants issue twice the demand of the others to stress the system's performance isolation. Unmodified Membase allows the 2x demand tenants to consume additional resources, degrading fairness. In contrast, fair queuing denies the 2x tenants any additional share, preserving fairness when enabled. Interestingly, unmodified Membase with a feasible partition mapping and replica selection (not shown) can achieve high fairness ($>$ 0.95 MMR) for equal demand tenants, but, again, not in the performance isolation scenario ($<$ 0.68 MMR).

In addition to throughput fairness and isolation, Pisces also provides a measure of latency isolation as well. The first two groups in Figure 5.3 show the average median latencies for the 1x and 2x demand tenants in the previous experiments. The max error bar indicates the 95th percentile, while the min error bar indicates the spread between the tenants' median latencies. Without fair queuing, there is little isolation with median latencies for both 1x and 2x tenants hovering around 4 ms. Adding replica selection improves isolation where the 2x tenants experience about 1.7 times the median latency as the 1x tenants. However, the latency spread (3.2 ms) spans the entire gap between them. With fair queuing, the median latencies are far more well-behaved. The 2x tenants receive a 2.1 times latency penalty with minimal (<1 ms) spread.

5.2.6 Fairness Over Different Workloads

For a more thorough evaluation of Pisces fairness, we experimented over a range of log-normal distributed object sizes, where each tenant has object sizes drawn from a distribution with the same mean value (1kB or 10B) and standard deviation (0, 0.5, or 1). We also varied the workloads between all read-only, all read-heavy, or a mix of read-only, read-heavy, and write-heavy. Table 5.1 summarizes these results for Pisces with all mechanisms enabled. We measured mean bandwidth consumption for 1kB objects and mean request rates for 10B objects, as well as median request latency (1ms averaged) and average fairness (in terms of bandwidth consumed or request rates, respectively). For mixed workloads, values for both GET/PUT (Out/In bandwidth) requests are shown.We also include a fairness comparison in terms of the MMR ratio between Pisces and unmodified Membase.

Pisces is able to achieve over 0.94 MMR fairness for most size variations and workloads. Moreover, Pisces exceeds the fairness of the unmodified base system by more than 1.3 times for most cases as well. The mixed workload for 1kB objects, however, proved to be a particularly troublesome combination. While the read-heavy and write-heavy tenants received their appropriate shares of inbound write bandwidth (0.99 MMR), the read-only tenants were able to consume more outbound read bandwidth than either one, resulting in an overall fairness between 0.63 and 0.64 MMR. One reason for this is the head-of-line blocking of a tenant's read requests due to its write requests fully consuming its current inbound bandwidth allocation. This can push subsequent read requests to the next scheduler round, despite having unconsumed outbound bandwidth remaining. Since read-only

Fixed Size Objects				
1kB Mean	**BW (Gbps)** Out/In	**Lat (ms)** GET/PUT	**MMR** Out/In	**MMR ratio** Out/In
Read-Only	6.95	2.48	0.99	1.73
Read-Heavy	6.87/1.30	2.99/2.15	0.98/0.98	1.56/1.58
Mixed	6.81/2.06	2.55/2.41	0.68/0.77	1.62/1.05
10B Mean	**Tput (kreq/s)** GET/PUT	**Lat (ms)** GET/PUT	**MMR** Requests	**MMR ratio** Requests
Read-Only	3,160	3.37	0.97	1.33
Read-Heavy	2,567/285	3.04/8.19	0.98	1.28
Mixed	2,343/445	2.80/7.79	0.96	1.5

LogNormal 0.5				
1kB Mean	**BW (Gbps)** Out/In	**Lat (ms)** GET/PUT	**MMR** Out/In	**MMR ratio** Out/In
Read-Only	6.93	3.67	0.99	1.57
Read-Heavy	6.82/1.32	3.34/2.62	0.99/0.98	1.32/1.29
Mixed	6.81/2.08	3.11/2.75	0.67/0.77	1.81/0.92
10B Mean	**Tput (kreq/s)** GET/PUT	**Lat (ms)** GET/PUT	**MMR** Requests	**MMR ratio** Requests
Read-Only	3,151	3.52	0.97	1.26
Read-Heavy	2,593/288	2.96/8.30	0.98	1.42
Mixed	2,325/444	2.88/7.84	0.96	1.32

LogNormal 1.0				
1kB Mean	**BW (Gbps)** Out/In	**Lat (ms)** GET/PUT	**MMR** Out/In	**MMR ratio** Out/In
Read-Only	6.94	5.20	0.99	2.15
Read-Heavy	6.90/1.35	4.70/3.45	0.99/0.99	1.65/1.57
Mixed	6.76/2.01	4.01/3.71	0.63/0.74	1.85/1.19
10B Mean	**Tput (kreq/s)** GET/PUT	**Lat (ms)** GET/PUT	**MMR** Requests	**MMR ratio** Requests
Read-Only	3,104	3.36	0.96	1.35
Read-Heavy	2,540/282	3.00/8.49	0.94	1.34
Mixed	2,295/437	2.94/7.78	0.94	1.38

Table 5.1: Pisces performance over a range of log-normally distributed object sizes and GET/PUT workloads

tenants never bottleneck on inbound bandwidth, they can fully consume their share (and more) of outbound bandwidth.

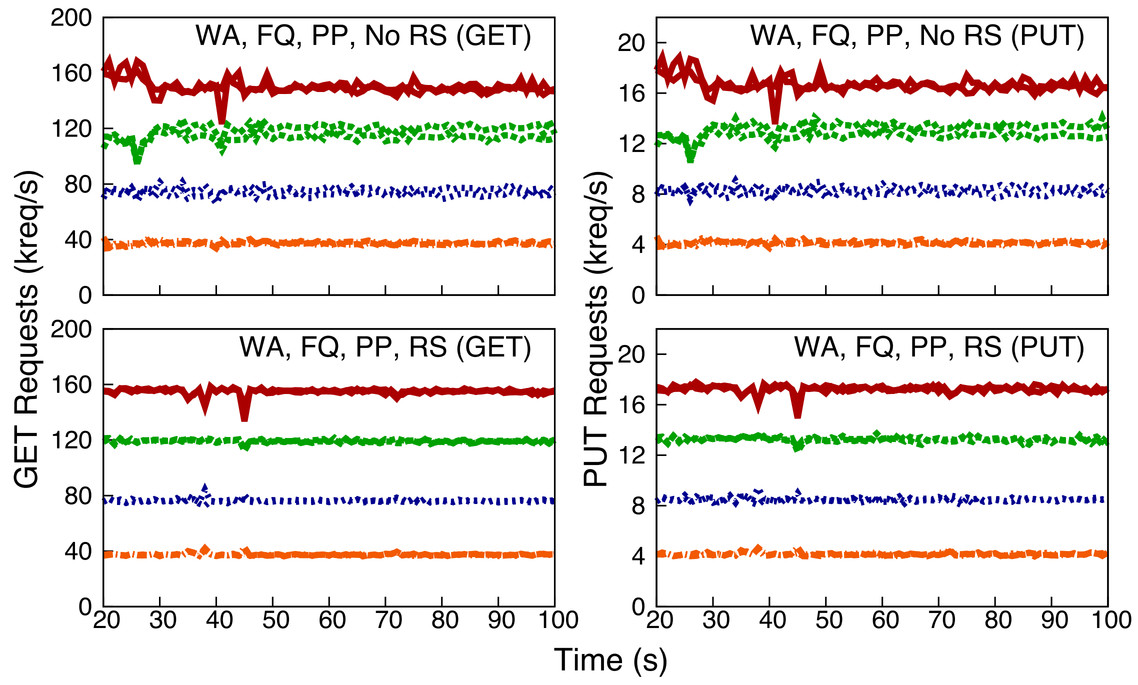
To fix this issue, we modified the clients to prioritize read requests over write requests when sending requests to the servers. While this largely resolved the issue for read-heavy tenants ($>$ 0.9 MMR), the write-heavy tenants remained obstructed by the bandwidth bottleneck. The low MMR ratio for writes (bytes in) compared to Membase is also attributable to this performance write-bottlenecking variance. For 10B workloads, the problem disappears ($\geq$ 0.95 MMR) since both read and write workloads share a single bottleneck: request-rate without suffering packet-level congestion effects. Pisces is able to once again claim its fairness crown from Membase for these cases ($>$ 1.28x more fair than Membase).

5.3 Service Differentiation

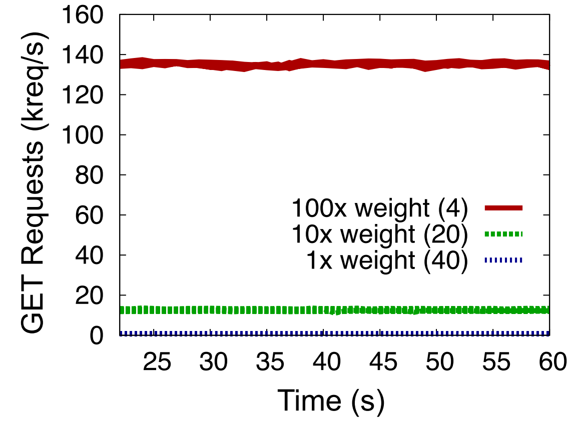
So far, we have demonstrated that Pisces's mechanisms, working in concert, can achieve both isolation (FQ) and nearly ideal equal fair shares (PP + WA or RS). We now turn our attention to providing *weighted* fairness at the global level (for service differentiation) and at the local level (for dominant resource fairness).

5.3.1 Global Differentiation

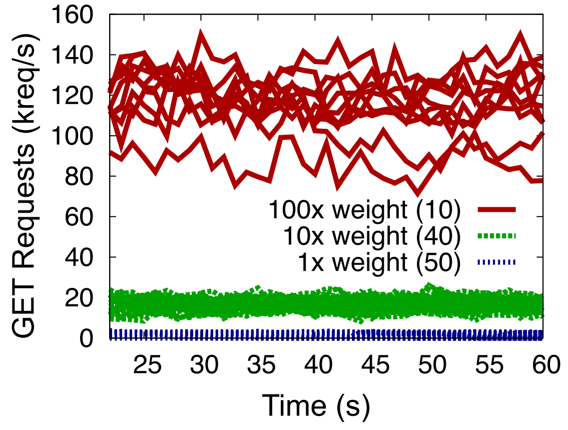
In Figure 5.4a, the tenant are assigned global weights in decreasing order 4:4:3:3:2:2:1:1 to differentiate their aggregate share of the system resources. Both with and without replica selection, Pisces is able to achieve high global weighted fairness ($>$ 0.9 MMR) for both in and out bandwidth under a 1kB request read-heavy workload. Similarly, request latency remains strongly isolated between the tenants. Since all tenants generate the same demand, the lower weight tenants (3, 2, and 1) exceed their share by 1.3, 2, and 4 times respectively. Per Figure 5.3, the latencies of these tenants closely mirror their demand ratios. In both cases, the median (or mean) latency spread is small ($<$1 ms), except for that of the smallest-weight tenant ($<$2.7 ms).



(a) Global weighted 90% GET / 10% PUT throughput



(b) 64 tenant weighted thruput



(c) 100 tenant weighted thruput

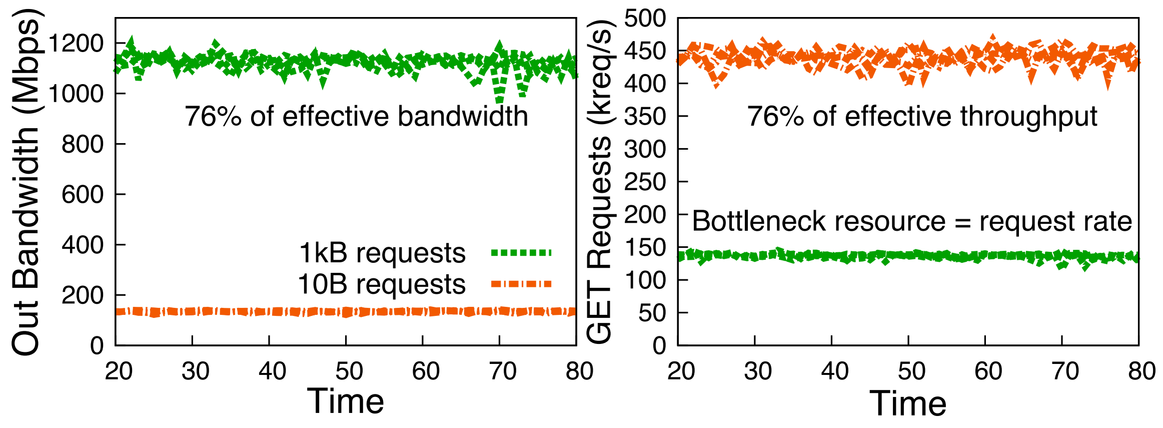
Figure 5.4: Subfigure (a) demonstrates global 4:3:2:1 weighted fair sharing for a read-heavy workload. In (b) and (c), Pisces abides by the skewed tenant weights on an 8 and 20 node cluster respectively.

To further stress the fairness properties of the system, we ran two larger scale experiments where the number of tenants far exceeded the number of servers, to mimic more realistic service scenarios. In Figure 5.4b, each of 64 tenants reside on 4 out of the 8 available servers with each server housing 32 tenants. To reflect the highly skewed nature of tenant shares, i.e. a few heavy hitters and many small, low-demand users, we configured the tenant weights along a log-scale: 4 tenants with weight 100, 20 with weight 10, and 40 with weight 1. Within each weight class, Pisces achieves > 0.91 MMR. Between classes, however, weighted fairness decreases to 0.56 MMR. This deviation is due to the limits of the DWRR scheduler token granularity. With such highly skewed weights, tenants in the smallest weight class (1) only receive fractional tokens, which means their requests are processed once every few rounds, which results in a lower relative share. Fairness between the weight 100 and weight 10 tenants, however, remains high (0.91 MMR). In practice, to work around the limited resolution of the WFQ scheduler, the service provider can cap the number of tenants per server to ensure reasonable local weights (token) and match the desired rate guarantees. Figure 5.4c shows a larger scaled out experiment, with 100 tenants resident on 6 of 20 servers (30 tenants per server). While we see a qualitatively similar result, the actual fairness degrades considerably (average 0.68 MMR between high and medium weight tenants, 0.46 MMR across all classes). However, this is mostly due to performance variance on the scale-out testbed [55] arising from CPU scheduling and network bottleneck effects, which affects the high-weight tenants disproportionately. As a result, the low-weight tenants can consume a larger share.

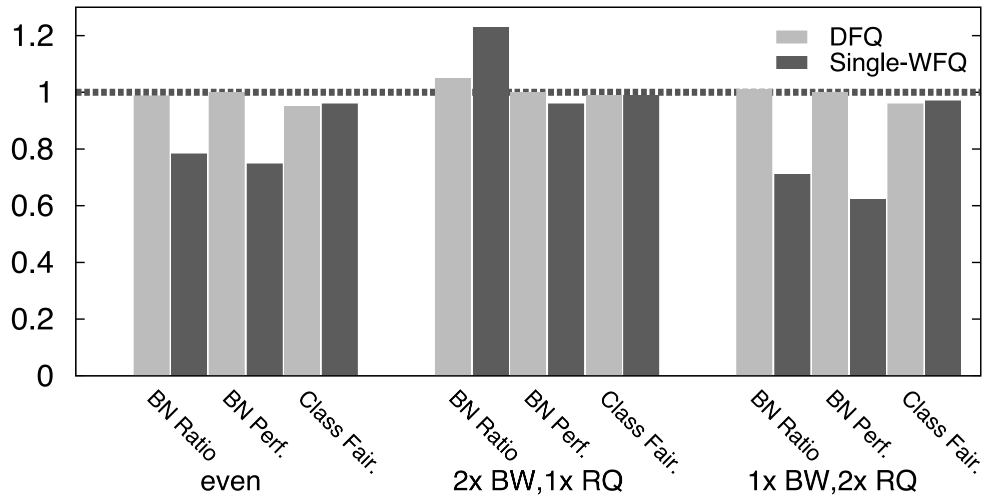
5.3.2 Local Dominant Resource Fairness

While Pisces provides weighted fairness on the global level, we want to ensure that each tenant receives a weighted share of its dominant resource on the local level. To stress different dominant resources and their corresponding bottlenecks, we experimented with four tenants requesting 1kB objects (bandwidth limited), while another four operated on 10B objects (request-rate limited). Under even weighting, Figure 5.5a shows how Pisces enforces fairness within each dominant resource type (> 0.95 MMR) and evenly splits the dominant resource shares between types: the 1kB tenants receive ~ 76% of the effective outbound bandwidth and the 10B tenants receive ~ 76% of the effective request rate.[1]

[1]This is lower than the optimal rates since transmitting a 1kB request takes longer than processing a 10B request.



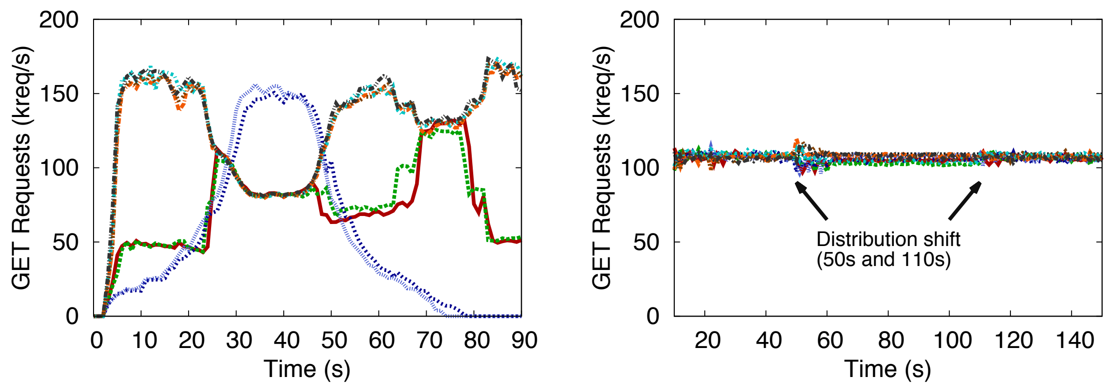
(a) Dominant resource fairness between 1kB and 10B workloads



(b) Bottleneck ratio fairness and performance with and without multi-resource queuing

Figure 5.5: Pisces provides dominant resource fairness between bandwidth (10kB) and request-rate (10B) bound tenants (a). Compared to single-resource fair queuing, DRF provides a better share of the bottleneck resource (BN Ratio) and better performance (BN Perf) over different tenant weights.

Although the tenants differ in dominant resource, they share the same bottleneck resource (request rate in this case). By computing the bottleneck resource ratio (BNR) between the different dominant resource tenants, we can directly determine the dominant resource shares as shown in section 3.4 without having to estimate the effective resource rates. In this instance, the BNR is around 3.2, which gives the 10B tenants 76% of the request rate, and allows the 1kB tenants to consume 76% of the bandwidth.



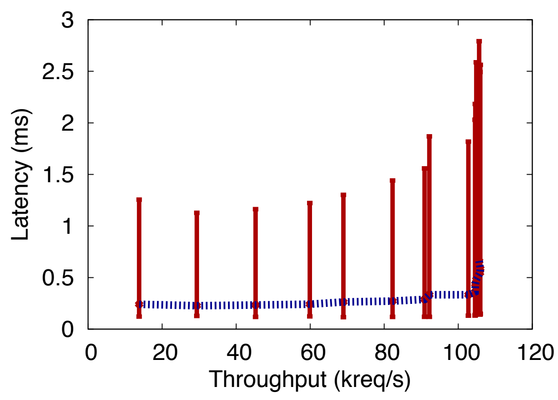
(a) Dynamic intensity (b) Dynamic distribution

Figure 5.6: Pisces responds to demand dynamism (a) and distribution shifts (b) to preserve fairness.

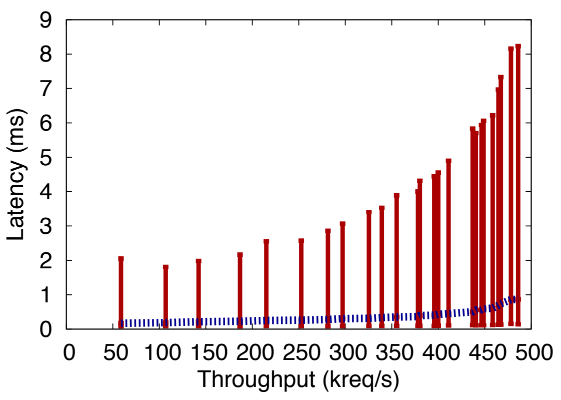
Using BNR, we compare the dominant resource fairness of the DFQ scheduler versus the single-resource (request) version for even weighted tenants, 2x weighted 1kB (bw) vs. 1x weighted 10B (rq) tenants, and 1x weighted 1kB vs. 2x weighted 10B tenants. For the even and 1x bandwidth, 2x request cases, the bottleneck resource is request rate. In the 2x bandwidth, 1x request scenario, the tenants bottleneck on bandwidth. As Figure 5.5b shows, the DFQ scheduler outperforms the single resource version in all cases, achieving the best normalized BNR value and bottleneck resource performance (BN Perf). The single-resource scheduler holds a slight edge in fairness between tenants of the same dominant resource class (Class Fair) in the even and 1x bandwidth, 2x request cases. As in the mixed workload experiments, achieving the highest 10B request throughput required additional packet prioritization. We enabled the linux priority queue scheduling discipline to reduce the scheduler delay for small requests while waiting for connection send buffers to clear for the 1kB tenants. Employing additional techniques for prioritizing low bandwidth, high-throughput requests would likely further improve fairness.

5.4 Dynamic Workloads

Dynamic workloads present a challenge for any system to provide consistent, predictable performance. In Figure 5.6, 2 bursty demand tenants (weight 1), 2 diurnal demand tenants



(a) Pisces μ-benchmark (1kB)



(b) Pisces μ-benchmark (10B)

Figure 5.7: Median throughput versus latency micro-benchmark with 99th-percentile error bars.

(weight 2), and 4 constant demand tenants (weight 1) access the storage system. Initially, the tenants consume less than the full system capacity which allows the constant tenants to consume a larger proportion. As the diurnal tenants ramp up between 0 and 20 seconds, they begin to consume their share of throughput which cuts into the excess share consumed by the constant tenants. Around 20s, both the bursty tenants and diurnal tenants ramp up to their full load, which results in a nearly 2 to 1 ratio, according to the tenant weights. The diurnal tenants tail off around 50s along with the bursty tenants which allows the constant demand tenants to, once again, consume in excess of their fair share ($\sim$ 80 kreq/s). Lastly, at 70s, the bursty tenants issue one last barrage of requests, which forces the constant tenants to share the throughput equally. Both with and without replica selection enabled, Pisces is able to handle the demand fluctuations and provide fair access to the storage resources.

Demand distributions can evolve as well. In Figure 5.6b, the tenants switch from the current Zipfian demand distribution to a different, equally skewed distribution at 50s, and then switch back to the original at 110s. With WA and RS working together, Pisces is able to preserve fairness ($>$ 0.94 MMR), despite the potential ”infeasible” mismatch of the partition demand for the new distribution and the original partition mapping.

5.5 Efficiency and Overhead

To assess the efficiency and overhead, we ran a micro-benchmark of a single Pisces node. In this experiment, tenants issue requests at increasing rates to a single service node. As shown in Figure 5.7, Pisces is able to achieve over 106 kreq/s for 1kB requests, which is > 96% of Membase throughput, with the same average request latency (0.14 ms). For 10B requests, Pisces actually outperforms Membase by 20% (485 vs. 405 kreq/s) with lower average request latency (0.15 vs. 0.16 ms) due to DFQ's work stealing mechanism.

Chapter 6

Libra

Up to this point, our discussion of Pisces has largely focused on memory-intensive storage systems where the majority of tenant requests can be satisfied either by a cache-hit for reads or asynchronous write backs and are bottlenecked by the network. However, the simplifying assumptions that Pisces makes as a result are not suitable for tenants bottlenecked by disk IO, i.e., those with large working sets or who require synchronous write through persistence. From here on out, we will use *reservation* to mean tenant-specified normalized (1KB) app-request throughput e.g., GETs and PUTs per second and *allocation* to refer to low-level, system-provisioned IO resources.

Although Pisces provides max-min fair shares of system-wide throughput, it does not differentiate between different request types. Rather, the local storage node scheduler averages each tenant's workload over all request types to determine its dominant resource. However, tenant throughput reservations are generally given in terms of request-specific rates, each with their own particular disk IO characteristics. Pisces also assumes constant, or at least stable per-node throughput capacity. While this may be true of network resources, disk IO throughput can be highly volatile. Lastly, Pisces can apply DRF to schedule over network resources because they are largely independent, i.e., full-duplex. Disk IO

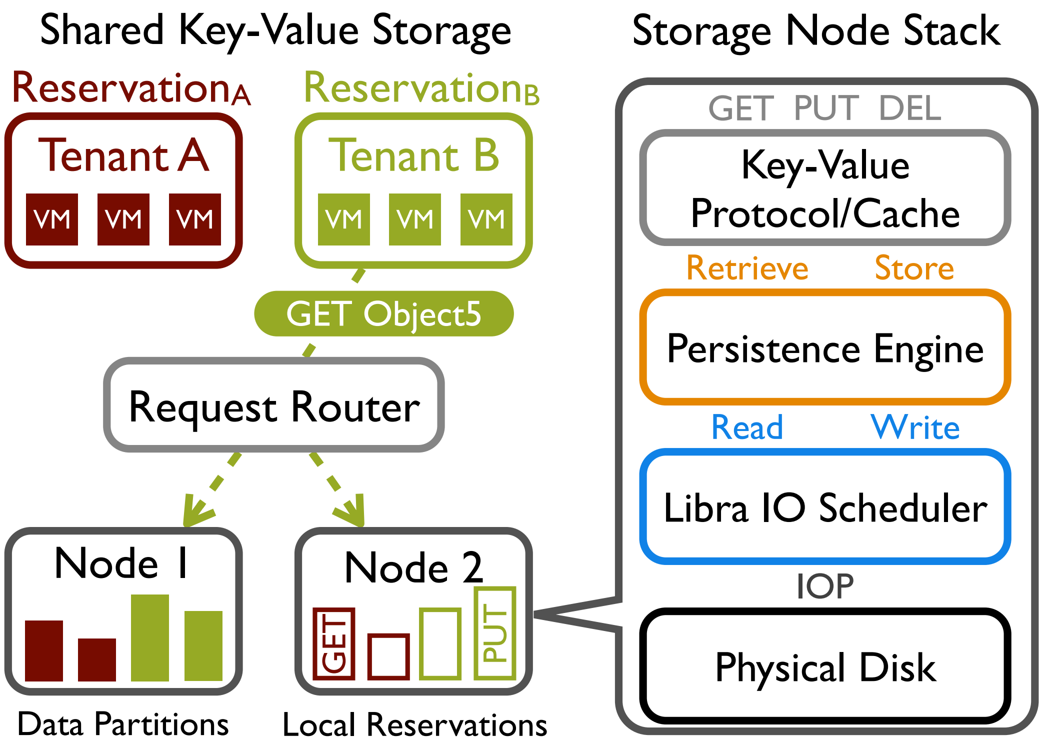


Figure 6.1: In the multi-tenant storage architecture (left), each storage node runs a complex storage stack (right). Libra (right) provisions low-level IO resource allocations and schedules tenant IO to achieve per-node tenant app-request reservations that provide the basis for system-wide reservations in a storage service like Pisces.

operation throughput and IO bandwidth, on the other hand are highly correlated since reads and write share the same data channel and physical storage substrate.

The effectiveness of the system-wide resource allocations determined by Pisces' higher level mechanisms (Partition Placement, Weight Allocation, and Replica Selection) ultimately depend on how well the low-level mechanisms at each storage node enforce local app-request reservations. Although key-value storage systems like Pisces typically expose a simple request API to their clients (e.g., GET or PUT), the internal architecture of each storage node can be layered and complex, as shown in the middle of Figure 6.1. When a tenant request arrives at the storage node, the protocol layer reads and parses it from the network. If the requested object is cached, it is returned immediately for a GET request, while PUTs and cache misses continue on to the persistence engine, which manages the on-disk data layout for updates and retrievals. Internally, the persistence engine issues an often complex series of IO requests to handle the application-level request. The IO scheduler submits these IO requests to the storage device (SSD) which executes the IO operations, percolating results (e.g., object data for reads) back up the stack.

If the individual nodes fail to arbitrate between disparate tenants' requests for objects in collocated partitions at the IO operation level in the storage stack, both local and system-wide tenant reservations would suffer. Thus, in the remainder of this thesis, we turn our attention from the system-wide reservation problem to focus on provisioning disk (SSD) IO resources to achieve tenant app-request reservations. We present Libra, an IO resource scheduling framework that extends the Pisces DFQ scheduler to handle disk IO. At each storage node, Libra *provisions* low-level IO resource *allocations* to satisfy each tenant's local throughput *reservation* by tracking application request cost and mediating tenant IO resource consumption. Libra leverages high-throughput, low-latency SSD storage to provide predictable IO performance for disk-bound workloads, while preserving high utilization. SSDs are fast becoming a viable option for high-throughput and low-latency cloud storage, as evinced by Amazon's DynamoDB [10] and Rackspace's Cloud Block Storage [57] offerings.

To our knowledge, Libra is the first multi-tenant IO scheduler to provide app-request throughput reservations while preserving high-utilization for SSD-based key-value storage. Providing these per-tenant app-request reservations requires answering three essential questions: (i) how much low-level disk-IO does a tenant's GET/PUT request, and hence its reservation, generate, (ii) what is the true IO resource capacity of the underlying SSD media, and most crucially (iii) how should the system schedule and account for the cost of individual IO operations (IOPs) to enforce tenant IO allocations? The difficulty in answering these questions lies in the non-linearity and complexity of modern storage stacks:

- *IO operations are subject to amplification:* In modern storage engines, a single application-level request can trigger *multiple* IO operations (e.g., a 1KB PUT written once to a log and then to a data table) which can vary non-uniformly with tenant workload distribution (e.g., by the GET/PUT ratio and request size).

- *IO throughput degrades under interference:* Disk-level IO interference between reads and writes can cause severe degradation in IOP/s and IO bandwidth, which also varies unpredictably with tenant IOP size and workload.

- *IO cost varies non-linearly with operation size:* Even for pure read or write workloads, IOP/s and bandwidth vary non-linearly with operation size, shifting bottlenecks from the controller (IOP) to the data channel (bandwidth) as IOP sizes increase.

Libra uses two key techniques to combat these complexities.

(1) Track app-request IO cost. Libra marks tenant IO operations by their associated application request across all foreground and background operations in the storage stack. In this way, Libra can attribute secondary IO costs back to the originating request type and track each tenant's IO resource consumption profile. Libra uses these profiles to provision the IO resource allocations necessary to achieve the tenant app-request reservations according to their amplified IO costs.

(2) Schedule and allocate "Virtual" IO operations. Libra charges IO resources in terms of *virtual IO operations* (VOP) using an IO cost model that captures the non-linear performance characteristics of the underlying SSD media. This allows Libra to accurately account for tenant IO over a wide range of IOP sizes and enforce low-level tenant IO resource allocations. To handle the effects of IO interference, we systematically examine how IO throughput capacity (VOP/s) varies under conflicting workloads, e.g., read vs. write requests, large vs. small ops. Libra uses this data to construct a simple IO capacity model that safely (under)estimates the amount of *provisionable* IO resources available.

Using these techniques, Libra is able to provision at least half of the maximum (i.e., interference-free) SSD IO throughput for tenant application-level throughput reservations in our LevelDB-based [37] prototype storage node. Libra achieves these reservations over a wide range of workloads including intermixed reads, writes, and varying IOP sizes, with highly variable interference effects. Although up to half the IO resources may be left unprovisioned, as in Pisces, the Libra scheduler still preserves high utilization by allowing tenants to share any excess IO throughput in a work-conserving manner. Under most realistic workloads that exhibit a mix of operation sizes and types, this provisionable resource gap drops to 16% or less.Central to Libra's ability to accurately provision IO resources is our IO cost model. This model allows Libra to achieve more accurate allocations—0.98 min-max ratio (MMR) across equal-allocation tenants—compared to other extant IO cost models (< 0.84 MMR), as shown in Chapter 7.

6.1 Libra Design

We first introduce the high-level design of Libra, then describe the confounding factors inherent to storage stacks in detail and how Libra addresses each of these challenges.

6.1.1 The Libra IO Scheduler

The Libra framework is a multi-tenant resource scheduler that supports multiple IO devices and provides per-tenant app-request reservations. It extends the Pisces DFQ scheduler, described in Section 4.3.4, to include per-resource cost models and per-request resource profiles. Figure 6.2 shows the basic Libra design. While the Pisces scheduler manages network resource consumption in the protocol layer, as packets are read and written, the Libra scheduler sits below the persistence engine to mediate low-level IO operations before they reach the underlying SSD. Libra manages SSDs based on their measured IO throughput capacity and the cost model associated with their IO operations. The cost model allows Libra to charge a commensurate IO cost for each tenant IO operation, similar to h in virtual time based schemes as described in Section 2.2.2. To satisfy the tenant app-request reservations, the Libra resource policy provisions IO resource allocations based on the observed IO cost of a tenant's requests.

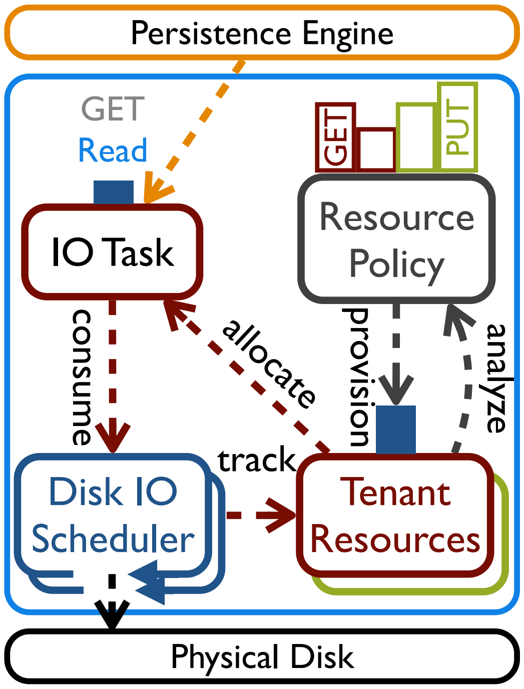


Figure 6.2: Libra IO scheduler

On the fast scheduling path, the persistence engine issues IO tasks (reads and writes) to the Libra scheduler, tagged with the originating resource principal (tenant) and application request (GET or PUT). In each scheduling round, Libra consumes IO resources for each tenant IO task according to the underlying cost model. Libra interleaves IO operations from different tenants in a deficit round robin [62] fashion, scheduling tasks until all tenants have either run out of work or exhausted their IO allocation, which starts a new round. The application request tag allows Libra to track the consumed resources for each request and build a tenant resource profile. On the slow provisioning path, Libra's resource policy periodically analyzes these resource profiles to determine the necessary IO resources to satisfy the app-request reservations and provisions each tenant's resource allocation accordingly. Should tenant workload fluctuations cause the cost of locally allocated IO operations to exceed the IO throughput, Libra can signal Pisces's higher-level policies to migrate partitions and redistribute local reservations.

6.2 The Problem of Predictability

In this section, we examine the nature of IO amplification, IO interference, and non-linear IO performance in detail.

6.2.1 Non-uniform IO Amplification

The cost of an application request in IO resources depends largely on the persistence engine's data format and the number of IO operations needed to satisfy it. Modern storage engines issue multiple IO operations to handle both GETs (reads) and PUTs (writes), with some executed asynchronously. For instance, most engines employ an append-only write-ahead log (WAL) to sequentialize writes and reduce request latency. Although the WAL ensures reliable failure recovery, sequential scans are prohibitively expensive for servicing normal reads. Eventually, the storage engine must *re-write* (FLUSH) the object data in a more efficient, indexed format for retrieval.

Storage engines with immutable data formats, such as log-structured merge (LSM) trees [49, 17, 12, 37], do not allow in-place writes and instead use background processes to cull stale objects and merge data indices. These COMPACT operations entail (potentially many) sequential reads and writes. Compaction cost and frequency are non-uniform and depend heavily on the storage workload. All write-heavy workloads will eventually trigger compaction; however, writes to objects distributed uniformly over a tenant's key space experience few overwrites, resulting in less compaction data savings (and thus more write IO) than compacting data from a highly skewed distribution that frequently overwrites the same keys.

App-request reads can be equally complex even though they do not generate any background operations. Indexed storage engines require one or more page-sized (e.g., 4KB) block reads to find the key index, and another read to load the object data block(s). In-place engines like InnoDB that modify objects in existing data files require extensive file locking to handle concurrent reads and writes, but do not consume additional IO. Immutable for-

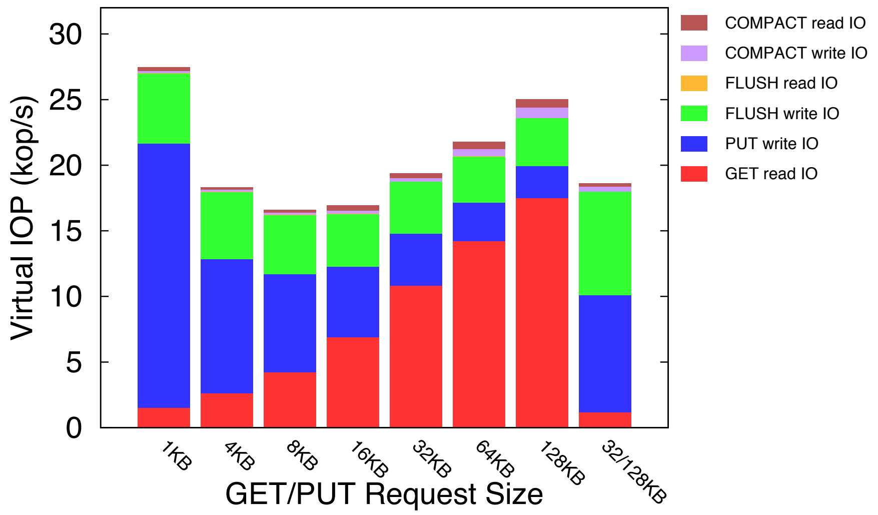


Figure 6.3: App-request IO consumption varies non-uniformly with tenant workload.

mats, on the other hand, may require additional (indexed) lookups. For a given lookup key, any data file with an overlapping key range is a potential search candidate.

To illustrate IO amplification, we measured the app-request IO cost for a tenant with a 50:50 GET/PUT workload, spread over a range of request sizes, accessing our LevelDB-based storage prototype which implements an LSM tree. All keys are sampled uniformly over the keyspace. Here, IO cost is measured in Virtual IOPs (VOP), Libra's unified metric for IO cost and throughput, which we define later in §6.3.3. Figure 6.3 shows the breakdown of app-request IO consumption. At small request sizes, PUT requests consume the majority of IO, since small objects are individually appended to the WAL at a high IO cost-per-byte. As request sizes increase, the PUT cost decreases, in keeping with the lower cost-per-byte of larger IOPs. FLUSH costs remain relatively constant since the entire in-memory data set is written to disk sequentially at a single IOP size regardless of the original object size.

The upswing in GET IO costs at large request sizes shows how IO amplification can vary with workload. Larger PUT requests generate more frequent FLUSHs of the size-limited WAL, due to the higher IO bandwidth. This in turn expands the set of live data files until a COMPACT can merge them. Since PUTs writes keys uniformly over the keyspace, each FLUSHed data file spans a large section of the keyspace. Finding a random GET key forces LevelDB to search a greater number of eligible data files, at the cost of at least one (4KB) index block read per file. In contrast, the last workload in the figure stresses different regions of the keyspace for GETs and PUTs. Here, the 32KB GETs terminate after searching only a single (pre-existing) indexed data file covering its key range.

6.2.2 Unpredictable IO Interference

IO throughput can vary drastically and unpredictably with the type of workload (read vs. write, random vs. sequential) and IOP size. Modern storage engines intentionally sequentialize their IO accesses when possible, e.g., by using write-ahead logs and batching, to mitigate interference and improve performance. Sequential workloads are generally more efficient than their random counterparts, even for SSDs [4]. However, as more distinct tenants access a storage node, the overall IO workload becomes more random due to request interleaving, which can degrade IO throughput.

Despite having much more consistent random IO performance than HDDs, which are bound by mechanical seek delay, SSDs still experience considerable read/write interference. SSDs exploit die-level parallelism [4] (i.e., multiple NAND chips and data channels) to stripe and interleave reads and writes, which reduces resource contention. However, writes are generally much more expensive than reads. Not only do SSD NAND writes take longer to complete than reads [54], but they also incur a heavy erase-before-write penalty. The SSD must first be clear a set of data blocks (4-8 KB) in erase-block increments ($\geq$ 256 KB) before overwriting them with new values. When interleaved, write IOPs can adversely affect read latency and overall IOP throughput.

SSD firmware typically employs a log-structured approach to minimize the overwrite penalty by first appending new data writes onto pre-erased blocks. It then manages a "flash-translation layer" in memory that maps the logical block address of the modified data to the new physical block address. However, the SSD must occasionally perform garbage col-

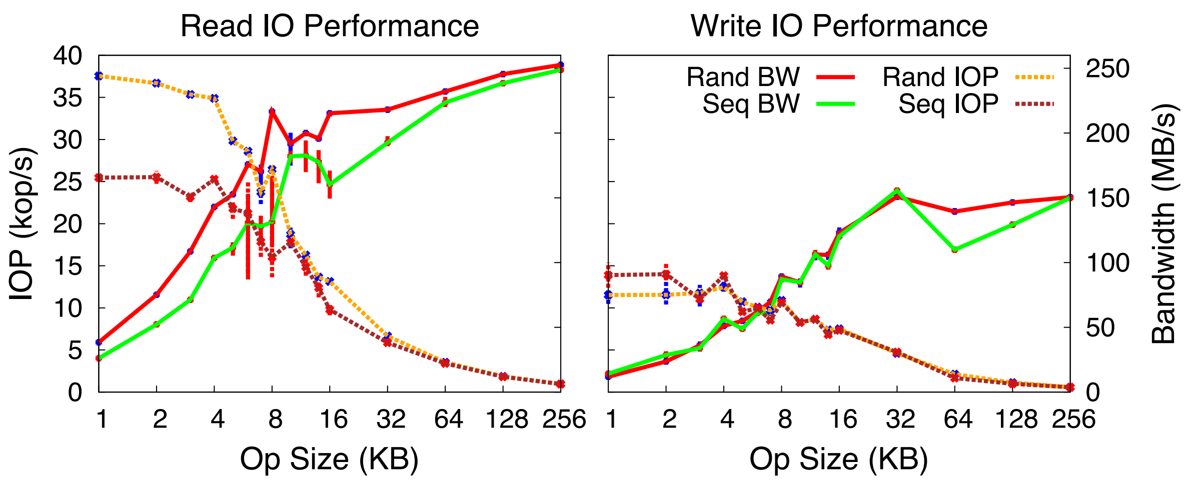


Figure 6.4: SSD IOPS and bandwidth vary non-linearly with IOP size. The measurements were made on an Intel 320 series SSD formatted with the ext4 filesystem.

lection to replenish its pool of pre-erased blocks. When write sizes are small and random, this process can incur a heavy read-merge-write penalty similar to LSM-Tree compaction to write out data in logically contiguous erase-block increments.

6.2.3 Non-linear IO performance

SSD throughput can be characterized along two basic axes: IO operation throughput (IOP/s) and data bandwidth. Although the latter is a function of the former (BW = IOP $\times$ IOP-size), both vary non-linearly with operation size due to shifting bottlenecks, as shown in Figure 6.4. IOP throughput peaks at small IOP sizes for both reads and writes, where the SSD is processor bound by its controller and on-die logic. Thereafter, IOP throughput decreases sub-linearly until the bandwidth bottleneck (i.e., SATA bus and SSD data channels) takes hold, around 64KB for reads and 32KB for writes. Furthermore, filesystem (ext4) overhead affects sequential IO performance more than random IO in these experiments. We can account for these anomalies in our IO cost model, since the Libra sits above the filesystem.

Accounting for only one IO bottleneck can leave the system underutilized. For example, in DynamoDB, one 100KB GET costs the same as one hundred 1KB GETs [11]. If we directly translate this pricing model to an IO cost model, then IOP cost would vary linearly with IOP size while bandwidth cost remains constant. However, at small IOP sizes, the IOP bottleneck throttles bandwidth far below the maximum, which means that the effective max throughput is limited by the worst case (small IOP size) workload. Hence, a 100KB GET costs more than it should since it is treated the same as one hundred IOP-limited 1KB GETs. On the other hand, if IOP cost is fixed, as in Amazon's provisioned EBS, then the resource model would have to constrain IOP throughput by the bandwidth bottleneck at large IOP sizes. Thus, if the IO resource model fails to account for both resource bottlenecks, IO throughput will be left fallow and underutilized.

6.3 Achieving Tenant Reservations

Libra addresses the challenges of IO amplification, interference, and non-linear performance by tracking IO cost, quantifying IO capacity, and modeling IO cost.

6.3.1 Determining application request cost

The key to determining the IO cost of an application request is to account for both its *direct* and *indirect* IO. In Libra, the persistence engine tags each IO task with its associated app-level request (e.g., GET or PUT) and internal persistence engine operation (e.g., FLUSH or COMPACT), when needed. Libra uses these tags to track both the direct tenant resource consumption (u_a^t for tenant t and app-request a) and the indirect resources consumed by internal operation i on behalf of a (u_i^t). These tags also indicate how often an app-request triggers an internal operation ($e_{a,i}^t$), which allows Libra to build a statistical model of resource usage that accounts for the amplified IO cost of a tenant's application request.

While the Libra scheduler updates the resource consumption counters on each IO task execution, the Libra resource policy periodically (once per second in our prototype) (re)computes the app-request resource profiles and (re)provisions tenant IO resource allocations accordingly. In each policy interval, Libra computes the per app-request and per operation resource costs q for each tenant as an exponentially-weighted moving average

(EWMA)

$$q_a^t = EWMA(u_a^t / s_a^t)$$
$$q_i^t = EWMA(u_i^t / s_i^t)$$

where s_a^t and s_i^t are the number of normalized (1KB) requests and internal operations executed over the interval, respectively. Using these base costs, Libra scales the internal operation cost by how often an app-request triggers the operation to compute the indirect resource cost $q_{a,i}^t$:

$$q_{a,i}^t = q_i^t \frac{e_{a,i}^t}{s_a^t}$$

Some internal operations, especially COMPACT, are triggered sporadically and may take many intervals to complete. To handle this case, Libra normalizes $q_{a,i}^t$ by the total number of requests executed since the last trigger, instead of just over the current interval, and attributes partial resource consumption for ongoing operations. Note that the resources costs only capture application-induced IO amplification. OS-level effects due to filesystem operations or SSD-internal read-modify-write operations are beyond Libra's reach and are rolled into the underlying IO cost model.

Combined together, these resource costs represent the full app-request resource profile. The resource policy uses the profiles to provision the necessary IO resource allocation r_a^t for each tenant's app-request reservation v_a^t:

$$\text{profile}_a^t = q_a^t + \sum_i q_{a,i}^t$$
$$r_a^t = v_a^t \cdot \text{profile}_a^t$$

By periodically recomputing the profiles and adjusting tenant IO allocations, Libra can adapt to shifts in tenant workload distributions and app-request IO cost. However, the ten-

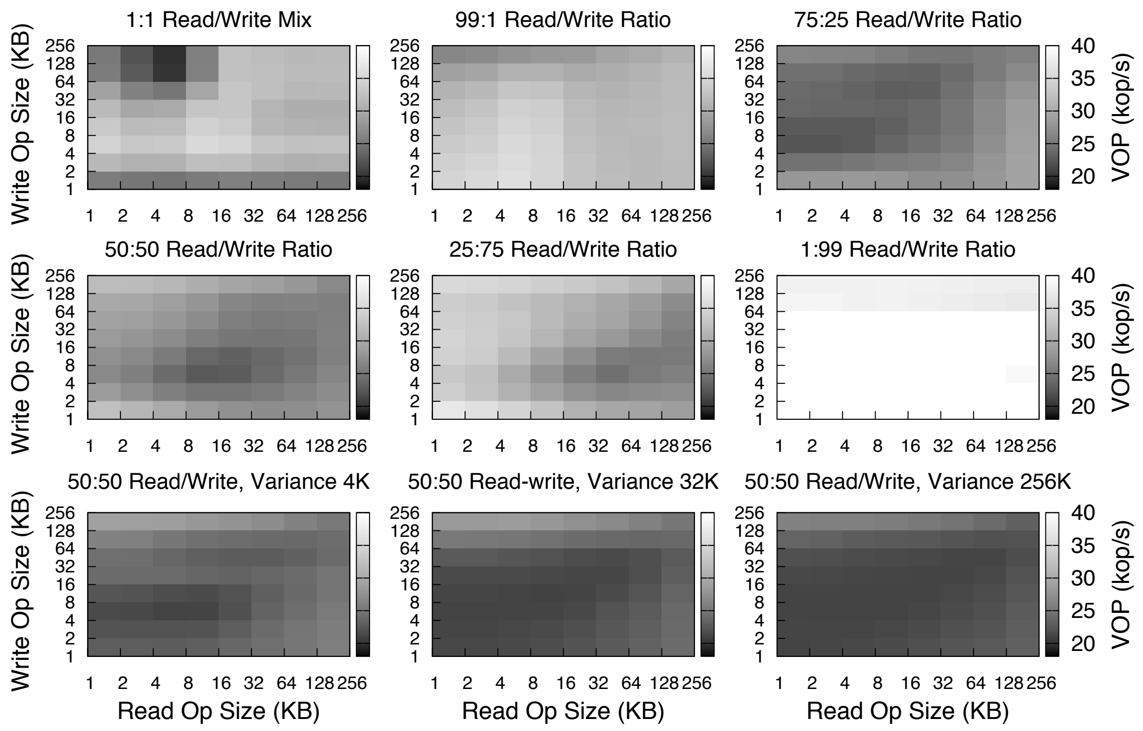


Figure 6.5: IO throughput varies unpredictably with IO interference. Throughput valleys (darkest regions) shift with read/write ratio and overall performance flattens out as IOP size variance increases. Each heat map shows experiments across a range of read (x-axis) and write (y-axis) IOP sizes from 1 KB to 256 KB.

ant allocations must not exceed the IO throughput capacity of the node. Normally, Libra's IO capacity model prevents overbooking by enforcing a lower limit on provisionable IO. However, under worst-case IO amplification and workload fluctuation, the storage node may not be able to satisfy the tenants' reservations. Under these overflow conditions, the resource policy scales down each tenant's resource allocation proportionally to fit within the capacity constraint. Libra then notifies Pisces's higher-level policies of the violation along with its current view of IO capacity and app-request resource profiles. If the storage node is underbooked, the scheduler shares any unallocated resources among the tenants proportionally.

6.3.2 Estimating IO throughput capacity

Libra requires a robust IO capacity model to ensure that its provisioned allocations can be met, even in the presence of IO interference. Since tenants can generate diverse IO workloads that may change over time, this estimate of *provisionable* IO resources should be a lower bound on the actual capacity to prevent overbooking and SLA violations. Ideally, IO interference would be both mild and predictable. This would allow Libra to model fluctuations with a simple, tractable capacity model that tightly bounds a smooth capacity curve. Unfortunately, we find that the effect of interference on IO capacity can be both severe and unpredictable.

To quantify the effects of IO interference in Libra, we ran a series of experiments exercising different read-write workloads over a range of IOP sizes. In each experiment, the 8 tenants issue low-level IO requests to the Libra scheduler according to two backlogged random-access workloads of the specified IOP sizes. In the 1:1 workload, half the tenants act exclusively as readers and the other half as writers. For mixed request-type workloads, each tenant issues both reads and writes according to the specified ratio. Lastly, in variable IOP-size workloads, each tenant samples IOP sizes from a log-normal distribution with the specified variance. Note that all tenants have an equal allocation of IO resources and equal demand specified by a bounded number of concurrent IO request workers. For the SSD in question (an Intel 320), the interference-free maximum IO throughput is 37.5 kop/s, measured in VOP/s. As described in §6.3.3, barring interference, fully backlogged tenants should achieve the full VOP/s throughput regardless of IOP size and operation type. Other SSDs (OCZ Vector and Samsung 840 Pro) behaved similarly.

Figure 6.5 shows that IO throughput is highly sensitive to the workload ratio of reads and writes. For exclusive reader-writer (1:1) and read-dominant (99:1) workloads, IO interference is relatively mild. IO throughput varies between 29 and 37 kop/s across most IOP sizes, with the exception of the small read, large write regime. Here, IO throughput drops below 19 kop/s due to the delay imposed by large write operations. As the ratio moves toward writes (75:25), however, IO throughput decreases dramatically. The throughput valley spreads out over the small read, medium write region. With the higher mix of writes, each tenant experiences increased IO interference from its own workload since its IO workers

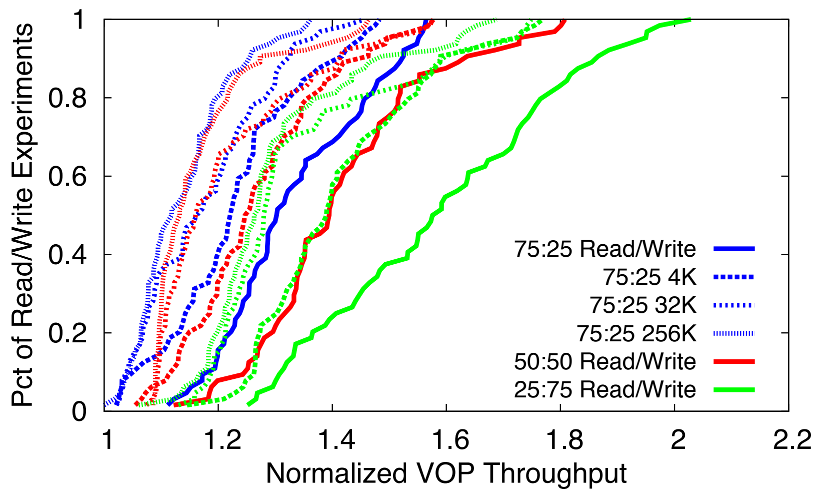


Figure 6.6: CDF of IO throughput for the IOP sizes shown in Figure 6.5. Solid lines correspond to workloads with uniform IOP sizes, while dashed and dotted line workloads issue variable request sizes with the indicated σ.

eventually bottleneck on the more expensive write operations, delaying the cheaper reads. At a 50:50 ratio, the throughput valley migrates to medium-sized reads and starts to shrink. Under write-heavy (25:75) workloads, IO throughput improves due to the more uniform workload of higher cost writes and fewer delayed reads. The valley moves further along the read axis to larger IOP sizes. Write-dominant (1:99) workloads experience little IO degradation except at large write sizes.

Randomizing IOP sizes—shown in the last row of Figure 6.5—consistently degrades IO throughput. The larger the IOP-size variance, the lower and flatter the IO throughput graph becomes. This is due to the increasing likelihood of sampling IOP sizes from high interference regions. To better illustrate this trend, Figure 6.6 replots the results as a CDF of IO throughput normalized by the minimum achieved throughput (~18kop/s). Experiments are sorted by their normalized throughput. For each read-write ratio, as IOP-size variance increases, IO throughput drops closer to the minimum value.

These experiments illustrate the highly unpredictable behavior of IO throughput under interfering workloads. Modeling throughput variation across all possible read/write ratios could be computationally expensive and susceptible to over-fitting, which could lead to overestimates of IO throughput. Libra takes a more robust and conservative approach by using the floor of the capacity curve—18 kop/s for this SSD configuration—to (under)estimate the provisionable IO capacity. The resource policy is free to provision tenant IO within this limit, but no more. While Libra could also monitor the current IO capacity to detect infeasible IO allocations, it uses the IO capacity threshold as a consistent bound for local admission control and to inform the placement and throughput distribution decisions handled by Pisces's higher-level policies.

Under light IO interference this approach may leave up to half of the maximum IO capacity (37.5 kop/s) unavailable for provisioning. However, it is the safer option for complex persistence engines that continually generate secondary read and write IO operations of variable size and for realistic key-value workloads that issue variable size requests. For 80% of the low-variance (4K) workloads in Figure 6.6, at most one third of IO throughput is left unprovisioned, but still usable since Libra is work-conserving.

6.3.3 Defining an IO metric and cost model

The IO throughput metric and its associated cost model are essential for resource accounting, provisioning, and capacity estimation. Any viable metric should fulfill three key criteria: (i) present a unified view of IO throughput (ii) capture the inherent non-linearity of the IO performance curves and (iii) provide an intuitive measure of capacity and cost. Given these criteria, neither disk bandwidth nor raw IOP throughput alone are suitable metrics. Recently, allocation schemes like dominant resource fairness [25] attempt to share multiple *independent* resources between its resource principals. However, since all SSD operations share a common SATA bus, internal data channel, and controller, IOP throughput and IO bandwidth are highly correlated across read and write operations and should be represented by a unified metric.

Existing IO schedulers have generally taken one of two time-based approaches: explicit time-slicing [73, 54, 13] or virtual-time fair queuing [30, 59]. Because time-slicing allots

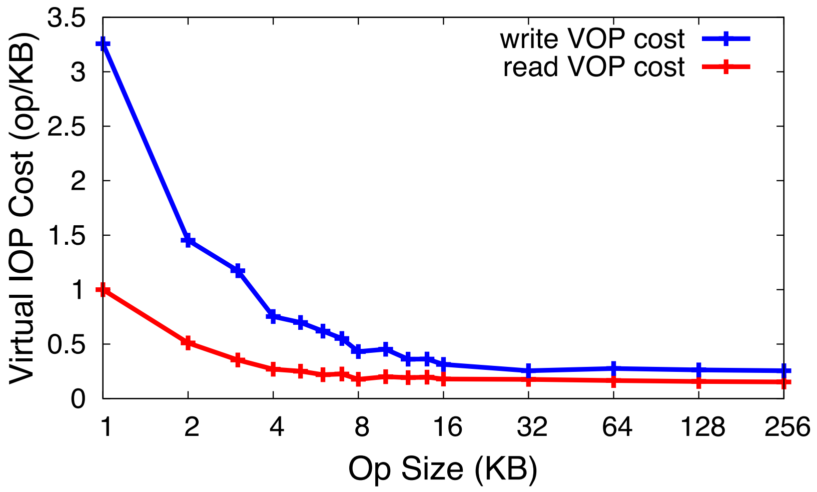


Figure 6.7: Libra IO cost model.

exclusive disk access during a time slice it can lead to wasted IO throughput and unnecessary delay. If a tenant has less work in its queue than its current time-slice allows, the remainder of the slice goes unused. Virtual-time fair queuing, as described in Section 2.2.2, maintains high utilization by scheduling IO requests according to virtual start or finish time without introducing any unnecessary gaps. However, scheduling overhead for virtual-time fair-queuing can be high (log of the number of requests) compared to constant-time round-robin scheduling.

In contrast, Libra represents IO throughput directly in terms of an IO rate with the virtual IOP (VOP). VOPs not only unify the non-linear bandwidth and IOP performance curves into a single resource, but also serve as the currency of IO capacity and cost. Underlying the effectiveness of the VOP is an IO cost model that captures the non-linear dependence of IO performance on IO operation size. While the VOP is a close analog of virtual-time from a fair-queuing scheduling perspective, it allows for more efficient round robin scheduling and provides an arguably more intuitive notion of IO performance. Libra uses the VOP resource model to schedule tenant IO, build app-request resource profiles, provision resource allocations, and estimate provisionable IO capacity.

The virtual IOP can be thought of as a size-normalized, variable-cost IOP. Where typical IOP scheduling treats each (normalized) IOP as equal cost, Libra charges each IOP according to a non-linear cost model derived directly from the IOP throughput curves, as shown in Figure 6.7. Libra calculates the cost model in terms of VOPs-per-byte by dividing the max IOP throughput by the achieved (read or write) IOP throughput, normalized by IOP size.

$$\text{VOP}_{\text{CPB}}(\text{IOP-size}) = \frac{\text{Max-IOP}}{\text{Achieved-IOP}(\text{IOP-size}) \times \text{IOP-size}}$$

In this model, the maximum IO throughput in VOP/s is constant for the pure read/write throughput curves.

As in DFQ, Libra's scheduler threads perform distributed deficit round robin [38] (DDRR) to efficiently schedule parallel IO requests (up to 32, which corresponds to the SSD queue depth). For each IO operation, the scheduler computes the number of VOPs consumed

$$\text{VOP}_{\text{cost}}(\text{IOP-size}) = \text{VOP}_{\text{CPB}}(\text{IOP-size}) \times \text{IOP-size}$$

and deducts this amount from the associated tenant's VOP allocation to enforce resource limits and track resource consumption. DDRR incurs minimal inter-thread synchronization and schedules IO tasks in constant time to efficiently achieve fair sharing and isolation in a work-conserving fashion. In general, virtual-time [21] and round-robin [62] based generalized processor sharing approximations are susceptible to IO throughput fluctuations, since they only provide proportional (not absolute) resource shares. However, Libra's IO capacity threshold ensures that each tenant receives at least its allocated share. Any excess capacity consumed by a tenant can be charged as overage or used by best-effort tenants.

The VOP cost model allows Libra to charge an IO operation in proportion to its actual resource usage. For example, 10000 1KB reads, 3000 1KB writes and 160 256KB reads all represent about a quarter of the SSD IO throughput at their respective IOP sizes. Hence, Libra charges each workload the same 10000 VOP/s, or about one quarter of the max IOP capacity. Thus, barring interference effects, Libra can divide the full IO throughput arbitrarily among tenant workloads with disparate IOP sizes. Note that while the shape of the read and write VOP cost curves are similar, their magnitudes reflect the relative cost of their operations. Writes are always more expensive than reads, but the gap diminishes as IOP sizes increase, due to lower erase block compaction overhead.

Determining the VOP cost model and minimum IO capacity for a particular SSD configuration requires benchmarking the storage system using a set of experiments similar to the ones described for the throughput curves. While these experiments are by no means exhaustive, they probe a wide range of operating parameters and give a strong indication of SSD performance and the minimum VOP bound. Pathological cases where IO throughput drops below the minimum can be detected by Libra, but should be resolved by higher-level mechanisms. Assuming timely resolution, these minor throughput violations can be absorbed by the provider's SLA, e.g., EBS guarantees 90% of the provisioned throughput over 99.9% of the year [7].

6.4 Implementation

Libra exposes a posix-compliant IO interface to the application, wrapping the underlying IO system calls to enforce resource constraints and interpose scheduling decisions, as shown in Table 6.1. To utilize Libra resource scheduling, applications, i.e., persistence engines, simply replace their existing IO system calls (read, write, send, recv, etc) with their corresponding wrappers. Libra also provides a task marking API for applications to tag a thread of execution (task) and its associated IO calls with the current app-request or internal operation context, e.g., `task_get_current` and `task_set_request_type`. An admin API manages resource objects, e.g., tenants, resources, request types, etc, and gives the application the flexibility to dynamically add, remove, and modify Libra objects

Libra IO API	
`open_ra(fd,...)`	`close_ra(fd,...)`
`read_ra(fd,...)`	`write_ra(fd,...)`
`socket_ra(fd, ...)`	`accept_ra(fd, ...)`
`send_ra(fd, ...)`	`recv_ra(fd, ...)`
Libra Task API	
`task_create(resource,tenant)`	`task_set_process_op(task_op)`
`task_get_current()`	`task_set_request_type(task, request_type)`
`task_reactivate(task)`	`task_trigger_request(task)`
Libra Admin API	
`init_ra()`	`fini_ra()`
`init_request_types(config)`	`init_resource_policy(config)`
`init_system_resources(config)`	`init_tenants(config)`

Table 6.1: Key functions in the Libra API. Libra wraps IO system calls with Libra stubs to for resource scheduling and accounting. Application can directly submit IO tasks to Libra via the task interface. The admin API manages Libra objects and performance.

or initialize them from a pre-defined config, e.g., `init_system_resources`. The Libra IO scheduling framework is implemented as a user-space library in ~20000 lines of C code.

IO processing in Libra can operate in one of two general modes. The first mode converts application IO operations into fine-grained IO tasks and submits them to the Libra scheduler. While simple to use and transparent to the application, it can incur high overhead since the application thread must block until notified by a scheduler thread on task completion. The second mode uses more coarse-grained tasks that intermingle data processing with IO operations. Here, applications explicitly create `task` objects (`task_create`), supply a `task_op` handler function (`task_set_process_op`), and submit them to the Libra scheduler (`task_reactivate`). Each `task_op` typically corresponds to a high-level operation, e.g., request protocol parsing, persistence engine GET or PUT request, or background FLUSH. In this mode, the Libra scheduler pool effectively replaces the application's threading model, i.e., thread pool workers, as in user thread packages like Capriccio and Lithe [72, 52]. This drastically reduces inter-thread signaling and scheduling overhead compared to the fine-grained task model, provided that the data processing code in the `task_op` is relatively short and efficient compared to the IO operations.

Libra employs coroutines to handle blocking disk IO and inter-task coordination, i.e., mutexes and conditionals. Coroutines allow Libra to pause a tenant's task execution by swapping out processor state, i.e., registers and stack pointer, to a compact data structure and resume from a different coroutine. Libra uses this facility to reschedule an IO task on resource exhaustion or mutex lock. This allows Libra to delay IO operations that would otherwise exceed a tenant's resource allocation until a subsequent scheduling round when the tenants resources have been renewed. Libra defaults to synchronous disk operations (O_SYNC) and disables all disk page caching (i.e., O_DIRECT on linux). The page cache masks IO latency and queue back pressure, which undermines Libra's scheduling decisions and capacity model. We also run Libra in tandem with a noop kernel IO scheduler to force IOPs to disk with minimal delay and interruption.

Enabling Libra in our LevelDB-based storage prototype required less than 30 lines of code. Since LevelDB abstracts interaction with the operating system behind a set of file Read/Write interfaces, it was easy to just replace the existing system calls with their Libra counterparts. However, unlocking LevelDB's full performance for synchronous PUTs required extensive modifications. Our prototype enables parallel writes, similar to HyperLevelDB [71], to take full advantage of SSD IO parallelism (LevelDB serializes all client write threads by default). Our prototype also issues sequential writes (e.g., indexed data file flush) in an asynchronous, io-efficient manner (LevelDB defaults to memory mapped IO which is incompatible with O_DIRECT). Lastly, our prototype runs FLUSH and COMPACT operations in parallel (LevelDB schedules both in the same background task). To minimize the effect of background COMPACT on foreground resource consumption, Libra limits COMPACT to use at most 10% of a tenant's VOP resources.

We decided to implement Libra as a user-space library for two main reasons. First, as described in Section 3.2, the storage node multi-tenancy model we support is a single server process with multiple threads that handle requests from any tenant, frequently switching from one tenant to another. This model is commonly used by high-performance storage servers [20, 12]. Existing kernel mechanisms for multi-tenant resource allocation, (e.g., linux cgroups [41]), work well for the process-per-tenant model where tasks (processes or threads) are bound to a single cgroup (tenant) over their lifetime. Frequent switching between cgroups, however, is slow due to high lock contention overhead in the kernel. Second, tracking app-request resource consumption across system call boundaries requires additional OS support for per-IO request tagging (as described in [43]), which is beyond the scope of this work. Conceptually, the VOP resource model should work in the OS scheduler, but we leave an in-kernel implementation to future work.

6.5 Related Work

Predictable cloud storage. Most work on provisioning IO for multi-tenant shared storage have either focused on achieving tenant service-level objectives (SLO) [42, 74], or providing proportional shares [28, 54, 59, 64]. Maestro [42] controls IO port queue depth for local clients accessing a disk array to achieve tenant-specified throughput and latency SLOs. To handle IO interference, Maestro uses an adaptive feedback model to estimate the port queue depths needed to meet tenant requirements. Unlike Libra however, Maestro does not differentiate IOPs by size, is not explicitly work-conserving, and only supports a basic block storage API.

Cake [74] also targets tenant SLOs, but for latency-sensitive tenants accessing a two-tier storage system alongside batch-workload tenants. Like Maestro, Cake actively monitors request latency and allocates resources (request handler threads) to meet tenant-specified SLO latencies and ensure performance isolation. However, it does not support explicit per-tenant app-request reservations. mClock [30] and Parda [28] provide per-VM hypervisor IO resource allocation by applying limits and reservations to virtual-time IO scheduling and client-based IO congestion control respectively. Although Mclock supports IO-level reservations, it uses a non-optimal linear IO cost model. Neither approach addresses application-level request reservations.

All the above mentioned systems were designed for HDD-based systems with much lower IOP throughput performance. Although the adaptive feedback model works well for the longer time horizons afforded by HDD performance, it may not be able to keep pace with the low-latency and high throughput of SSDs. Tuning the feedback loop to react to IO interference and resource imbalances on a sub-second level may be prohibitively expensive. In contrast, Libra's scheduler mediates IO resource consumption in real time to enforce tenant allocations within interference thresholds, while its resource policy adaptively reprovisions allocations at a longer time timescale to handle workload-induced IO amplification.

Disk IO scheduling. The literature is replete with work on fair IO resource scheduling ranging from the network [21, 62, 27, 68], to CPU [16, 38], to storage [22, 29, 30, 73]. Most relevant to our work are the FIOS [54] and FlashFQ [59] IO schedulers for SSD storage. FIOS was one of the first to address fair resource sharing on SSDs. It highlighted the write interference behavior peculiar to SSDs and how the coarse-grained delays and optimizations built into HDD IO schedulers have adverse affects on SSD performance. FIOS adapts the traditional time-quanta based approach [73, 13], to SSD scheduling by incorporating request parallelism, read prioritization, and judicious use of IO delay to combat deceptive idleness. However, as with any time-slicing approach, FIOS trades off high utilization for better IO insulation and may induce unnecessary delay.

FlashFQ employs virtual time (VT) to improve responsiveness. It issues parallel requests for high throughput while preserving fair-shares by throttling aggressive tenants via virtual-time delays. Virtual time, like the virtual IOP, presents a consistent measure of resource capacity for fair-queuing IO schedulers. As such, it also depends on having an accurate IO cost model to be used effectively. Although FlashFQ achieves fair shares and high utilization, its linear cost model does not account for the full range of SSD IO performance, leading to less than ideal IO throughput as shown in our evaluation. VT-based schedulers also incur higher scheduling overhead than the round-robin scheduler used in Libra. Lastly, we chose to use virtual IOPs as our IO metric because it is, arguably, more intuitive a measure of IO throughput and capacity than virtual time since it is a rate that closely models IOP cost.

Chapter 7

Libra Evaluation

In this evaluation, we examine how well Libra addresses IO amplification, interference, and non-linear performance from the bottom up to answer the following questions.

- Does Libra's IO resource model capture SSD performance and enable accurate resource allocations?
- Does Libra's IO threshold make an acceptable tradeoff of performance for predictability in a real storage stack?
- Can Libra ensure per-tenant app-request reservations while achieving high utilization?

We start with the IO resource model to establish a basis for accurate resource accounting and allocation. Then we examine the IO capacity threshold to confirm its viability for provisioning IO resources under key-value workloads. Lastly, we evaluate Libra's ability to achieve tenant app-request reservations with all mechanisms in place.

7.1 Experimental Setup

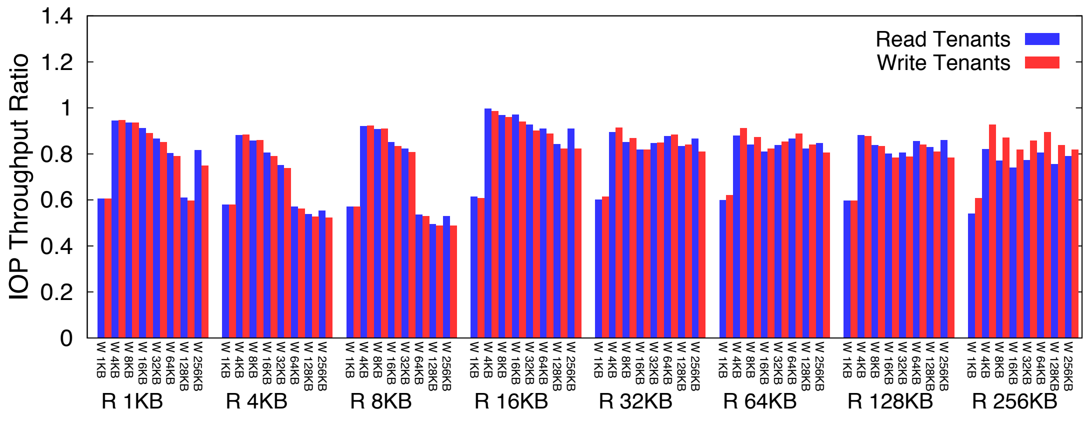
We ran our experiments on two separate configurations using three different SSDs. The lower spec machine runs Ubuntu 11.04 on two 2.4 GHz Intel E5620 quad-core CPUs, 12GB of memory, and a SATA II 160 GB Intel 320 series SSD. The higher spec machine runs Ubuntu 12.04.2 LTS on two 3.07 GHz Intel X5675 hexa-core CPUs, 48 GB of memory, and two SATA III SSDs: a 256 GB Samsung 840 Pro and a 256 GB OCZ Vector. All SSDs are ext4 formatted. The three SSDs allow us to evaluate the Libra over a range of controller architectures (Intel, Samsung, and Indilinx) and bandwidths (3 Gbps for SATA II and 6 Gbps for SATA III). All experiments run with a queue depth of 32 and backlogged demand since Libra is designed for high-utilization environments.

7.2 Libra Achieves Accurate IO Allocations

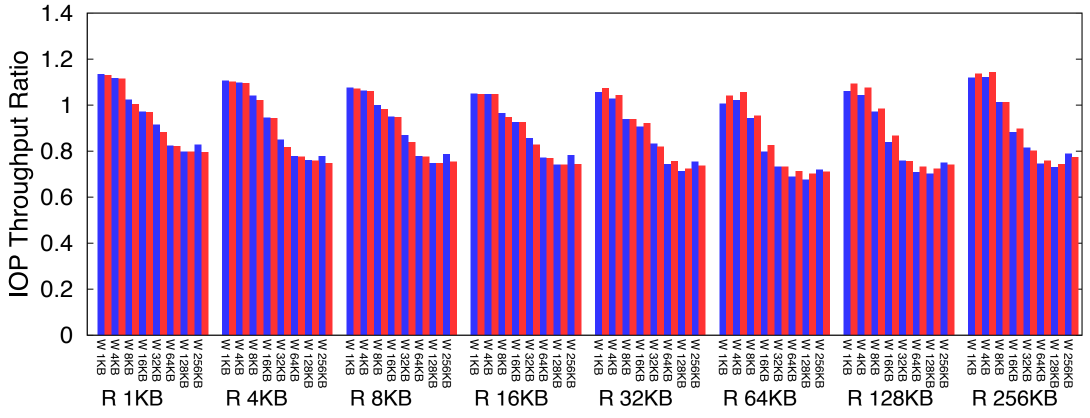
To measure the accuracy of resource allocation, we use the IO throughput ratio x^t of achieved throughput over expected, as well as the Min-Max Ratio (MMR) of x^t over all tenants t:

$$x^t = \frac{r^t_{\text{achieved}}}{r^t_{\text{allocated}}} \quad ; \quad \text{MMR} = \frac{\min_t(x^t)}{\max_t(x^t)}$$

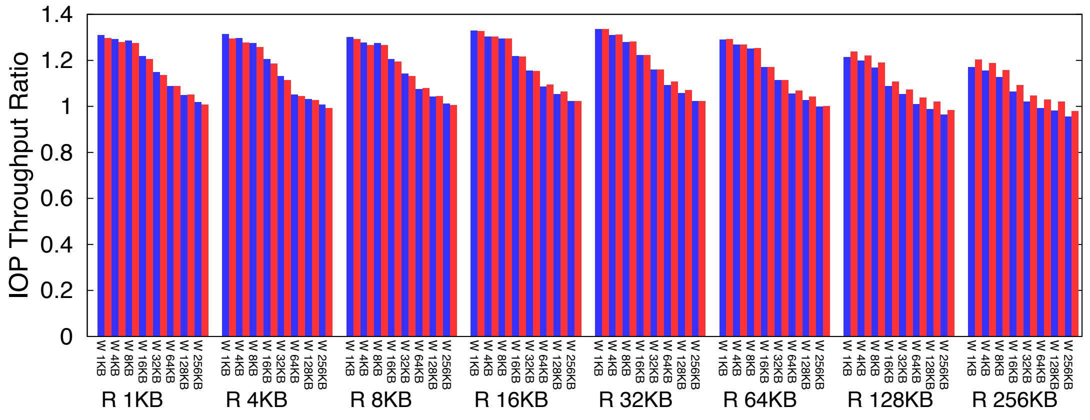
Here, expected throughput corresponds to resource proportion. If a tenant's allocation commands half the (interference-free) IO resources, then the expected IO throughput (in bandwidth or IOP/s) should be half of what the tenant's workload would achieve in isolation. Thus, a throughput ratio of 1 indicates perfect IO *insulation* [73]. In the same way, a virtual IOP allocation should yield a proportional share of the constant max VOP/s capacity regardless of the workload. Under IO interference, each tenant's throughput ratio should drop in proportion to the decrease in IO performance. For equal VOP allocations, this means the scheduler should be perfectly fair and penalize all tenants equally (MMR = 1). To evaluate the viability and accuracy of the Libra's IO resource model, we examine the IOP throughput ratio for a set of 8 tenants(half readers and half writers) with equal VOP allocations issuing IO requests over a range of IOP sizes.



(a) Intel SSD IO Throughput Ratio



(b) Samsung SSD IO Throughput Ratio



(c) OCZ SSD IO Throughput Ratio

Figure 7.1: The Libra VOP resource model achieves near perfect (equal) IOP throughput ratios between 4 read and 4 write tenants on different SSD architectures, even in the presence of IO interference (ratio < 1). Read IOP size is fixed for each experiment in a cluster, while write IOP size increases from 1 KB to 256 KB, on a log scale. Each read/write tenant *pair* within a cluster represents a single experiment at a specific read and write IOP size.

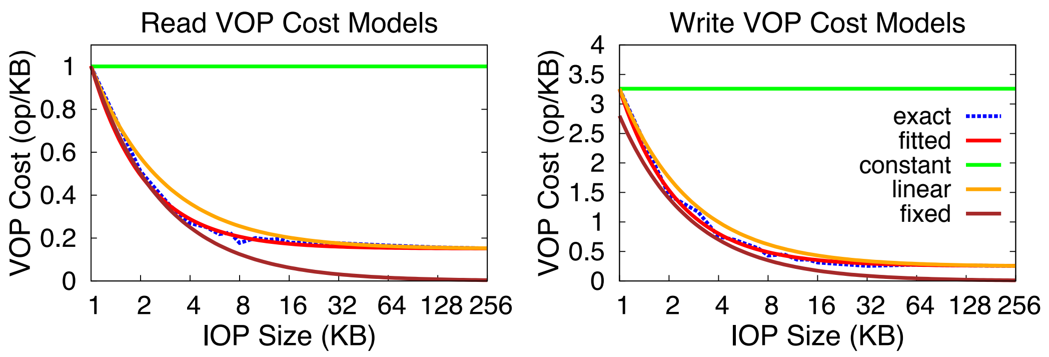
7.2.1 Virtual IOP allocation achieves physical IO insulation.

Figure 7.1 shows the IOP throughput ratios achieved by Libra on the three different SSDs. Libra achieves near perfect insulation for all tenants—mean 0.98 tenant throughput MMR averaged over all IOP sizes—on all SSDs, even when throughput fluctuates with IO interference. The only significant deviation in throughput ratio occurs on the Intel SSD at large read IOP sizes, in which the scheduler's chunking scheme breaks up large IOPs (> 128 KB) into smaller operations as a trade-off for better responsiveness. Using its IO cost model, Libra is able to share the available physical IO according to the tenants virtual IOP allocations and distribute the effect of IO interference, penalizing all tenants equally regardless of their workload and IOP size.

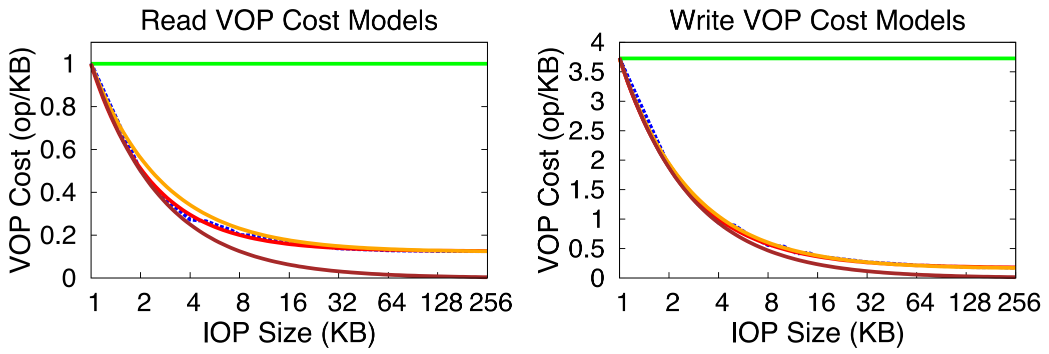
Figure 7.1 also illustrates how IO interference varies across the different SSD architectures. The Intel SSD retains ~80% of its throughput under most conditions, except for small reads coupled with large writes. Both the Samsung and OCZ SSDs exhibit greater interference for large writes, regardless of read size. The OCZ SSD, however, is better able to parallelize the multi-tenant IO workload compared to the single-tenant case, yielding throughput ratios > 1.

7.2.2 The Libra IO cost model achieves the best physical IO allocation.

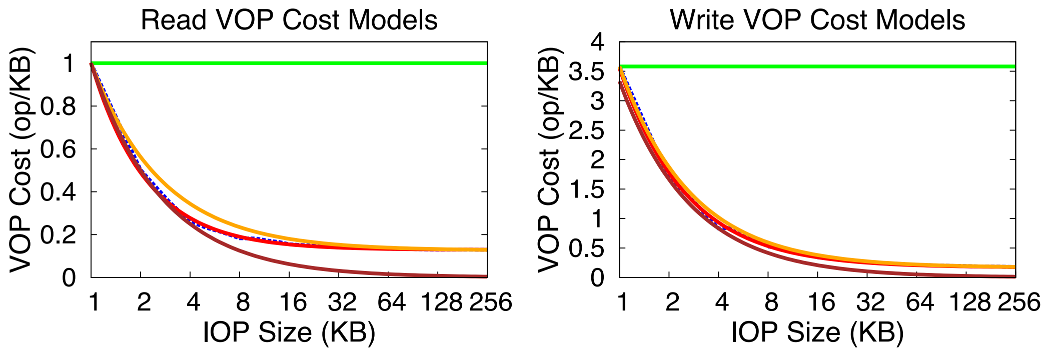
In these next experiments, we compare Libra's IO cost model against alternative cost models. To emulate DynamoDB's approach, where 100 1KB requests equals one 100KB request, we use a *constant* VOP cost-per-byte model. Several virtual-time-based IO schedulers [30, 59] estimate IO cost using a *linear* cost model with non-zero intercept. Lastly, we also model a *fixed* IOP cost scheme that charges all IOPs the same, regardless of IOP size. Figure 7.2 plots the different VOP cost curves, including the curve-*fitted* Libra cost model, for the three SSDs. The VOP *cost per byte* for the Intel SSD, shown in Figure 7.2a, deviates significantly from the linear model for both reads and writes with cost per byte IOP size exponents of -1.22 and -1.31 respectively, versus -1 for the linear model. The



(a) Intel SSD IO Cost Model



(b) Samsung SSD Cost Model

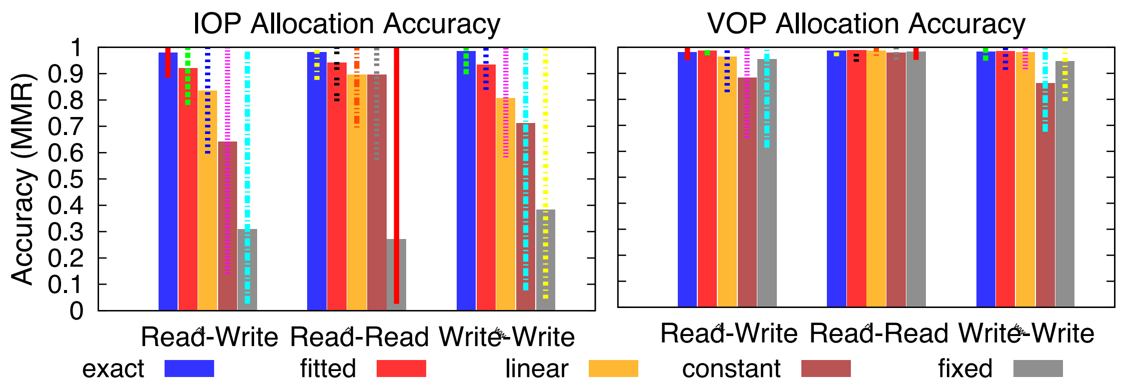


(c) OCZ SSD Cost Model

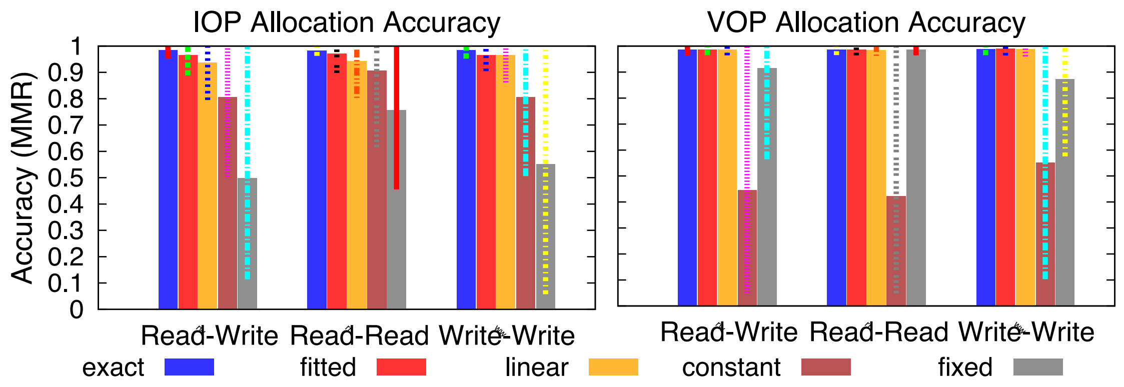
Figure 7.2: Virtual IOP cost models. The Intel SSD exhibits the most non-linearity, while both the Samsung and OCZ SSDs have less pronounced read non-linearity and almost linear write VOP cost per byte.

VOP cost curves for the Samsung and OCZ SSDs, shown in Figures 7.2b and 7.2c, exhibit less curvature, especially for write VOP costs. The two disks have read exponents of -1.17 and -1.27 and write exponents of -1.04 and -1.07 respectively. Compared to the Libra VOP cost model, the constant model charges a much higher cost-per-byte, while the fixed models undercut the VOP curve for both reads and writes. The linear model hews much closer to the VOP curve, but still deviates significantly for small read IOPs on all disks, and small write IOPs on the Intel SSD.

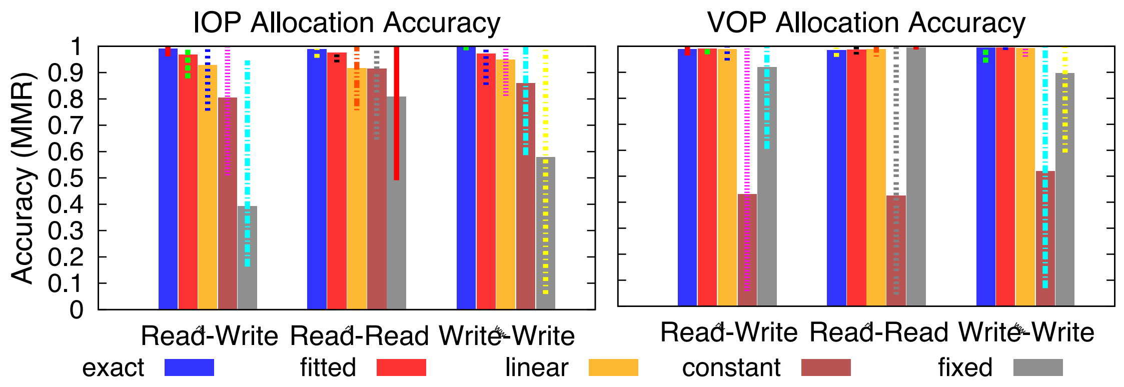
Figure 7.3 shows the median throughput ratio MMR achieved by the different IO cost models for the 8 tenants over 64 trials using the same range of IOP sizes from the previous experiments. For the Intel SSD, only the exact and fitted VOP cost models are able to achieve a median throughput ratio MMR greater than 0.9 under all workload conditions; their difference is due to the fitted model's approximation error. Among the non-Libra approaches, the linear model fairs best with a 0.83 median MMR, since it hews closely to the exact model near the interpolation end points (i.e., for small and large IOPs). However, between the end points, the VOP cost deviation results in skewed allocations and lower MMR. Despite egregiously over-charging IOP sizes greater than 1 KB, the constant model is able to maintain a rough balance ($>$ 0.5 MMR) between the read and write workloads because it over-charges them equally. In the fixed model, IO cost decreases so quickly that tenants with larger IOP sizes ($>$ 16 KB) are able to execute more IO requests than they should, which leads to skewed throughput. Results for the Samsung and OCZ SSDs are similar, though the linear model is able to achieve better allocations with median MMR close to 0.92 for Read-Write workloads. As shown in Figure 7.2, most of the gain in accuracy for the linear model comes from VOP write costs. However, minimum MMR for the linear model, under 0.8 MMR, still falls short of the Libra cost model which is close to 0.9 MMR for both SSDs.



(a) Intel SSD Allocation Accuracy



(b) Samsung SSD Allocation Accuracy



(c) OCZ SSD Allocation Accuracy

Figure 7.3: Libra achieves the most accurate physical and virtual IO resource allocation. Each bar summarizes results from the set of experiments in Figure 7.1 using the different cost models.

7.2.3 The Libra scheduler achieves accurate VOP allocations.

Using virtual IOPs, the Libra scheduler is able to enforce physical IO insulation between tenants given their VOP allocations. Enforcing accurate VOP allocations, in turn, ensures that Libra can deliver on its provisioned IO resource allocations for achieving app-level request reservations. Ideally, Libra's scheduler should provide accurate VOP allocations regardless of cost model. As the right hand graphs in Figure 7.3 show, the Libra's fitted and exact IO cost models also achieve the most accurate VOP allocations across all workloads and IOP sizes with median MMR > 0.98 and min MMR no lower than 0.91. Both the linear and fixed models also achieve accurate allocations (median MMR > 0.94). The constant cost model trails behind (median MMR < 0.9) due to its gross underestimate of large IOP cost, which allows those tenants to over-consume physical IO. Conversely, as the large IOPs take much longer to complete, they trigger timeouts that prematurely advance the scheduling round, which results in VOP under-consumption. The Samsung and OCZ SSDs exhibit similar behavior, though the constant model suffers even more. These results confirm that poor IO throughput insulation is due to IO cost model inaccuracies and not faulty scheduler VOP accounting.

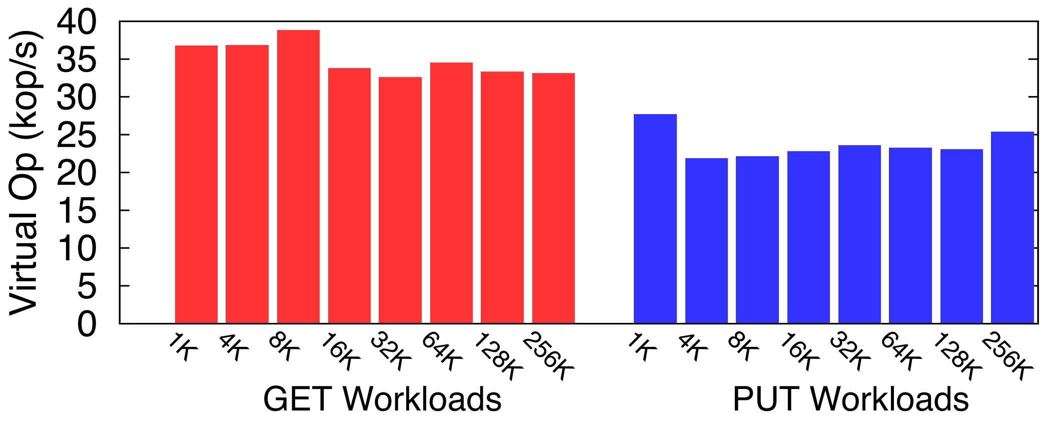
7.3 Libra Trades Nominal Performance For Provisionable VOP Capacity

Per Section 6.3.2, IO interference can be both severe and unpredictable for mixed tenant workloads. In light of this workload-dependent variability, Libra sets the provisionable IO capacity to a VOP threshold of 18 kop/s to ensure that tenant app-request allocations can be always be provisioned, no matter how wildly VOP capacity may vary. To understand whether this "floor" is a viable underestimate of provisionable capacity in a real storage stack we examined the IO throughput (in VOP/s) of our LevelDB-based prototype over a range of app-request workloads. Recall that LevelDB [37] is an LSM-tree-based key-value store designed to support write-heavy workloads. All experiments use the Intel SSD to probe the lower bound of performance; we report results after reaching steady state for background FLUSH and COMPACT operations.

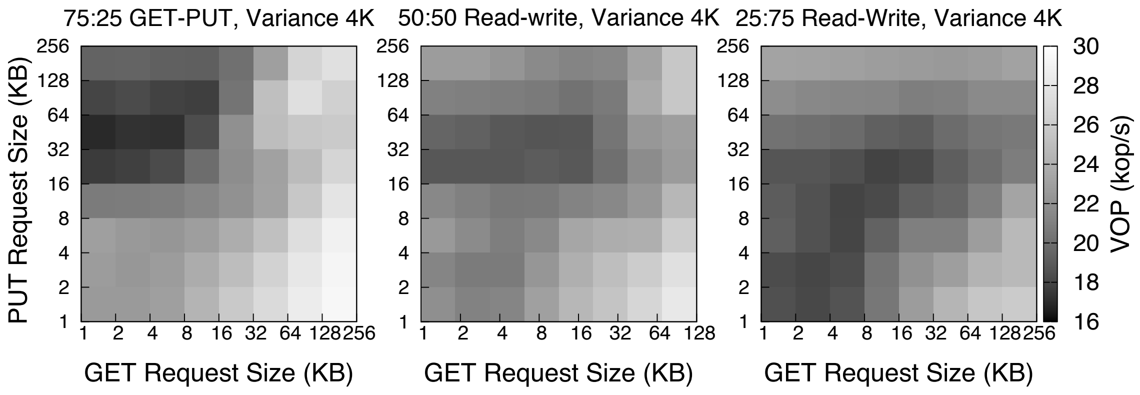
As a baseline, we ran pure GET and PUT workloads over the 1 to 256 KB request size range shown in Figure 7.4a. For GET workloads, which exclusively generates read IOPs, our prototype is able to achieve close to the maximum VOP/s (38 kop/s) . PUT workloads on the other hand, produce write operations of varying sizes from secondary FLUSH and COMPACT operations which issue reads as well. This interference causes throughput to drop down to 21.8 kop/s.

Mixing GET and PUT requests with varying ratios results in wide-ranging VOP throughput, as shown in Figure 7.4b. In these experiments, each tenant issues a mix of GET and PUT requests at the specified ratio. Tenants draw request sizes from a log-normal distribution with the indicated mean and a variance of 4K. For most request sizes, across all mix ratios, VOP throughput rarely exceeds 24 kop/s and overall throughput degrades as the ratio becomes more PUT heavy, with a minimum just above 16 kop/s. As in the case for raw disk IO, the shape of the IO throughput also varies as ratio shifts from GET to PUT, with the throughput valley (dark regions) both expanding its boundaries and shifting its epicenter. This suggests that the simple floor capacity model remains the most viable approach to (under) estimating the provisionable IO throughput.

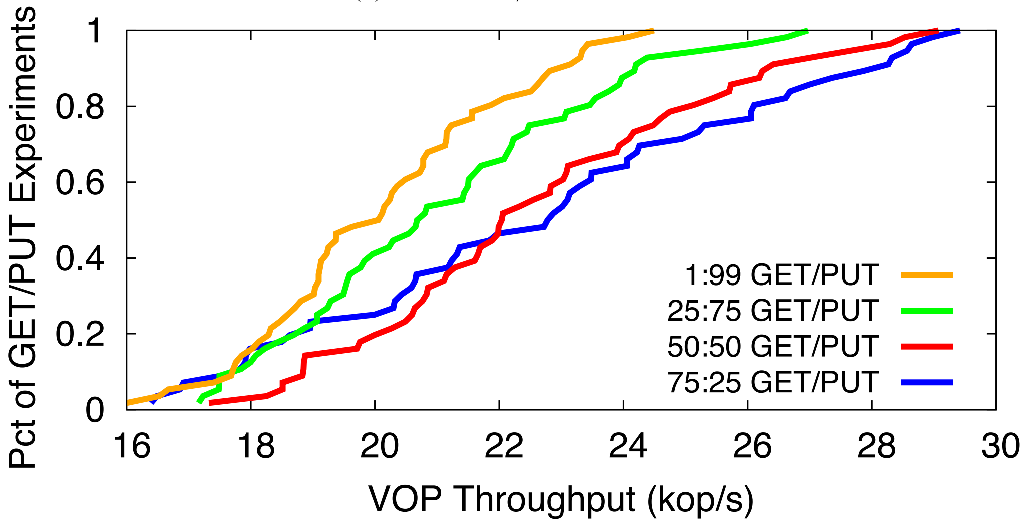
To understand how much IO throughput Libra leaves unprovisioned using the VOP floor, we examine the CDF of IO throughput for the workloads shown in Figure 7.4c. Here, the trend towards lower throughput at higher PUT ratios can be clearly seen, with 80% of the PUT-dominant (1:99) trials achieving less than 22 kop/s. Over all workload ratios, 80% of the trials achieve at most 26 kop/s, which leaves just over 40% beyond the reach of the VOP floor (18 kop/s). If we look at the median, this unprovisionable excess—which remains usable, just not provisionable in Libra—falls to about 30%. The few cases where VOP throughput may fall below the floor can be absorbed by the storage SLA in the



(a) Pure GET/PUT Workloads



(b) Mixed GET/PUT Workloads



(c) CDF of GET/PUT VOP Throughput

Figure 7.4: Libra's VOP threshold (18 kop/s) ensures at least 60% of the achievable throughput is provisionable for 80% of the workloads.

short-term and ultimately resolved by higher-level mechanisms (i.e., partition migration or redistributing local reservations).

Of course, not all workload ratios and request sizes are equally likely in practice. For a storage stack with an in-memory, write-through object cache, we expect most workloads that touch the SSD to be PUT-heavy. In most key-value storage use-cases, object value sizes are relatively small ($<$ 32 KB), corresponding to the low VOP throughput region of the PUT-heavy workloads (25:75) and (1:99), These regions account for the bottom 25% of their CDF curves which fall under 19 kop/s. Here, the VOP floor underestimates the achievable IO throughput by at most 15%. Note that workloads with highly variable request sizes diminish IO throughput and further reduce the unprovisionable excess.

7.4 Libra Realizes App-request Reservations

Achieving predictable application performance hinges on Libra's ability to provision local tenant app-request reservations. Libra builds app-request IO resource profiles using the VOP resource model to determine the IO resource allocations needed to satisfy app-level throughput. To evaluate Libra's ability to provision app-request reservations, we ran a series of experiments with 8 tenants exercising a range of different workloads and under dynamic conditions. Three read-heavy tenants issue a 90:10 workload of small requests (with mean 4KB GETs and 16KB PUTs). Two mixed 50:50 tenants generate moderate requests (64KB GETS, 16KB PUTs). Lastly, three write-heavy 10:90 tenants send large requests (128KB GETs and PUTs). All request sizes are sampled from a log-normal distribution, with the specified means and σ = 1KB.

7.4.1 Tracking resource profiles enables accurate app-request provisioning.

Figure 7.5 shows the results at steady state throughput both with (top) and without (bottom) tracking app-request resource profiles. Initially, all tenants reserve GET and PUT request

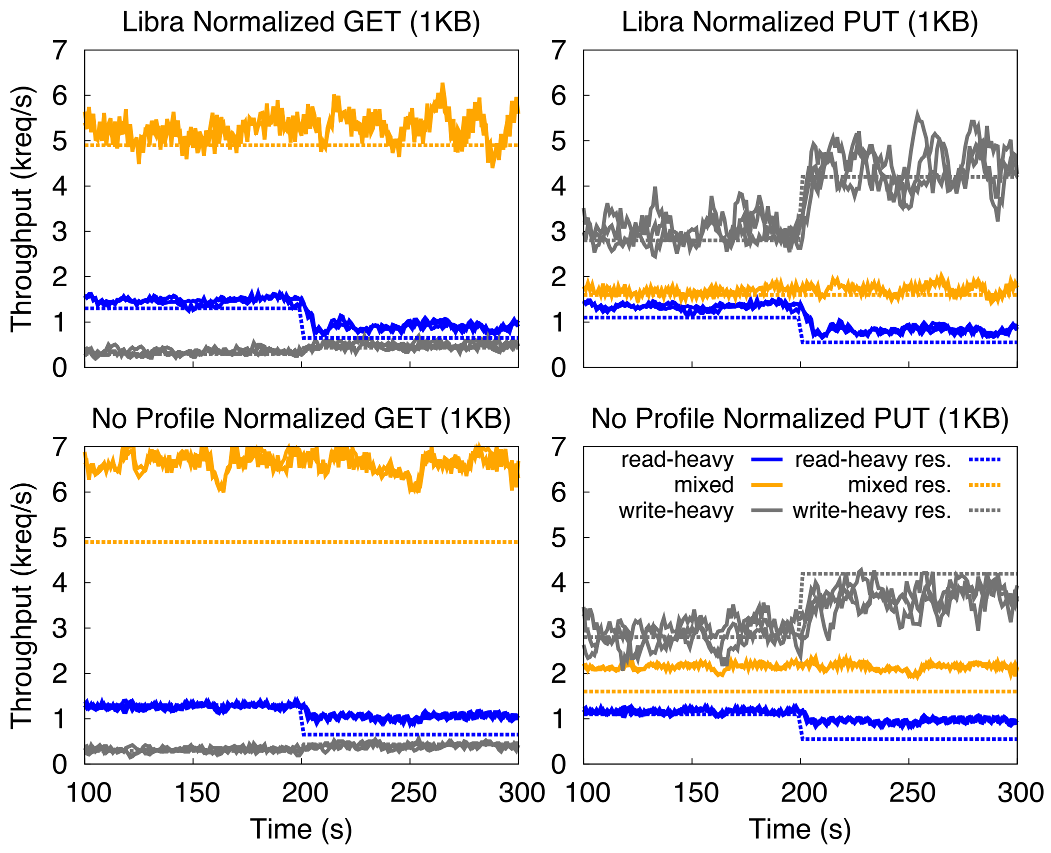


Figure 7.5: With app-request profile tracking, Libra achieves (dynamic) tenant app-request reservations.

rates that evenly divide the underlying IO resources between the tenants given their full (amplified) IO cost: 1300 GETs/1100 PUTs for the read-heavy tenants, 4900 GETs/1600 PUTs for mixed, and 290 GETs/2800 PUTs for the write-heavy tenants. All app-request reservations (dashed lines) and achieved throughput (solid lines) are given in terms of normalized 1KB requests. In both experiments, Libra is largely able to satisfy the tenant reservations from time 100 to 200. However, when lacking resource profiles, Libra provisions tenant VOP resources only in terms of the application-level object sizes, i.e., without accounting for secondary IO. This under provisions each allocation. Libra is thus able to meet the reservations only due to its work-conserving nature, which allows the tenants to freely share the unprovisioned resources. If the scheduler were rate-limited, tenant throughput would fall far short of their reservations.

At time 200, we decreased the app-request reservations for the read-heavy tenants by 50%, increased those of the write-heavy tenants by 50%, and left the mixed tenants unchanged. Under these conditions, Libra is able to fully achieve the desired app-level reservations only when app-request resource tracking is enabled. Since the write-heavy tenants fill up the WAL quickly with their frequent large PUT requests, background FLUSH operations constantly run to keep pace, consuming a nearly equivalent amount of write IO as the original PUTs. By tracking secondary VOP consumption with the app-request resource profiles, Libra can reprovision the requisite VOP allocations for the full app-request IO cost, from 2500 op/s to 3750 op/s. This gives the write-heavy tenants an average PUT throughput of ∼ 4300 PUT/s, satisfying their reservation of 4200 PUT/s. To afford these additional resources, Libra decreases the read-heavy VOP allocation just enough to still provide sufficient GET throughput (∼1300 GET/s) for its reduced app-request reservation. Without tracking enabled, Libra can only provision VOPs for the IO directly consumed by the write-heavy PUT requests. Given the imbalance of secondary IO costs, proportional sharing only provides an average PUT throughput of ∼3600 PUT/s, which violates the write-heavy tenants' reservations.

7.4.2 Libra adapts to dynamic tenant demand.

The second set of experiments starts with the same initial tenant workloads and reservations. Figure 7.6 shows the steady app-request throughput (top) and Libra's resource cost profiles (bottom). At time 200, the read-heavy and write-heavy tenants swap workloads—without changing their reservations—indicating an extreme shift in demand distribution. Then, at time 300, these tenants also swap their app-request reservations, which realigns with demand. The mixed tenants remain unchanged throughout. During the transition phase, the new read-heavy tenants ramp up their GET request rate slowly from time 200 to 250, while the new write-heavy tenants transition quickly from GET to PUT. This lag allows the mixed tenants to consume excess VOPs and boost throughput. Although Libra

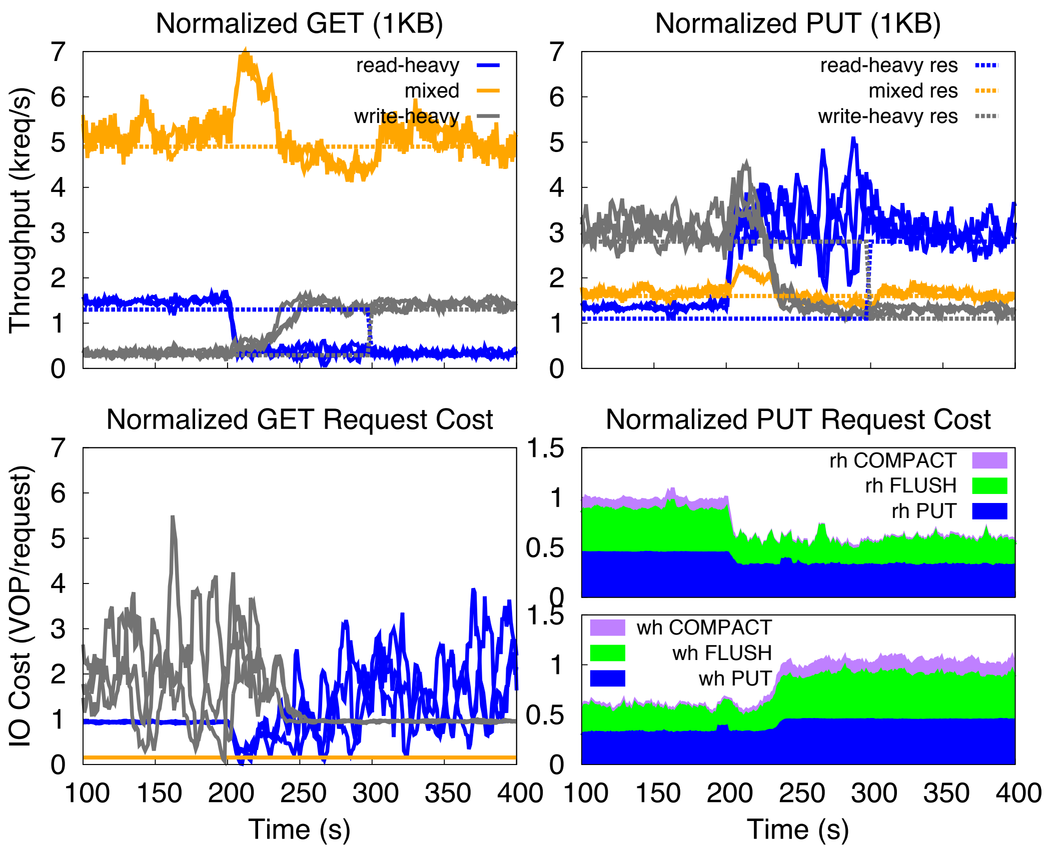


Figure 7.6: Libra adapts to shifting tenant demand.

provisions VOPs according to the individual app-request profiles and reservations, it does not impose a request-specific VOP limit; tenants can freely consume their VOP allocation according to any GET/PUT distribution.

The bottom graphs show the app-request cost transition for each tenant. At the outset, write-heavy tenants incur a highly amplified GET cost from having to search a larger set of eligible data files (see §6.2.1). Every so often, a background COMPACT reduces the data file set, causing the request cost to drop (and leading to the observed IO cost fluctuations in the bottom-left graph). As expected, the moderately sized GETs of the mixed tenants cost less per normalized request (i.e., per KB) than the read-heavy tenants' small GETs. For clarity, we only show the normalized PUT request cost for one representative read-heavy (rh) and one write-heavy (wh) tenant (in bottom right). Each graph breaks down the full PUT request cost by component. Since read-heavy tenants generate the smallest and fewest writes, they have high per-request PUT, FLUSH, and COMPACT costs. Write-heavy tenants, on the other hand, consume the least VOPs per request with their frequent large writes, which amortizes the cost of secondary operations across more normalized requests.

When the read- and write-heavy tenants swap workloads at time 200, Libra captures the shift in their resource profiles. However, because the reservations are misaligned—i.e., large PUT reservations for expensive read-heavy PUTs, and large GET reservations for expensive write-heavy GETs—the total VOP allocation exceeds the provisionable IO throughput. When overbooked, Libra penalizes the tenants equally. This results in a throughput drop for the mixed tenants which violates their reservations, and a gain for the workload-swapped tenants. Yet after reprovisioning the VOP allocations to align with the new reservations at time 300, Libra rebalances throughput and achieves the tenants' reservations.

Chapter 8

Conclusion

This thesis provides a principled solution to resource allocation and performance predictability for multi-tenant shared cloud storage. We presented a set of mechanisms that together provide *per-tenant* weighted fair sharing of *system-wide* resources for a multi-tenant, key-value storage service which we call Pisces. Using optimization decomposition, we showed how our novel set of four mechanisms—partition placement, weight allocation, replica selection, and fair queuing—combine to optimize throughput while maintaining fair resource allocation across the service nodes even when tenants contend for shared resources and demand distributions vary across partitions and over time. Our prototype implementation achieves near ideal fairness (0.99 Min-Max Ratio) and strong performance isolation for tenants with network bound workloads.

Libra addresses several fundamental challenges that arise for disk IO bound workloads at the local storage node level in provisioned multi-tenant storage. While the higher-level policies in Pisces continue to optimize for system performance within the tenant reservation constraints, Libra introduces a novel combination of virtual IOP resource modeling and application-level cost tracking to handle the non-linear, workload-dependent performance of SSDs and translate app-request throughput to low-level resources. With its resource consumption profiles, Libra can track the full, amplified IO cost of each application-level request and provision tenant throughput reservations accordingly. By capturing non-linear SSD performance with its VOP-based IO cost model, Libra not only realizes more accurate IO throughput allocations than previous resource models, but it can also safely provision IO resources for tenant reservations—using its IO capacity threshold—while still achieving high utilization for a wide range of workloads and IOP sizes. Together, these two techniques allow Libra to provide the basic per-node building block for shared key-value storage with provisioned tenant throughput.

8.1 Future Work

While Pisces and Libra provide a framework for building predictable shared cloud storage systems, several open questions remain.

- **Resource Scheduling:** Memory and CPU remain unregulated resources in the DFQ framework. While the storage systems we examined are primarily network and disk IO bound, in general large working sets and long-running queries can tax memory allocations. Similarly, user-defined functions with arbitrary computations can induce CPU contention. Although CPU scheduling has been studied in isolation, relatively little work has been done in the multi-resource setting. Moreover, for user-level multi-tenancy CPU scheduling must take application-level semantics into account since a single thread may service multiple tenants in a given time quanta. Arbitrating memory access is likely beyond the scope of a software resource scheduler. However, memory allocation can be an effective way to regulate resource consumption. By building a model of object cache usage, the scheduler can alleviate resource bottlenecks by allocating memory to the tenant that can use it most, i.e., the one with the highest cache hit rate.
- **Multi-tenant Shared Services:** Beyond key-value storage, we envision the Pisces framework as a more general architecture for predictable, provisioned service. Multi-

tenant messaging systems partition workload based on tenant queue, databases distribute and shard tenant tables by key, and application servers host web service modules from multiple tenants. While each of these systems exhibit very different workload characteristics that stress disparate resources, because they use a shared-nothing design they all can benefit from the Pisces mechanisms. However, the key question is what specific resource policies and algorithms work best in each setting to achieve maximum performance within the allocation constraints .

- **Multi-tier Shared Services:** Many storage systems, and more broadly, shared services are built on multi-tiered architectures, e.g., partitioned web servers accessing a shared key-value service that stores data in a distributed file system. Ensuring end-to-end application level throughput for a whole web service is a difficult proposition. One way to extend the Pisces mechanisms across multiple tiers would be to model the system as a tiered flow network of a vector-valued resource with edges weighted by their service node capacities. App-request profiles specific to each tier would inform the overall partition placement and weight (resource) allocation mechanisms on how to convert a tenant app-request flow from one tier to the next. This would allow the high level policies in Pisces to optimize the parameters for each service node in the system to achieve tenant allocations while providing maximum performance. Collecting and analyzing the per-tenant cross-tier app-request profiles could be accomplished through a control framework like IO-Flow [70].

Bibliography

[1] http://aws.amazon.com/dynamodb/faqs/, 2014.

[2] http://docs.amazonwebservices.com/amazondynamodb/latest/developerguide/Limits.html, 2014.

[3] http://code.google.com/p/spymemcached/, Jan. 2014.

[4] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual*, June 2008.

[5] Mohammad Al-Fares, Alex Loukissas, and Amin Vahdat. A scalable, commodity, data center network architecture. In *SIGCOMM*, Aug. 2008.

[6] Amazon. Amazon Relational Database Service. http://aws.amazon.com/rds/, 2014.

[7] Amazon. EBS Provisioned IOPs. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html, 2014.

[8] Amazon. Elastic Block Storage. http://aws.amazon.com/ebs/, 2014.

[9] Amazon. Simple Queue Service. http://aws.amazon.com/sqs/, 2014.

[10] Amazon DynamoDB. http://aws.amazon.com/dynamodb/, 2014.

[11] Amazon DynamoDB Pricing. http://aws.amazon.com/dynamodb/pricing/, 2014.

[12] Apache. Cassandra. http://cassandra.apache.org/, 2014.

[13] Jens Axboe. Linux Block IO—present and future. In *Ottawa Linux Symp.*, July 2004.

[14] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, Aug. 2011.

[15] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, UK, 2004.

[16] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *USENIX Annual*, June 2005.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *Trans. Computer Systems*, 26(2), June 2008.

[18] Mung Chiang, Stephen H. Low, A.R. Calderbank, and J. C. Doyle. Layering as optimization decomposition: A mathematical theory of network architectures. *Proceedings of the IEEE*, 95(1):255–312, January 2007.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, June 2010.

[20] Document-oriented NoSQL database. `http://www.couchbase.com/`, 2014.

[21] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, Sept. 1989.

[22] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *SOSP*, Dec. 1999.

[23] Simson L. Garfinkel. An evaluation of Amazon's grid computing services: EC2, S3 and SQS. Technical Report TR-08-07, Harvard Univ., 2007.

[24] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *SIGCOMM*, Aug. 2012.

[25] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, Mar. 2011.

[26] Google. App Engine Datastore. `https://developers.google.com/appengine/features/#datastore`, 2014.

[27] Pawan Goyal, Harick M. Vin, and Haichen Chen. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *SIGCOMM*, Aug. 1996.

[28] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *FAST*, Feb. 2009.

[29] Ajay Gulati, Arif Merchant, and Peter J. Varman. pClock: An arrival curve based approach for QoS guarantees in shared storage systems. In *SIGMETRICS*, June 2007.

[30] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: Handling throughput variability for hypervisor IO scheduling. In *OSDI*, Oct. 2010.

[31] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *CoNext*, Nov. 2010.

[32] Jiayue He, Rui Zhang-Shen, Ying Li, Cheng-Yen Lee, Jennifer Rexford, and Mung Chiang. Davinci: dynamically adaptive virtual networks for a customized internet. In *CoNext*, Dec. 2008.

[33] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, Mar. 2011.

[34] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In *CCGrid*, May 2011.

[35] Kimberly Keeton, Terence Kelly, Arif Merchant, Cipriano Santos, Janet Wiener, Xiaoyun Zhu, and Dirk Beyer. Don't settle for less than the best: Use optimization to make decisions. In *HotOS*, May 2007.

[36] Terry Lam, Sivasankar Radhakrishnan, Amin Vahdat, and George Varghese. Netshare: Virtualizing data center networks across services. Technical Report CS2010-0957, UCSD, May 2010.

[37] A fast and lightweight key/value database library by Google. `https://code.google.com/p/leveldb/`, 2014.

[38] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP*, Feb. 2009.

[39] Yandong Mao, Eddie Kohler, and Robert Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, Apr. 2012.

[40] John C. McCullough, John Dunagan, Alec Wolman, and Alex C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *USENIX Annual*, June 2010.

[41] Paul Menage. Cgroups. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`, Aug. 2013.

[42] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: Quality-of-service in large disk arrays. In *ICAC*, June 2011.

[43] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *SOSP*, Oct. 2011.

[44] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, May 1996.

[45] Microsoft. Windows Azure SQL. `http://www.windowsazure.com/en-us/services/sql-database/`, 2014.

[46] Ingo Molnr. [patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]. `http://lwn.net/Articles/230501/`, 2007.

[47] Robert M. Nauss. Solving the generalized assignment problem: An optimizing and heuristic approach. *INFORMS*, 15:249–266, 2003.

[48] Jason Nieh, Christopher Vaill, and Hua Zhong. Virtual-time round-robin: An O(1) proportional share scheduler. In *USENIX Annual*, June 2001.

[49] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4), June 1996.

[50] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *EuroSys*, Mar. 2009.

[51] D.P. Palomar and Mung Chiang. A tutorial on decomposition methods for network utility maximization. *JSAC*, 24(8):1439–1451, 2006.

[52] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with Lithe. In *PLDI*, June 2010.

[53] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *Trans. Networking*, 1(3), June 1993.

[54] Stan Park and Kai Shen. FIOS: A fair, efficient flash I/O schedule. In *FAST*, Feb. 2012.

[55] Larry Peterson, Andy Bavier, and Sapan Bhatia. VICCI: A programmable cloud-computing research testbed. Technical Report TR-912-11, Princeton CS, Sept. 2011.

[56] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, Aug. 2012.

[57] Rackspace. Cloud Block Storage. `http://www.rackspace.com/cloud/block-storage/`, 2014.

[58] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *WIOV*, June 2011.

[59] Kai Shen and Stan Park. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *USENIX Annual*, June 2013.

[60] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *NSDI*, Mar. 2011.

[61] David B. Shmoys and Eva Tardos. An approximation algorithm for the generalized assignment problem. *Math. Prog.*, 62(1):461–474, 1993.

[62] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *Trans. Networking*, 4(3):375–385, 1996.

[63] David Shue and Michael J. Freedman. From application requests to Virtual IOPs: Provisioned key-value storage with Libra. In *EuroSys*, 2014.

[64] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, Oct. 2012.

[65] David Shue, Michael J. Freedman, and Anees Shaikh. Fairness and isolation in multi-tenant storage as optimization decomposition. *OSR*, Jan. 2013.

[66] Emin Gun Sirer. Heuristics considered harmful: Using mathematical optimization for resource management in distributed systems. *IEEE Intelligent Systems*, pages 55–57, April 2006.

[67] Ion Stoica, Hussein M. Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *RTSS*, Dec. 1996.

[68] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *SIGCOMM*, Sept. 1997.

[69] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bulletin*, 9(1):4–9, 1986.

[70] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony I. T. Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: a software-defined storage architecture. In *SOSP*, Nov. 2013.

[71] LLC United Networks. Hyperleveldb. `http://hyperdex.org/performance/leveldb/`, 2014.

[72] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *SOSP*, Oct. 2003.

[73] Matthew Wachs, Michael Abd-el-malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, Feb. 2007.

[74] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling high-level SLOs on shared storage systems. In *SOCC*, Oct. 2012.

[75] J. Wang, P. Varman, and C. Xie. Optimizing storage performance in public cloud platforms. *J. Zhejiang Univ. – Science C*, 11(12):951–964, Dec. 2011.

[76] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. Fast TCP: Motivation, architecture, algorithms, performance. *Trans. Networking*, 14(6):1246–1259, Dec. 2006.

[77] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, Dec. 2008.