Fairness and Isolation in Multi-Tenant Storage as Optimization Decomposition

David Shue
Princeton University
dshue@cs.princeton.edu

Michael J. Freedman
Princeton University
mfreed@cs.princeton.edu

Anees Shaikh
IBM Research
aashaikh@us.ibm.com

ABSTRACT

Shared storage services enjoy wide adoption in commercial clouds. But most systems today provide weak performance isolation and fairness between tenants, if at all. Most approaches to multi-tenant resource allocation are based either on per-VM allocations or hard rate limits that assume uniform workloads to achieve high utilization. Instead, Pisces, our system for shared key-value storage, achieves datacenter-wide *per-tenant* performance isolation and fairness.

Pisces achieves per-tenant weighted fair sharing of system resources across the entire shared service, even when partitions belonging to different tenants are co-located and when demand for different partitions is skewed or time-varying. The focus of this paper is to highlight the optimization model that motivates the decomposition of Pisces's fair sharing problem into four complementary mechanisms—partition placement, weight allocation, replica selection, and weighted fair queuing—that operate on different time-scales to provide system-wide max-min fairness. An evaluation of our Pisces storage prototype achieves nearly ideal (0.98 Min-Max Ratio) fair sharing, strong performance isolation, and robustness to skew and shifts in tenant demand.

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, as well as use common platform services which include key-value stores, block storage volumes, and SQL databases. These services leverage the expertise of the cloud provider in building, managing, and improving common platforms, and enable the statistical multiplexing of resources between tenants for higher utilization and cost savings.

Because they rely on shared infrastructure, however, these services face two key, related issues:

- ***Multi-tenant interference and unfairness:*** Tenants simultaneously accessing shared service nodes 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 [17, 19].

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

Shared back-end storage services face different challenges than virtual machine (VM) provisioning of shared physical infrastructure. These stores divide tenant workloads into disjoint partitions, which are then distributed (and replicated) across different service nodes. 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 request rates can be scaled on demand. As with VMs, resource contention arises when tenants' partitions are co-located, however the degree of resource sharing between tenants may be significantly higher and more fluid.

To improve predictability for shared storage systems with a high degree of resource sharing and contention, we target ***global max-min fairness***. 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, when some tenants use less than their full share, unconsumed resources are divided among the rest to ensure high utilization. In comparison, recent commercial systems that offer request rate guarantees (*i.e.*, Amazon DynamoDB [1]) do not ensure fairness, assume uniform load distributions across tenant partitions, and are not work conserving. While our approach may be applicable to a range of services with shared-nothing architectures [14], we focus our design and evaluation on a key-value storage service, which we call *Pisces* (*P*redictable *S*hared *C*loud *S*torage).

Providing fair resource allocation and isolation 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 (read or write) rate and request size. 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, Pisces decomposes the problem of global fairness into four

Permission for classroom and personal use is granted, providing this notice appears on all copies.
LADIS Workshop '12. Madeira, Portugal.
Copyright 2012 by the Authors. This work derives from work published at the workshop. Appeared in *ACM SIGOPS Operating System Review*, Vol. 47 Issue 1, Jan. 2013 .

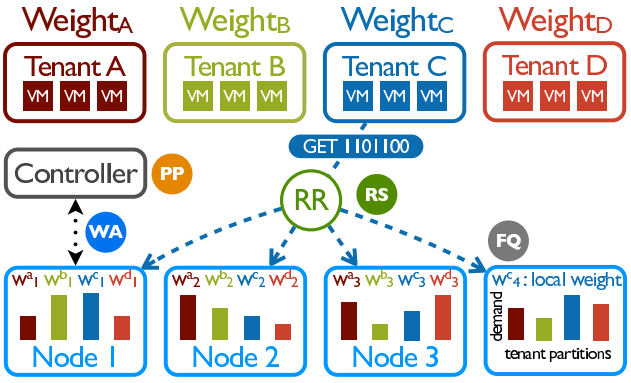


Figure 1: Pisces multi-tenant storage architecture.

mechanisms based on the NUM optimization framework [4]. 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 (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).

In this paper, we focus on the design and decomposition of the system mechanisms from an optimization perspective. While the global fairness problem is amenable to different approaches, leveraging optimization theory not only leads to performant algorithms, but also provides a basis for understanding and proving system properties [7]. Through experimental evaluation, we demonstrate that Pisces significantly improves the multi-tenant fairness and isolation properties of our key-value store, built on Membase [2], even as workloads shift dynamically. While this paper stresses our optimization-based approach, a more detailed treatment of the system design, implementation, and evaluation can be found elsewhere [13].

2. ARCHITECTURE AND SYSTEM MODEL

Figure 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 values (binary blobs) and supports simple key lookups (get), modifications (set), and removals (delete). To partition the workload, the keys are first hashed into a fixed-size key space and then subdivided into disjoint partitions.

A controller assigns these partitions (and their replicas) to service nodes while request router(s), which can be implemented in client libraries, or deployed as intermediate proxies (as shown in Figure 1), direct tenant requests to a node hosting the appropriate partition replica. Each service node schedules incoming requests and serves tenant data. Note that this architecture is not limited to key-value storage systems. Any system that employs workload partitioning (sharding) *i.e.*, partitioned databases, distributed block storage, scalable message queues etc, will have a similar structure and invoke similar mechanisms.

	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^t_p	t's fair share demand for partition p: $\sum_p f^t_p = z^t$
ρ^t	replicas per partition for tenant t
c_n	resource capacity for service node n
	Decision Variables
r^t_n	local resource share for t at n
w^t_n	local weight t at n: $w^t_n = \frac{r^t_n}{\sum_u r^u_n}$
Q^t	$\lvert N\rvert \times \lvert P^t\rvert$ partition replica selection matrix
	Global Fair-Sharing Optimization

maximize: $\Lambda(r^{t\in T}_{n\in N}, Q^{t\in T})$: throughput utility function (1)

subject to: $\sum r^t_n \le z^t$: fair share constraint (2)

$$\sum_n r^t_n \le c_n \text{ : node capacity constraint} \quad (3)$$

$$\sum_t Q^t_{n,p} \le r^t_n \text{ : local share constraint} \quad (4)$$

$$\sum_p Q^t_{n,p} > 0 = \rho^t \text{ : replication constraint} \quad (5)$$

$$\sum_n Q^t_{n,p} \le f^t_p \text{ : partition demand constraint} \quad (6)$$

Table 1: Pisces global fair-sharing system model

2.1 Achieving Global Fairness

Pisces provides per-tenant fairness at the system-wide level, which we model as the global optimization problem shown in Table 1. 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. 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^t_p, 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)*.

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 (Table 1, eq. 2) across the system, no node is over-burdened (3), all local allocations are enforced (4), and per-partition tenant demand is satisfied (6), we want to find the partition mapping $I(Q^t) > 0$, local resource allocation r^t_n, and replica selection policy Q^t that maximizes overall throughput (1).

While the global formulation gives a bird's-eye view of the system objective and fairness constraints, it serves solely as an orienting framework. The collection and coordination

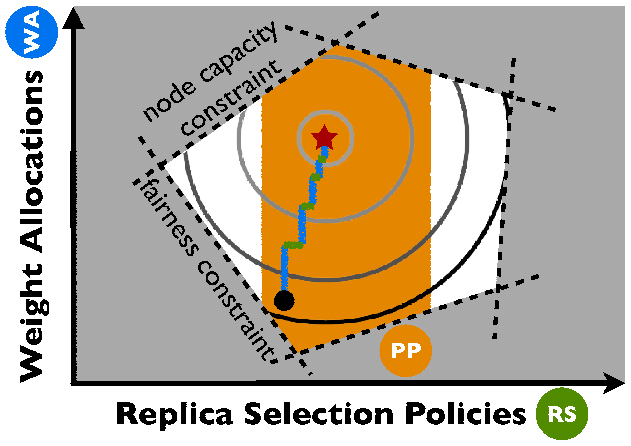


Figure 2: Partition Placement (PP) ensures *feasibility* by fitting each tenant's per-partition fair-share demand within the node capacity constraints (shaded region within the constraint set). Weight Allocation (WA) and Replica Selection (RS) then iteratively search for the optimal solution (star) starting from an initial configuration (dot) via coordinate gradient ascent, climbing through regions of progressively higher throughput (isoclines).

costs of 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 (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.

3. MECHANISMS AS DECOMPOSITION

We use optimization decomposition [4] to break the global problem down into the more tractable and adaptive pieces shown in Table 2. 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 2.

3.1 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 (Table 2, eq. 10) fits within the node capacities (8). This ensures that the other mechanisms search within the set of globally fair solutions, as shown in Figure 2). 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_p^t and optimizes the PP sub-problem shown in Table 2. The optimal partition assignment $I(Q_{n,p}^t) > 0$ minimizes the max node utilization (7) to give WA and RS greater headroom for optimizing throughput.

Partition placement typically runs on a long timescale (every few minutes or hours), since we assume that tenant demand distributions (proportions) are relatively stable, although demand intensity (load) can fluctuate more frequently. This minimizes the frequency of data partition migration which can affect on-going workloads. It also allows the faster timescale mechanisms to stabilize under the current partition mapping. That said, PP can also be executed in response to large demand shifts, severe fairness violations, or the addition or removal of tenants or service nodes. Although the bin-packing problem is NP-hard, for the system to work PP need only ensure that the assignment is *feasible*, not necessarily optimal. There are many approximation techniques (*e.g.*, [12]) that find near-optimal solutions in polynomial time. We use a simple greedy algorithm that works reasonably well.

Partition Placement (long timescale: min/hrs)

$$\textbf{minimize:}\quad \Upsilon(Q^{t\in T}, c_{n\subset N}) = \max_n \frac{\sum_{t,p} Q_{n,p}^t}{c_n} \tag{7}$$

$$\textbf{subject to:}\quad \sum_{t,p} Q_{n,p}^t \le c_n \text{ : node capacity constraint} \tag{8}$$

$$\sum_n Q_{n,p}^t > 0 = \rho_t \text{ : replication constraint} \tag{9}$$

$$\sum_n Q_{n,p}^t \ge f_p^t \text{ : demand constraint} \tag{10}$$

Weight Allocation (medium timescale: sec)

$$\textbf{maximize:}\quad \Lambda(r_{n\in N}^{t\in T}, Q^{*t\in T}) = \sum_{t,n} \log Q_n^{*t} \tag{11}$$

$$\textbf{subject to:}\quad \sum_n r_n^t \le z^t \text{ : fair share constraint} \tag{12}$$

$$\sum_t r_n^t \le c_n \text{ : node capacity constraint} \tag{13}$$

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

Replica Selection (real-time: ms)

$$\textbf{maximize:}\quad \Lambda(r_{n\subset N}^{t\subset T}, Q^{*t\subset T}) = \sum_{t,n} \log Q_n^{*t}$$

$$\textbf{subject to:}\quad Q_n^t \le r_n^{*t} \text{ : local share constraint} \tag{14}$$

$$\textbf{parameters:}\quad r_n^{*t} \text{ (fixed by WA). } I(Q^t) > 0 \text{ (fixed by PP)}$$

Table 2: Pisces mechanism decomposition

3.2 Weight Allocation

Given a feasible partition placement, weight allocation divides each tenant's global share into local shares r_n^t 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 variables together 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 2, we decompose the problem into the "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_n^t to match tenant demand then adjusting Q^t to leverage the larger local shares, as depicted in Figure 2.

Using the primal [10] approach, WA observes per-node tenant request latencies, which acts as a proxy for tenant demand, and increases local shares r_n^t to match tenant demand. On each iteration, WA collects an estimate of the per-node tenant demand $x_n^t = \sum_p Q_{n,p}^{*t}$ generated by RS. Using

this estimate, WA approximates the gradient of the RS Lagrangian w.r.t r_n^t ($\frac{\partial L(Q,\lambda)}{\partial r_n^t} = \lambda_n^t$), with a latency-based cost function: $l_n^t = 1\,/\,(r_n^t - x_n^t)$. Minimizing this cost function maximizes throughput (Table 2, eq. 11) since λ_n^t is a "congestion" price corresponding to the request queuing delay experienced by tenant t at node n. Thus, WA minimizes the max latency by performing a *reciprocal swap* that shifts weight (local share) from a lower latency tenant u on node n to the max latency tenant t ($\text{argmax}_{t,n} l_n^t$) and reciprocates the exchange (from t to u) on a different node m to preserve global fairness (12, 13). WA computes the swap as the linear bisection of the latencies, $y(t,u,n) = \frac{(r_n^u - x_n^u) - (r_n^t - x_n^t)}{2}$, and uses the minimum of the swap steps, $\min(y(t,u,n), y(t,u,m))$ in the exchange. This ensures that the swap always reduces the maximum latency. We also model multilateral swap exchanges as a maximum bottleneck flow (MBF) problem [13], but omit the details for space. Since it requires global information (max latency), WA also runs on the controller.

Since WA 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} within each WA iteration. Taken together, these properties ensure that WA will converge to an optimal fair-share weight allocation [3] (< 20 iterations in our experiments). To avoid oscillations around the optimal point, only swaps that exceed a minimal threshold ϵ are executed.

3.3 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 (Table 2, eq. 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 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 allocation.

Here, we apply dual decomposition [10] 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 maintain per-node, rather than per-node per-partition, request windows (Q_n^t) for each tenant. Each RR updates its per-node windows according to the FAST-TCP gradient ascent equation, which optimizes the throughput objective (11) subject to per-node congestion (λ_n^t), approximated by request latency: $w(m+1)_n^t = (1-\alpha)\cdot w(m)_n^t + \alpha \cdot \left(\frac{l_{\text{base}}}{l_{\text{est}}}\right)$.

Each iteration of the algorithm adjusts the window based on the ratio of the desired average request latency l_{base} to the smoothed (EWMA) latency estimate l_{est}. The α parameter limits the window step size. Thus, the greater the local share, the larger the node's request window will be. Each request router makes adjustments to its own Q_n^t in a fully decentralized fashion: it only uses local request latency measurements to compute the replica proportions. This allows RS to handle short-lived fluctuations and converge to the optimal Q_n^t within the WA timescale according to the convergence and stability of FAST-TCP [18].

3.4 Fair Queuing

Although not explicitly involved in the optimization, 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 coordination. 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 every "round" of FQ, the server allocates tokens to each tenant according to its local weight $w_n^t = \frac{r_n^t}{\sum_u r_n^u}$, which it consumes when processing requests from the tenant queues. If the request requires more than the allocated resources, it must complete on a subsequent round after its tenant's tokens have been refilled. This guarantees that each tenant will receive its local fair share r_n^t in a given round of work, if multiple tenants are active. Otherwise, tenants can consume excess resources left idle by the others without penalty. We implement FQ using deficit weighted round robin; details [13] omitted for space.

4. EVALUATION

In our evaluation, we consider how the mechanisms in Pisces work together to (i) provide fairness and performance isolation, (ii) achieve weighted fair sharing, and (iii) handle dynamic demand. We quantify fairness as the Min-Max Ratio (MMR) of the dominant resource (typically throughput) across all tenants, $\frac{x^{\min}}{x^{\max}}$. This corresponds directly to a max-min notion of fairness.

Our testbed consists of sixteen 2.4 GHz quad-core machines (8 clients and 8 servers) connected to a single 1 Gbps switch. Each client uses the Yahoo Cloud Storage Benchmark (YCSB) [5] to generate a Zipf-distributed key-value request workload ($\alpha = 0.99$) over a fully cached data set of 100,000 1kB objects. All workloads are read-only (all GET), unless otherwise noted. We only present the most illustrative examples (see [13] for the full set).

4.1 Achieving Fairness and Isolation

As a basis for comparison, we start with an unmodified system (Membase), to establish a baseline, as shown in Figure 3. Then we add in fair queuing, followed by partition placement and weight allocation. Lastly, we enable replica selection. In the top row, 8 tenants with equal global weights access the system with the same demand.

Unmodified Membase: The unmodified system provides poor throughput fairness between tenants. This is largely due to the infeasible (uniform) partition mapping of the skewed tenant demand distributions. In contrast, PP packs the partitions according to the fair-share demand and node capacity constraints to ensure feasibility.

Multi-tenant Weighted FQ: Unsurprisingly, fair queuing alone barely improves fairness due to over-contention

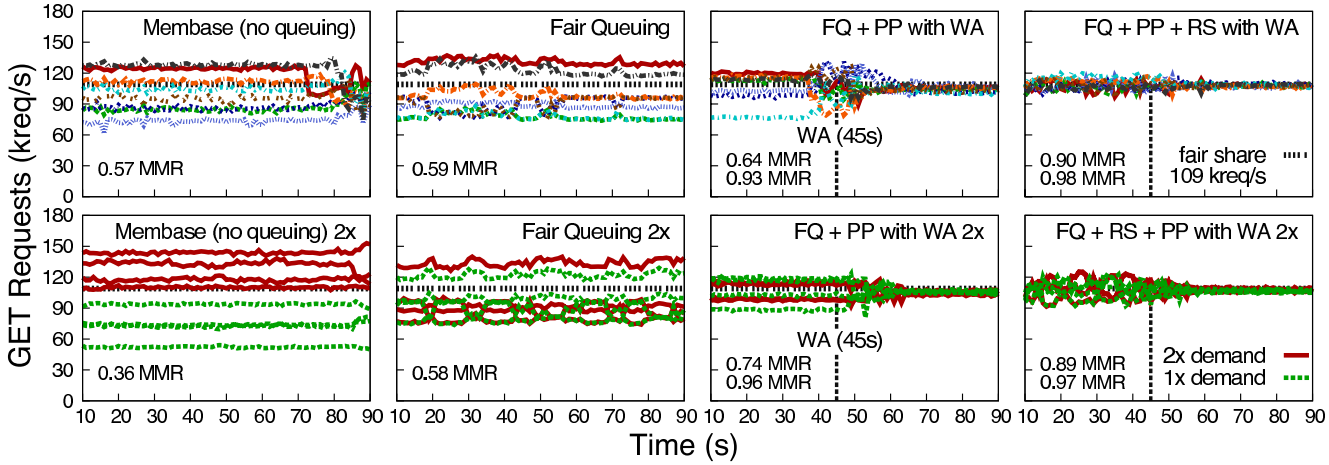


Figure 3: System-wide throughput fairness (top) and performance isolation (bottom) with Pisces mechanisms. For experiments involving weight allocation (columns 3 and 4), WA is activated at time 45s.

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.

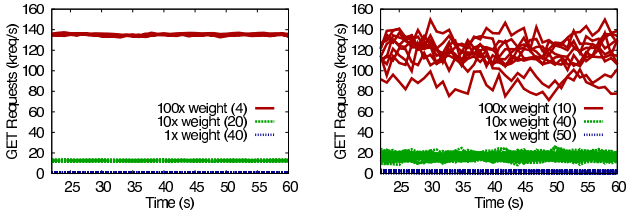
FQ and Partition Placement: Despite starting with a pre-computed feasible partition placement , fairness only improves marginally. Although the tenant demand should fit within node capacity, hotspots still remain. The mismatch between the hotspots and the initially fixed (uniform) local weights allows tenants with an unfairly large local share on a given node to "over" consume and thus exceed their global fair share. Once weight allocation starts at 45s, the local shares converge within 10 seconds (5 iterations) to their optimal fair values (0.93 MMR).

FQ, PP, and Replica Selection: When enabled, replica selection improves fairness by alleviating 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 able to adjust the selection policy in tandem with WA to find the optimal fair solution 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.

In the bottom row of Figure 3, 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).

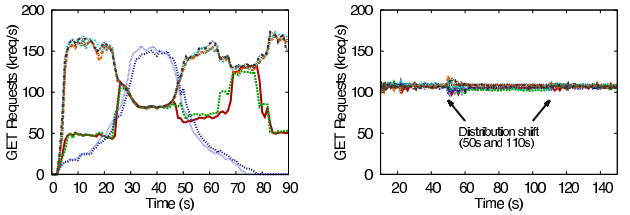
4.2 Service Differentiation

Thus far, we have demonstrated that Pisces's mechanisms can enforce isolation (FQ) and achieve near ideal even fair



(a) 64 tenant thruput (b) 100 tenant thruput

Figure 4: Pisces achieves global fairness for skewed tenant weights on an 8 (a) and 20 node cluster (b).



(a) Dynamic intensity (b) Dynamic distribution

Figure 5: Pisces responds to demand dynamism (a) and distribution shifts (b) to preserve fairness.

sharing (PP + WA + RS). We now examine *weighted* fairness for service differentiation. In Figure 4a, 64 tenants reside on 4 out of 8 servers (32 tenants per server). To reflect the highly skewed nature of tenant shares, *i.e.*, a few heavy hitters and low-rate users, we assigned 4 weight-100, 20 weight-10, and 40 weight-1 tenants. Within each weight class, Pisces achieves > 0.91 MMR. Unfortunately, fairness between the highest and lowest classes decreases to 0.56 MMR, due to the limits of the FQ scheduler.

Figure 4b shows a larger experiment, with 100 tenants resident on 6 of 20 servers (30 tenants per server). While we see a qualitatively similar result, fairness degrades (0.46 MMR across all classes on average). This is mostly due to performance variance on the (shared) scale-out testbed [11] arising from CPU scheduling and network bottlenecks, which penalizes the high-weight tenants.

4.3 Dynamic Workloads

Dynamic workloads present a challenge for any system to provide consistent, predictable performance. In Figure 5a, two bursty tenants (weight 1), two diurnal demand tenants (weight 2), and four constant demand tenants (weight 1) access the system. Initially, the constant tenants are able to exceed their fair share and consume the excess capacity. As the diurnal tenants ramp up (0–20s), they gradually reduce the excess share. When the bursty and diurnal tenants peak around 20s, Pisces enforces the proper 2-to-1 weighted ratio between all tenants. Around 50s, the diurnal and bursty tenants tail off which allows the constant demand tenants to once again consume the excess. Finally, at 70s, the bursty tenants spike again, which forces the constant and bursty tenants to receive equal shares.

Demand distributions can evolve as well. In Figure 5b, 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. RELATED WORK

Recent work on cloud storage resource sharing has focused mainly on single-tenant or single-server scenarios. Parda [6] applies FAST-TCP congestion control to provide per-VM fairness, which Pisces uses as well, but for replicated service nodes. Maestro [9] optimizes I/O resource and port allocation for multiple applications, but on a single disk array. Similarly, Argon [15] uses caching schemes and time-sliced disk scheduling for performance insulation between multiple clients accessing a single shared file server. FAST [8] presents a block-storage specific design for minimizing workload interference, but does not address weighted resource sharing. Cake [16] adapts resource shares in a two-tier system to achieve latency-based SLO's for disk-bound workloads. Additional related work can be found in [13].

6. CONCLUSION

In this paper 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 the 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.

Although we focus on key-value storage in this work, we believe that the optimization model and mechanisms should apply to a wider range of services. Any system built using a shared-nothing architecture will have to manage partition placement and replica selection. Enforcing fairness and isolation requires some form of fair queuing or resource allocation at the point of contention. To these we introduce one additional component, weight allocation, and link them together using optimization decomposition. Thus we see these mechanisms as providing a fairness framework for a variety of services, which we intend to pursue in future work.

Acknowledgments We thank Jennifer Rexford for helpful discussions early in this project. Funding was provided through NSF CAREER Award #0953197.

7. REFERENCES

[1] `http://aws.amazon.com/dynamodb/faqs/`, 2012.
[2] `http://www.couchbase.org/`, Jan. 2012.
[3] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, UK, 2004.
[4] M. Chiang, S. H. Low, A. 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.
[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, June 2010.
[6] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *FAST*, Feb. 2009.
[7] K. Keeton, T. Kelly, A. Merchant, C. Santos, J. Wiener, X. Zhu, and D. Beyer. Don't settle for less than the best: use optimization to make decisions. In *HotOS*, May 2007.
[8] X. Lin, Y. Mao, F. Li, and R. Ricci. Towards fair sharing of block storage in a multi-tenant cloud. June 2012.
[9] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-service in large disk arrays. In *ICAC-11*, 2011.
[10] D. Palomar and M. Chiang. A tutorial on decomposition methods for network utility maximization. *JSAC*, 24(8):1439–1451, 2006.
[11] L. Peterson, A. Bavier, and S. Bhatia. VICCI: A programmable cloud-computing research testbed. Technical Report TR-912-11, Princeton CS, Sept. 2011.
[12] D. B. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Math. Prog.*, 62(1):461–474, 1993.
[13] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, Oct. 2012.
[14] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bulletin*, 9(1):4–9, 1986.
[15] M. Wachs, M. Abd-el-malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, Feb. 2007.
[16] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: enabling high-level SLOs on shared storage systems. In *SOCC*, Oct. 2012.
[17] 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.
[18] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. Fast TCP: Motivation, architecture, algorithms, performance. *Trans. Networking*, 14(6):1246–1259, Dec. 2006.
[19] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, Dec. 2008.