From application requests to Virtual IOPs: Provisioned key-value storage with Libra

David Shue* and Michael J. Freedman
Princeton University

Abstract

Achieving predictable performance in shared cloud storage services is hard. Tenants want reservations in terms of system-wide application-level throughput, but the provider must ultimately deal with low-level IO resources at each storage node where contention arises. Such a guarantee has thus proven elusive, due to the complexities inherent to modern storage stacks: non-uniform IO amplification, unpredictable IO interference, and non-linear IO performance.

This paper presents Libra, a local IO scheduling framework designed for a shared SSD-backed key-value storage system. Libra guarantees per-tenant application-request throughput while achieving high utilization. To accomplish this, Libra leverages two techniques. First, Libra tracks the IO resource consumption of a tenant's application-level requests across complex storage stack interactions, down to low-level IO operations. This allows Libra to allocate per-tenant IO resources for achieving app-request reservations based on their dynamic IO usage profile. Second, Libra uses a disk-IO cost model based on *virtual IO operations* (VOP) that captures the non-linear relationship between SSD IO bandwidth and IO operation (IOP) throughput. Using VOPs, Libra can both account for the true cost of an IOP and determine the amount of *provisionable* IO resources available under IO interference.

An evaluation shows that Libra, when applied to a LevelDB-based prototype with SSD-backed storage, satisfies tenant app-request reservations and achieves accurate low-level VOP allocations over a range of workloads, while still supporting high utilization.

1. Introduction

The rapid adoption of cloud storage services, e.g., key-value stores [5], block storage volumes [3, 27], and SQL databases [2, 15], has escalated the challenge of resource sharing. By adopting these services, cloud tenants leverage the expertise of the service provider in building, managing, and provisioning the common storage platform, while providers reap the benefits of statistical multiplexing for higher margins and lower costs. However, because these multi-tenant services rely on shared infrastructure, resource contention arises wherever tenant workloads overlap on the same physical substrate. This contention can degrade and destabilize throughput, leading to highly variable application performance.

For tenants running critical workloads that require predictable performance , *provisioned* (that is, guaranteed) resource allocation is essential. While provisioning low-level IO resources such as IO operations (IOP) is simpler for the provider, it requires the often opaque and onerous task of IO resource estimation on the part of the tenant. Since tenants only see the high-level requests they issue to the storage service, not the underlying IO operations, they must fine-tune their workloads to meet the provisioned resource constraints. Instead, tenants prefer to reserve resources in terms of *application-level* request (app-request) throughput, such as key-value GET/s rather than IOP/s, which allows them to specify exactly what they need from the storage system.

This paper describes Libra, an IO resource scheduling framework for provisioning app-request throughput in a multi-tenant key-value storage system. 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. These local reservations, which are computed by higher-level policies (e.g., Pisces [30]), combine together to satisfy the tenants' system-wide throughput reservation. Libra leverages high-throughput, low-latency SSD storage to provide predictable IO performance for disk-bound workloads, while preserving high utilization.

Although reservations in terms of application-level requests, rather than low-level IO, present a simple and accessible resource interface to the tenant, they introduce new technical challenges for the provider. In particular, the provider must address 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 to enforce tenant IO allocations? The

*Current affiliation: Google, Inc.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). Copyright is held by the owner/author(s).
EuroSys 2014, April 13–16 2014, Amsterdam, Netherlands
ACM 978-1-4503-2704-6/14/04.
http://dx.doi.org/10.1145/2592798.2592823

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 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 op sizes increase.

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. Existing approaches typically use hard rate limits [3, 5] 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. Libra uses two key techniques to combat the complexities of provisioning app-request reservations in terms of low-level IO resources.

(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) Account for "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 to achieve the tenants' per-node application-level throughput reservations in our LevelDB-based [20] prototype storage node. Libra achieves these reservations over a wide range of workloads including intermixed reads, writes, and varying op 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. 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), per our later evaluation.

While we specifically address provisioned key-value storage in this paper, we believe that Libra's app-request resource tracking, IO capacity threshold, and low-level IO cost model, should generalize to other shared storage services with synchronous disk workloads as well.

2. Provisioned Key-Value Storage

Large-scale key-value storage services commonly employ a shared-nothing architecture, depicted on the left side of Figure 1. For scalability and fault tolerance, tenant data is divided into disjoint (replicated) partitions and dispersed across storage nodes. Clients (or intermediate request routers) route tenant object requests to local nodes based on the partition mapping and, if enabled, a replica-selection policy.

2.1 Provisioning system-wide throughput

To achieve system-wide request throughput reservations, the storage system must first map tenant partitions to storage nodes that can accommodate the aggregate demand. Then, the system should subdivide each tenant's global reservation among the local nodes based on the request load at each node. Commercial systems like DynamoDB often push this burden to the tenant: they require uniform demand across all partitions, which allows the provider to distribute local tenant reservations uniformly across the storage nodes. On the other hand, research systems like Pisces [30] employ dynamic policies for placing partitions, distributing local reservations, and balancing partition demand to support arbitrary tenant workloads that may vary over time.

However, the effectiveness of such system-wide policies ultimately depend on how well the low-level mechanisms at each storage node enforce local app-request reservations. If individual nodes fail to arbitrate between disparate tenants' requests for objects in collocated partitions, both local and system-wide tenant reservations would suffer. Thus, in Libra, we turn our attention from the system-wide reservation problem, which has been addressed by prior work [30, 33], to focus on provisioning disk (SSD) IO resources to achieve tenant app-request reservations for disk-bound workloads at each local node. We first introduce the design of Libra, then

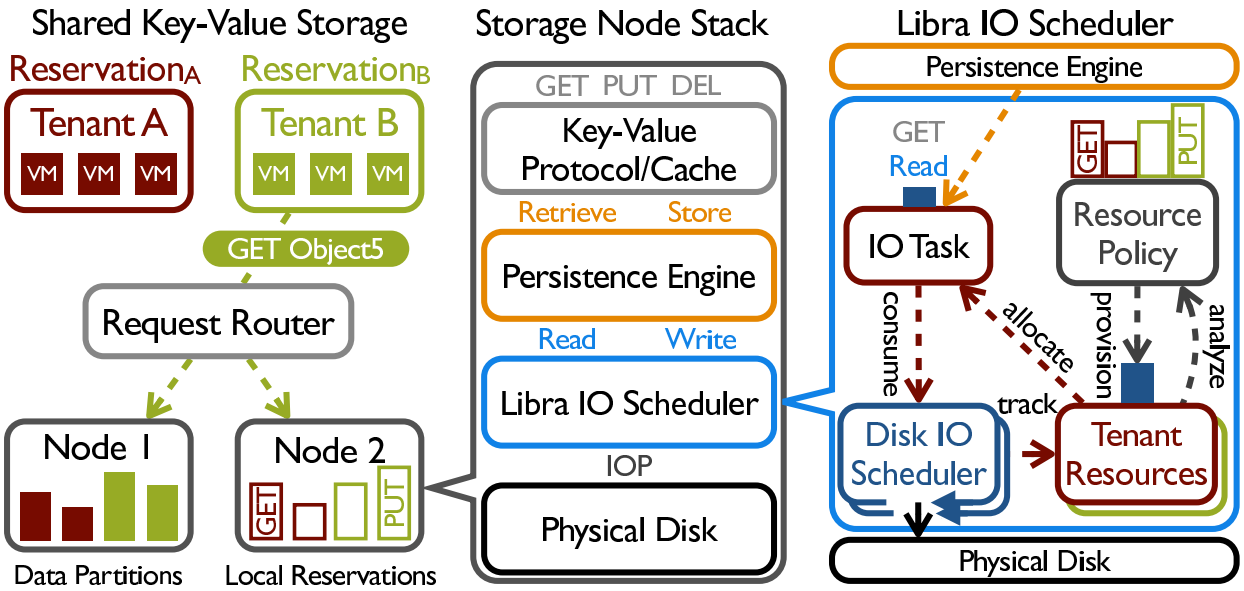


Figure 1: Tenant data is split into partitions and distributed across the key-value storage system. Each storage node runs a storage stack to store objects and serve requests. Libra provisions low-level IO resource allocations and schedules tenant IO to achieve per-node tenant app-request reservations.

describe the confounding factors inherent to storage stacks and how Libra addresses each of these challenges.

2.2 The Libra IO scheduling framework

Although key-value storage systems typically expose a simple request API to their clients (e.g., GET or PUT), the internal architecture of each storage node is layered and complex, as shown in the middle of Figure 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.

As shown on the right side of Figure 1, Libra is situated 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. 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. Should tenant workload fluctuations cause the cost of locally allocated IO operations to exceed the IO throughput, Libra can signal higher-level policies to migrate partitions and redistribute local reservations.

When the persistence engine issues IO tasks (reads and writes) to the Libra scheduler, each low-level IO task is tagged with the resource principal (tenant) and originating 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 [29] 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. In the background, Libra's resource policy analyzes these resource profiles to determine the necessary IO resources and provisions each tenant's resource allocation. Recall that the local app-request reservations handled by the resource policy are specified in normalized (1KB) GET and PUT requests and are set by higher level policies.

3. The Problem of Predictability

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

3.1 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.

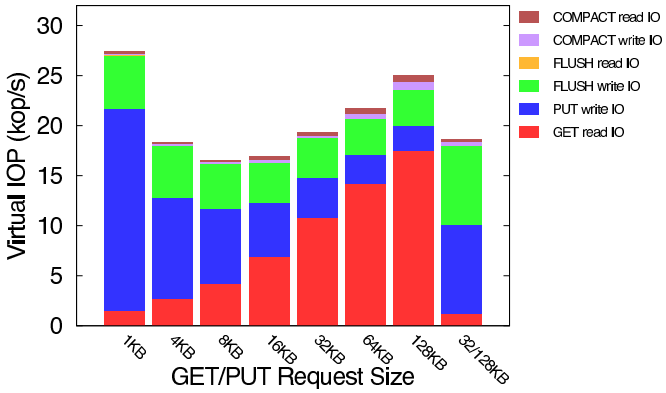


Figure 2: App-request IO consumption varies non-uniformly with tenant workload.

Storage engines with immutable data formats, such as log-structured merge (LSM) trees [7, 10, 20, 25], 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. 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 with frequent overwrites.

App-request reads can be equally complex. 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 formats, on the other hand, may require additional (indexed) lookups depending on the number of immutable indexed data files in the tree and their key ranges. 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 §4.3. Figure 2 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.

3.2 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 [1]. 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 [1] (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 [26], but they also incur a large erase-before-write penalty. The SSD must first 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 ops can adversely affect read latency and overall IOP throughput.

SSD firmware typically employs a log-structured approach to minimize the effects of 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 collection 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 erase-block increments.

3.3 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 $\times$ op-size), both vary non-linearly with operation size due to shifting

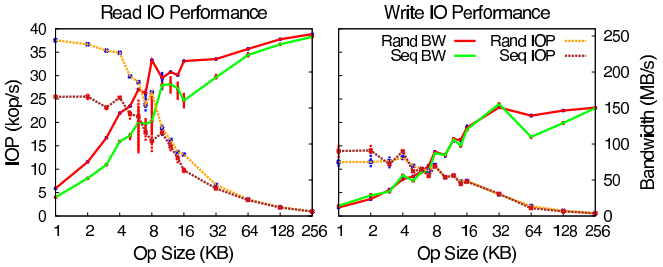


Figure 3: SSD IOP/s and bandwidth vary non-linearly with IOP size. The measurements were made on an Intel 320 series SSD formatted with the ext4 filesystem.

bottlenecks, as shown in Figure 3. 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. Further, filesystem (ext4) overhead affects sequential IO performance more than random IO in these experiments. Yet we account for these anomalies, as the storage stack sits above the filesystem.

Accounting for only one of these IO bottlenecks can leave the system underutilized. For example, in DynamoDB, one 100KB GET costs the same as one hundred 1KB GETs [6]. 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.

4. 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.

4.1 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^t_a for tenant t and app-request a) and the indirect resources consumed by internal operation i on behalf of a (u^t_i). These tags also indicate how often an app-request triggers an internal operation ($e^t_{a,i}$), 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)

$$q^t_a = EWMA(u^t_a / s^t_a)$$
$$q^t_i = EWMA(u^t_i / s^t_i)$$

where s^t_a and s^t_i are the number of normalized (1KB) requests and internal operations executed over the interval, respectively. From these base costs, Libra computes the indirect resource cost $q^t_{a,i}$ by scaling the internal operation cost by how often the app-request triggers the operation on average:

$$q^t_{a,i} = q^t_i \frac{e^t_{a,i}}{s^t_a}$$

Some internal operations, especially COMPACT, are triggered sporadically and may take many intervals to complete. To handle this case, Libra normalizes $q^t_{a,i}$ 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 give the full app-request resource profile, which the resource policy uses to determine the IO resource allocation r^t_a needed to provision each tenant app-request reservation v^t_a:

$$\text{profile}^t_a = q^t_a + \sum_i q^t_{a,i}$$
$$r^t_a = v^t_a \cdot \text{profile}^t_a$$

These 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 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.

4.2 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

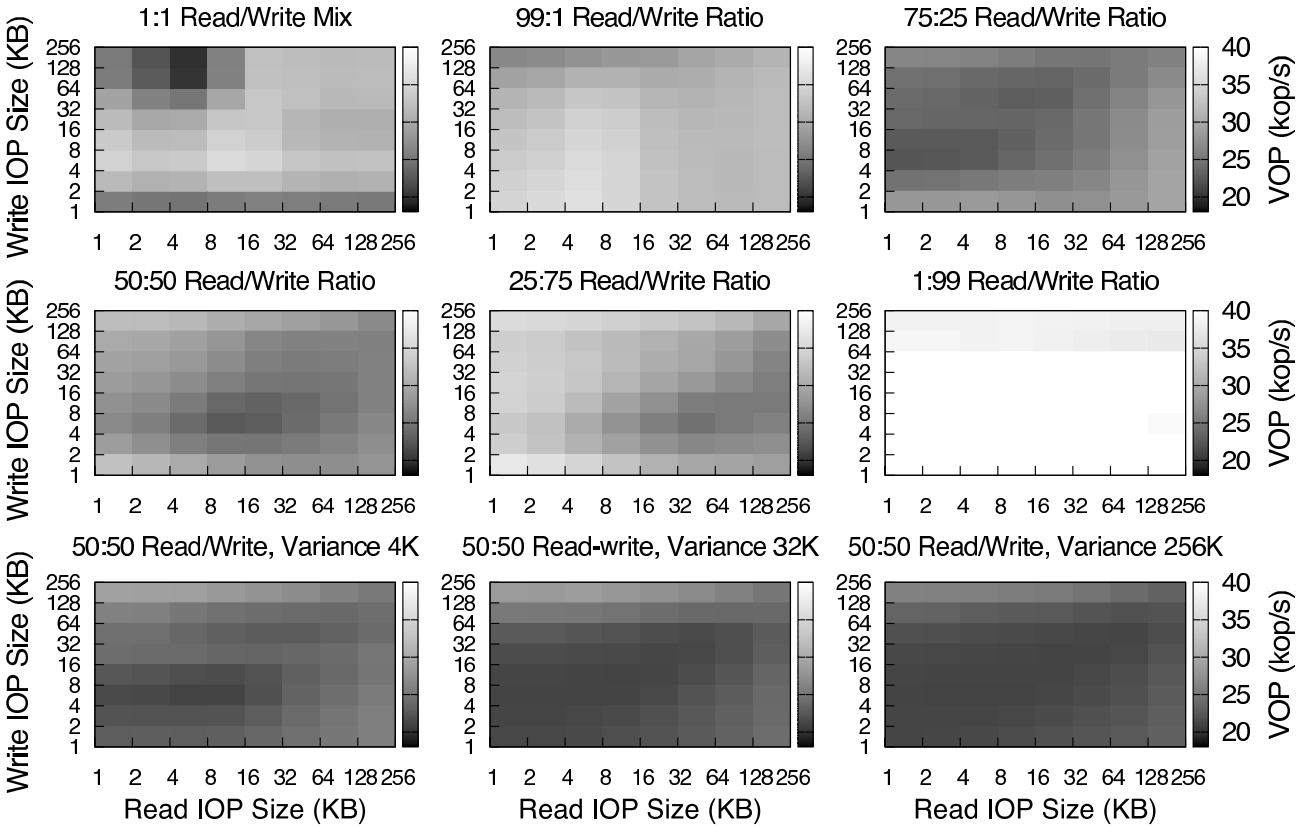


Figure 4: IO throughput varies unpredictably with IO interference. Throughput valleys (darkest regions) shift with read/write ratio and overall performance flattens out as IOP size variance increases. Each heat map shows experiments across a range of read (x-axis) and write (y-axis) IOP sizes from 1 KB to 256 KB.

IO interference. Since tenants can generate a 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 §4.3, 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 4 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 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 op sizes. Write-dominant (1:99) workloads experience little IO degradation except at large write sizes.

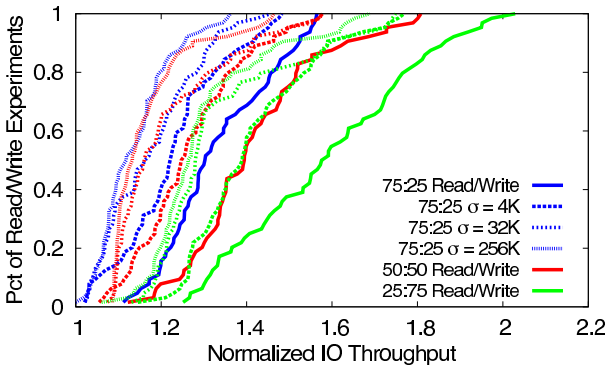


Figure 5: CDF of IO throughput for the IOP sizes shown in Figure 4. Solid lines correspond to workloads with uniform IOP sizes, while dashed and dotted line workloads issue variable request sizes with the indicated σ.

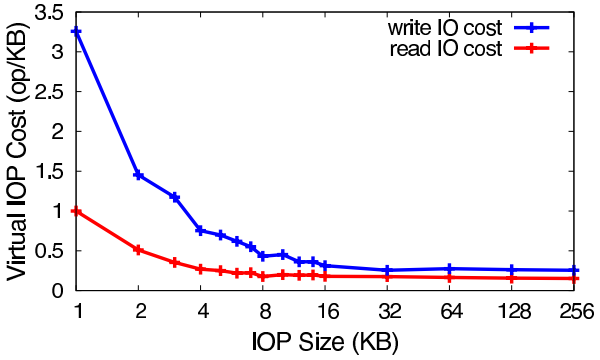


Figure 6: Libra IO cost model.

Randomizing IOP sizes—shown in the last row of Figure 4—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 5 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 overfitting, 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 can 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 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 5, at most one third of IO throughput is left unprovisioned, but still usable since Libra is work-conserving.

4.3 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 [14] 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 [8, 26, 32] or virtual-time fair queuing [19, 28]. Because time-slicing allots 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, on the other hand, 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 request 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 ac-

cording to a non-linear cost model derived directly from the IOP throughput curves, as shown in Figure 6. 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.

Internally, Libra's scheduler threads perform distributed deficit round robin [21] (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 [12] and round-robin [29] 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 [4].

5. Implementation

Libra exposes a posix-compliant IO interface, wrapping the underlying IO system calls to enforce resource constraints and interpose scheduling decisions. To utilize Libra, applications, i.e., persistence engines, simply replace their existing IO system calls (read, write, send, recv, etc) with the 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. The Libra IO scheduling framework is implemented in ~20000 lines of C code as a user-space library.

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), 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 for replacing system calls and marking application requests. However, unlocking LevelDB's full performance for synchronous PUTs required extensive modifications. Our prototype enables parallel writes to take full advantage of SSD IO parallelism (LevelDB serializes all client write threads by default). It also issues sequential writes (e.g., 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).

We implemented Libra as a user-space library for two main reasons. First, the model of multi-tenancy model we support at the storage node is a single 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 key-value storage servers [7, 11]. Existing kernel mechanisms for multi-tenant resource allocation, (e.g., linux cgroups [22]), 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 lock contention in the kernel. Second, tracking app-request resource consumption across system call boundaries requires additional OS support for IO request tagging (as described in [24]), 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. 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.

6.1 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.

6.2 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* [32]. In the same way, a virtual IOPS 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.

Virtual IOP allocation achieves physical IO insulation. Figure 7 shows the IOP throughput ratios achieved by Libra for the read/write tenants on 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 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.

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 [19, 28] 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 8 plots the different IO cost curves, including the curve-*fitted* Libra cost model, for the Intel SSD. The IO cost curves for the Samsung and OCZ SSDs (omitted for space) exhibit the same shape and behavior. Compared to Libra, the constant model charges a much higher cost-per-byte, while the linear and fixed models undercut the Libra cost curve for both reads and writes.

Figure 9 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. All results shown are for the Intel SSD. Only Libra's exact and fitted IO cost models were able to achieve a median MMR greater than 0.9 over all workloads; 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 ops). However, between the end points, the IO cost deviation results

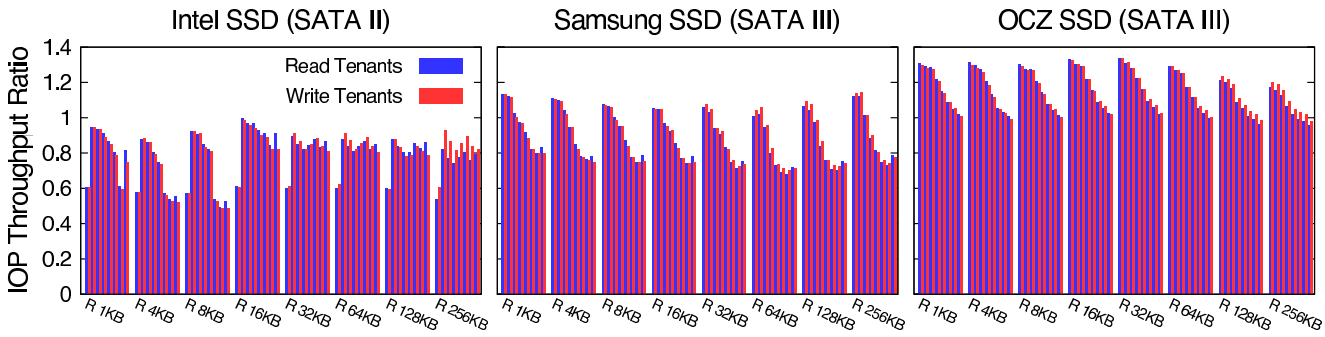


Figure 7: The Libra VOP resource model achieves near perfect (equal) IOP throughput ratios between read and write tenants on different SSD architectures, even in the presence of IO interference (ratio < 1). Read IOP size is fixed for each experiment in a cluster, while write IOP size logarithmically increases from 1 KB to 256 KB. Each read/write tenant *pair* within a cluster represents a single experiment at a specific read and write IOP size.

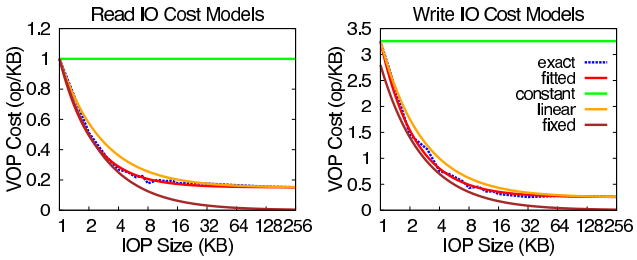


Figure 8: Virtual IOP cost models.

in skewed allocations and lower MMR. Despite egregiously over-charging IOP sizes greater than 1KB, 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 ratios. Results for the Samsung and OCZ SSDs were similar, though the linear model was able to achieve better insulation with median MMR close to 0.9. This is due to the lower IO throughput variance for those devices. However, medium IOP sizes still suffer (MMR < 0.8).

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 bottom graph in Figure 9 shows, 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

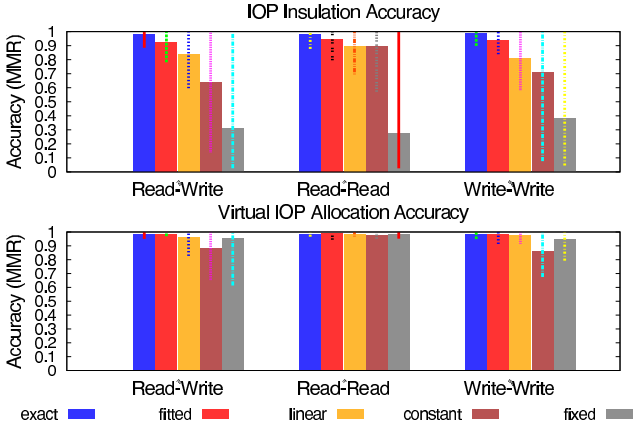
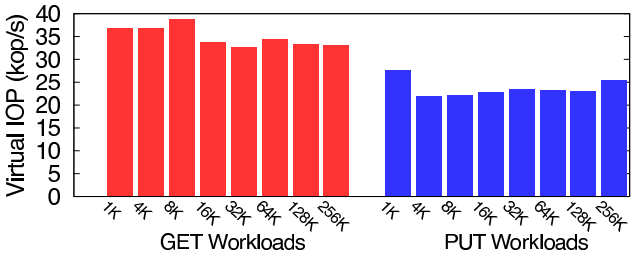


Figure 9: IOP insulation reflects how well a cost model captures tenant IOP cost while VOP allocation corresponds to the allocation and accounting fidelity of the Libra scheduler for enforcing tenant VOP shares. Each MMR bar (median with min/max error) summarizes the MMR results over the set of workloads shown in Figure 7.

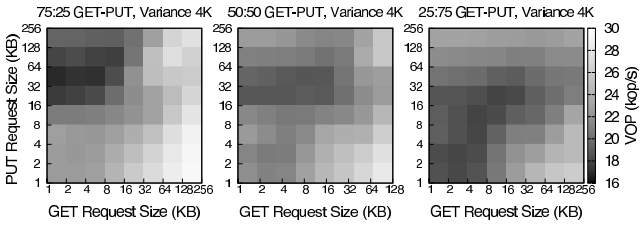
VOP under-consumption. These results confirm that poor IO throughput insulation is due to IO cost model inaccuracies and not faulty scheduler VOP accounting.

6.3 Libra trades nominal throughput for provisionable IO guarantee

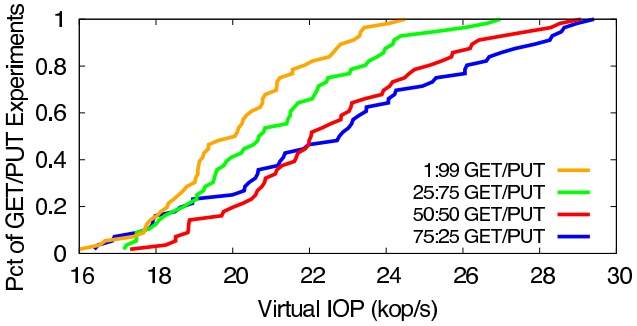
Per Section 4.2, IO interference can be both severe and unpredictable for mixed tenant workloads. In light of this workload-dependent variability, Libra sets the provisionable IO capacity to a VOP threshold of 18 kop/s. To understand whether this "floor" is a viable underestimate of provisionable capacity 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 [20] 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.



(a) Pure GET/PUT Workloads



(b) Mixed GET/PUT Workloads



(c) CDF of GET/PUT IO Throughput

Figure 10: IO throughput varies widely with tenant app-request workloads. For 80% of the workloads, Libra's IO threshold (18 kop/s) ensures that at least 69% of the achievable IO throughput (26 kop/s) is provisionable.

As a baseline, we ran pure GET and PUT workloads over the 1 to 256 KB request size range shown in Figure 10a. 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 results in wide-ranging VOP throughput, as shown in Figure 10b. In these experiments, tenants issue a mix of GET and PUT requests at the specified ratio. Request sizes are drawn 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 10c. 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 30% 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 below 20%. The few cases where VOP throughput may fall below the floor can be absorbed by the storage SLA in the 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 IO-bound workloads 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 16%. Note that workloads with highly variable request sizes diminish IO throughput and further reduce the unprovisionable excess.

6.4 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.

Tracking resource profiles enables accurate app-request provisioning. Figure 11 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 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

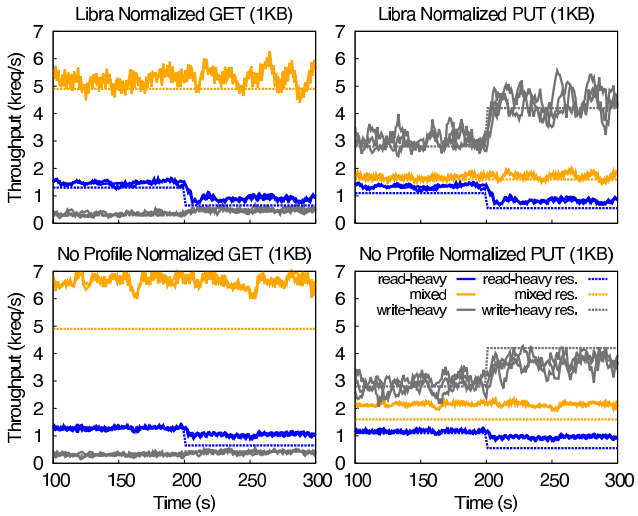


Figure 11: With app-request profile tracking, Libra achieves (dynamic) tenant app-request reservations.

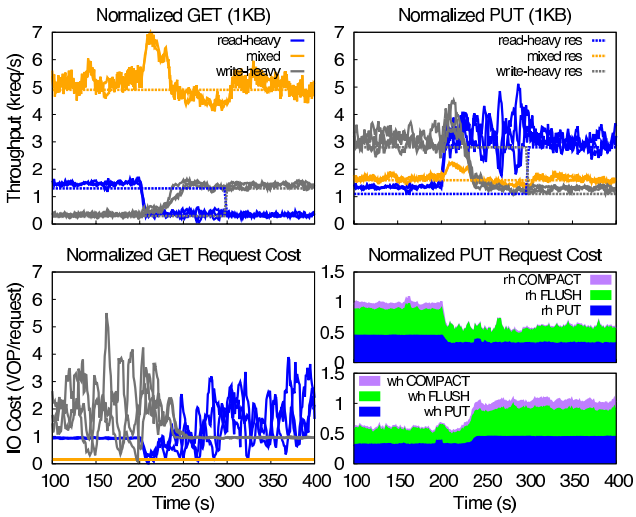


Figure 12: Libra adapts to shifting tenant demand.

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.

Libra adapts to dynamic tenant demand. The second set of experiments starts with the same initial tenant workloads and reservations. Figure 12 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 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 §3.1). 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.

7. 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) [23, 33], or providing proportional shares [18, 26, 28, 30]. Maestro [23] 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 [33] 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 [19] and Parda [18] 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.

Pisces [30] provides per-tenant weighted fair-shares of system-wide throughput in a key-value storage system. It uses a combination of high-level policies to place partitions, allocate local weights, and distribute load to ensure each tenant receives its aggregate throughput share. At the local node, Pisces employs DRF [14] to schedule over network resources. Libra fits into the overall storage model described in Pisces, but provides app-request throughput reservations in terms of disk IO resources for tenants that require synchronous writes. Pisces relies on its in-memory cache to support high read throughput and persists writes to disk asynchronously.

Disk IO scheduling. The literature is replete with work on fair IO resource scheduling ranging from the network [12, 16, 29, 31], to CPU [9, 21], to storage [13, 17, 19, 32]. Most relevant to our work are the FIOS [26] and FlashFQ [28] 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 [8, 32], 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.

8. Conclusion

Libra addresses several fundamental challenges in achieving provisioned application-level throughput reservations for multi-tenant key-value storage. It does so by tracking app-request IO resource consumption and modeling IO cost and capacity using virtual IOPs. With its resource consumption profiles, Libra can track the full, amplified IO cost of each application-level request and provision tenant throughput reservations accordingly. These resource profiles also provide system-wide schedulers with a model of app-request cost to inform (re)allocation when local demand exceed available resources. 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 pro-

vide the basic per-node building block for shared key-value storage with provisioned tenant throughput.

Acknowledgments. The authors would like to thank Amy Tai, Wyatt Lloyd, Erik Nordström, Rob Kiefer, Siddhartha Sen, Matvey Arye, our shepherd Hitesh Ballani, and the anonymous Eurosys reviewers for their helpful comments. This work was supported by funding from a National Science Foundation CAREER Award (CSR-0953197) and the Intel Science and Technology Center for Cloud Computing.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual*, 2008.

[2] Amazon. Relational database service. `http://aws.amazon.com/rds/`, 2014.

[3] Amazon. Elastic block storage. `http://aws.amazon.com/ebs/`, 2014.

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

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

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

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

[8] J. Axboe. Linux Block IO—present and future. In *Ottawa Linux Symp.*, 2004.

[9] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *USENIX Annual*, 2005.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *Trans. Computer Systems*, 26(2), 2008.

[11] Couchbase. Document-oriented NoSQL database. `http://www.couchbase.org/`, 2014.

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

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

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[15] Google. Google cloud SQL. `https://cloud.google.com/products/cloud-sql/`, 2014.

[16] P. Goyal, H. M. Vin, and H. Chen. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *SIGCOMM*, 1996.

[17] A. Gulati, A. Merchant, and P. J. Varman. pClock: An arrival curve based approach for QoS guarantees in shared storage systems. In *SIGMETRICS*, 2007.

[18] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *FAST*, 2009.

[19] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling throughput variability for hypervisor IO scheduling. In *OSDI*, 2010.

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

[21] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP*, 2009.

[22] P. Menage. Cgroups. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`, 2014.

[23] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-service in large disk arrays. In *ICAC*, 2011.

[24] M. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated storage services. In *SOSP*, 2011.

[25] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4), 1996.

[26] S. Park and K. Shen. FIOS: A fair, efcient flash I/O schedule. In *FAST*, 2012.

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

[28] K. Shen and S. Park. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *USENIX Annual*, 2013.

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

[30] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, 2012.

[31] I. Stoica, H. Zhang, and T. S. E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *SIGCOMM*, 1997.

[32] M. Wachs, M. Abd-el-malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, 2007.

[33] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling high-level SLOs on shared storage systems. In *SOCC*, 2012.