Burstable Cloud Block Storage with Data Processing Units

Junyi Shu[*†] Kun Qian[†] Ennan Zhai[†] Xuanzhe Liu[*] Xin Jin[*]

[*] *School of Computer Science, Peking University* [†] *Alibaba Cloud*

Abstract

Cloud block storage (CBS) is a key pillar of public clouds. Today's CBS distinguishes itself from physical counterparts (e.g., SSDs) by offering unique burst capability as well as enhanced throughput, capacity, and availability. We conduct an initial characterization of our CBS product, a globally deployed cloud block storage service at public cloud provider Alibaba Cloud. A key observation is that the storage agent (SA) running on a data processing unit (DPU) which connects user VMs to the backend storage is the major source of performance fluctuation with burst capability provided. In this paper, we propose a hardware-software co-designed I/O scheduling system BurstCBS to address load imbalance and tenant interference at SA. BurstCBS exploits high-performance queue scaling to achieve near-perfect load balancing at line rate. To mitigate tenant interference, we design a novel burstable I/O scheduler that prioritizes resource allocation for base-level usage while supporting bursts. We employ a vectorized I/O cost estimator for comprehensive measurements of the consumed resources of different types of I/Os. Our evaluation shows that BurstCBS reduces average latency by up to 85% and provides up to 5× throughput for base-level tenants under congestion with minimal overhead. We verify the benefits brought by BurstCBS with a database service that internally relies on CBS, and show that up to 83% latency reduction is observed on customer workloads.

1 Introduction

Cloud Block Storage (CBS) is a fundamental storage service on public clouds. It provides virtualized block-level storage volumes that can be dynamically provisioned and attached to compute instances. Beyond what an SSD can already offer, a CBS disk can provide additional benefits, including millions of IOPS, tens of terabytes capacity, higher durability with data replication, and out-of-box encryption support [1–4].

CBS adopts storage disaggregation to achieve better elasticity [5–9]. The disaggregated architecture of CBS empowers public clouds to independently scale storage and compute resources. Block storage volumes can be created, resized, and destroyed on demand without disrupting compute instances. Such agility and flexibility allow enterprises to right-size storage for different workloads in the cloud.

Due to the wide adoption of storage disaggregation, researchers have studied various technical aspects of it, including SSD co-optimization [10–12], kernel improvement [13, 14], kernel bypassing [7, 8, 10–12], storage-oriented network [5–8], performance analysis [15], and applications [16]. The performance of disaggregated storage is rapidly improving with the aforementioned system advances as well as the adoption of faster network/storage devices.

Through our analysis of the operational statistics of our production clusters and characterization of the storage agent (SA) which connects user VMs to the backend storage, we draw three key insights of CBS at Alibaba Cloud.

First, the bottleneck is shifted to compute nodes. Many existing systems target the bottleneck of SSDs [10, 11, 17]. We show that the distributed nature of cloud storage backend design and the over-provisioning tendency of cloud users result in relatively low utilization of storage servers and devices in terms of throughput. These characteristics of CBS have shifted performance bottleneck from the backend to the compute nodes. While backend traffic is well balanced by design and enjoys the benefit of the law of large numbers, compute nodes experience frequent traffic fluctuation due to unpredictable usage patterns of users.

Second, the burst capability of cloud block storage amplifies the chance of congestion. Cloud providers offer on-demand access to extra CPU and storage resources for running instances [18–20]. User VMs are provided with a *base-level* performance and the burst capability allows them to handle temporary traffic *bursts* above the base level. However, it can put a strain on the underlying server and cause performance degradation. When a single VM bursts to an extremely high throughput or multiple VMs on the same server burst concurrently, VMs compete for the limited available resources,

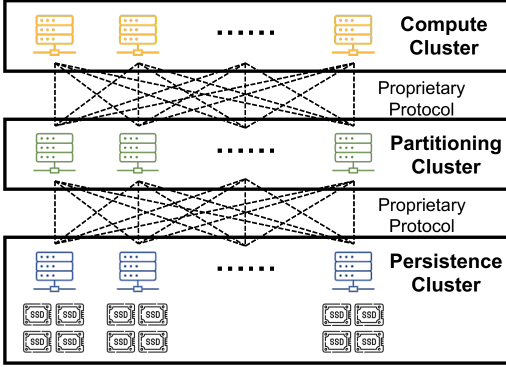


Figure 1: CBS architecture.

which leads to congestion. Moreover, heavy bursting VMs can crowd out other VMs running at a steady state.

Third, lack of resource scheduling at data processing units (DPUs) is the root cause of performance interference. Today, DPUs have become a standard component on compute nodes in the cloud [21–25]. A DPU runs hypervisor and network/storage functions freeing up host CPUs for customer usage. However, handling bursts on a DPU is extremely challenging for SA, as a DPU only possesses limited processing capability. Therefore, SA must strive for high resource utilization and provide performance isolation under congestion.

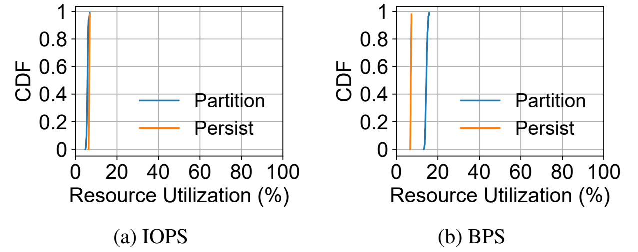
With a brief summary, there are two extra implicit requirements for CBS when supporting burst. (i) *Comprehensive resource utilization:* SA should fully leverage the available processing capacity of the DPU to support higher bursts and avoid congestion in the first place. (ii) *Base-level performance guarantee:* SLOs must not be violated for a tenant who does not exceed its base-level provisioning.

However, the existing SA meets neither of the goals by default. First, SA maps user queues statically to I/O threads running on DPU cores. An I/O thread can become congested while other threads are still idle when a certain user queue starts bursting. Second, an I/O thread serves I/Os in a First-Come-First-Serve (FCFS) manner as long as a VM is within its burst limit. VMs running at a steady state suffer high queuing delay as a result when other VMs are bursting.

In this paper, to overcome load imbalance and tenant interference induced by bursts, we present BurstCBS, a storage I/O scheduling system that leverages the hardware features of our custom xDPU and software characteristics of our SA. The design of BurstCBS offers a number of benefits. First, it equally distributes I/Os to all I/O threads. Second, it allows high bursts, while providing guarantees on base-level performance per VM. Third, it detects different types of bottlenecks on xDPU, avoiding triggering false congestion control. Finally, BurstCBS achieves the above benefits while keeping high resource utilization and minimal scheduling delay.

BurstCBS integrates three key techniques to realize the aforementioned benefits. First, considering the significant overhead introduced by inter-thread messaging, we leverage hardware capability and extensively optimize SA to implement I/O-granularity load balancing at line rate. With limited DPU memory, we design a two-tier memory pool that dynamically adjusts shared and queue-dedicated memory to achieve high I/O performance and memory efficiency. Second, we design a novel burstable I/O scheduler that is aware of the base-level/burst provisioning of each tenant. Rather than achieving high resource utilization with a work-conserving scheduler or guaranteeing strong performance isolation among multiple tenants, it dynamically attempts to provide burst capability to tenants that have excessive demand while monitoring and protecting base-level performance for other tenants in real time. Third, we design a vectorized I/O cost estimator that decouples potential resource bottlenecks that SA may encounter on xDPU. It vectorizes I/O cost and adjusts its estimation with a delay-based approach to allow the scheduling algorithm to react to resource contention effectively.

We implement BurstCBS as a standalone system package running on xDPU, and integrate it into the existing I/O workflow of SA. We conduct a comprehensive evaluation with various types of workloads and show that BurstCBS effectively protects base-level tenants while incurring a negligible overhead. Overall, BurstCBS reduces average latency by up to 85% and achieves up to 5× throughput for base-level tenants under congestion. A database service that internally relies on CBS reports that up to 83% reduction of SQL query average latency is observed with BurstCBS deployed.



(a) IOPS (b) BPS

Figure 2: CDF of IOPS and BPS (bits per second) utilization in a production cluster. 100% IOPS utilization is defined as the maximum IOPS a node can saturate on a 4KB random I/O workload and 100% BPS utilization is defined as the maximum bandwidth that a node can saturate on a 128KB sequential I/O workload.

2 Background

CBS architecture. Figure 1 shows the three-layer architecture of CBS at Alibaba Cloud. User VMs are hosted in the compute cluster, and all I/Os generated by VMs are forwarded to the partitioning cluster for further processing. The partitioning cluster controls data placement and failover, hiding the complexity from the compute cluster [5, 9]. The persis-

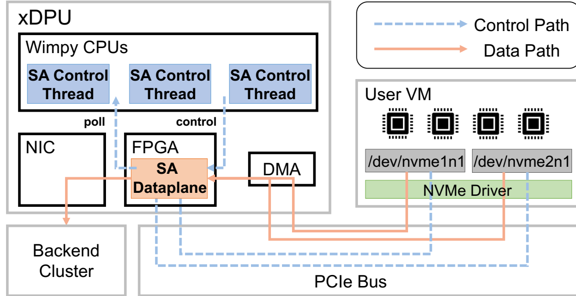


Figure 3: I/O workflow on xDPU.

tence cluster has tens of SSDs equipped on each node and is responsible for data persistence. Nodes in different layers are fully connected through a proprietary storage-optimized protocol to achieve load balancing at the backend clusters (i.e., partitioning cluster and persistence cluster).

CBS characteristics. A virtualized user disk is divided into multiple segments which are then distributed among all partitioning nodes. A partitioning node again divides each segment into smaller chunks which are evenly stored and replicated among persistence nodes. With this two-layer load-balancing design, storage I/Os are evenly distributed among all partitioning nodes and persistence nodes. An interesting phenomenon in the backend clusters is the asymmetry between IOPS/BPS utilization and disk capacity utilization. Figure 2 shows the distribution of IOPS/BPS per backend node in one of our most active production clusters in a peak hour. Storage accesses are balanced among all backend nodes resulting in low IOPS/BPS utilization on them while 78% disk capacity is utilized.

xDPU. At Alibaba Cloud, we design and build xDPU, an SoC that offloads infrastructure services from CPUs of compute nodes. It consists of its own compute resources (CPU, memory), programmable hardware accelerators (FPGA), network interfaces, and a DMA engine that can directly access guest VM memory over PCIe. The latest version of xDPU integrates eight 2.0GHz cores which are shared among storage, network, and administration functions. There are two 100Gbps network Ethernet ports available for use, and the DMA engine has about the same data movement throughput.

Storage agent. A storage agent (SA) is installed on an xDPU. It abstracts virtualized storage for VMs and connects the backend storage. SA consists of a control plane and a data plane. The SA control plane runs in wimpy cores of xDPU in user space. Among all cores, 2–4 of them are dedicated to SA control threads. The SA datapath logic is burned into FPGA, which moves data with the assistance of the DMA engine. A total of 100Gbps NIC/PCIe bandwidth is made available for SA. In Figure 3, we take NVMe WRITE operation as an example to explain how an I/O is processed by SA on xDPU. When a VM issues an NVMe WRITE command, the command is directly forwarded to FPGA on xDPU. The control

Table 1: Block storage burst capability of public clouds.

	CloudA	**CloudB**	**Alibaba Cloud**
Burst support	✓	✓	✓
Credit-based burst	✓	✓	✓
Paid burst	✗	✓	✓
Max burst IOPS	3k	30k	1000k
Max burst BPS (MB/s)	N/A	1000	4096

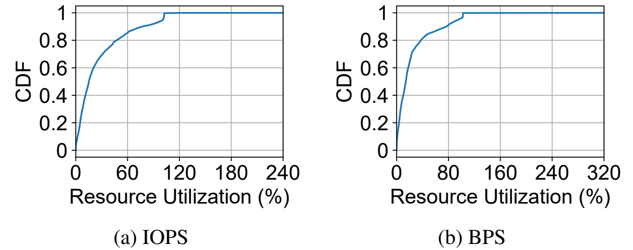


Figure 4: CDF of base-level throughput utilization in a production cluster.

threads of SA keep polling FPGA for new I/Os. When a control thread receives a new I/O, it splits the I/O into packets and constructs headers for them. Meanwhile, FPGA fetches the actual data via DMA. Once the header and body are ready, FPGA sends the packet to the backend via fabric.

We adopt an FPGA-CPU cooperative SA design instead of a fully FPGA-offloaded solution for three reasons. First, there is complex branching in I/O splitting and packet encapsulation for cloud block storage, which can hardly utilize the parallelism provided by FPGA. The wimpy CPU cores on xDPU yet have a much higher clock speed than FPGA, which makes CPU cores the right place to handle that part of logic. Second, SA must maintain a significant amount of states including thousands of connections to the distributed backend. The FPGA on xDPU does not have that much memory, while FPGA with more memory does not justify the cost. Third, from an engineering perspective, development and testing of FPGA code require much more effort which prevents us from rolling out new features quickly. We only consider offloading a software feature when it becomes mature enough.

3 Key Observations and Implications

We explain why the partitioning and persistence clusters are not the bottleneck in §2. In this section, we share two observations on the compute nodes of production CBS at Alibaba Cloud and reveal the corresponding challenges of providing predictable performance for cloud block storage.

Observation 1: The Burst capability of CBS makes compute nodes a common bottleneck. Conventionally, cloud providers provision VMs with fixed CPU, memory, and I/O resources to customers. However, a fixed amount of resources can barely match the dynamic workload faced by cloud users.

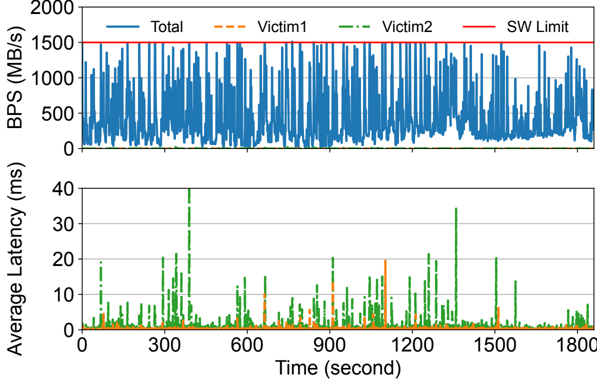


Figure 5: An incident of tenant interference under burst.

To bridge this gap, public clouds provide burstable VMs [26–29] as an option. Essentially, burstable VMs provide a base level of resources with the ability to burst above that when needed. Burstable VMs often come with a credit system. A VM accumulates credits when running below its base-level throughput while spending them to burst when it needs extra resources to saturate its demand.

Making block storage burstable is crucial for delivering burstable VMs because many workloads are bottlenecked on storage I/Os. In Table 1, we compare the burst capability of three widely used public clouds. All three cloud providers support credit-based burst which accumulates tokens for a user that does not use all of its base-level throughput. These tokens allow a user to burst when its desired throughput is beyond its provisioning. CloudB and Alibaba Cloud further allow users to burst on demand and pay for the extra throughput. What makes our case unique is that we allow a disk to burst up to 1 million IOPS and 4GB/s read/write BPS (subject to VM instance types and configurations).

We decide to allow this extreme burst capability for two reasons. First, it is a substantial requirement of our customers, as some of our CBS users have extremely bursty traffic. Figure 4 shows the distribution of base-level throughput utilization per disk in a production cluster. Some disk bursts over 300% of its base-level throughput. Second, fulfilling the burst requirement can further improve the overall resource utilization as well. The majority of our users over-provision their base-level throughput. In Figure 4, over 80% of disks use less than half of their base-level throughput. This is not a unique phenomenon at Alibaba Cloud. There is also previous work that reports the over-provisioning tendency of cloud users [30].

Although credit-based burst is attractive, it raises a strong challenge for us to provide predictable performance. Because users keep accumulating tokens as long as they are below base-level throughput, many VMs may possess tokens and start to burst at the same time. When it happens, every tenant on the impacted compute node observes higher latency and lower throughput due to congestion.

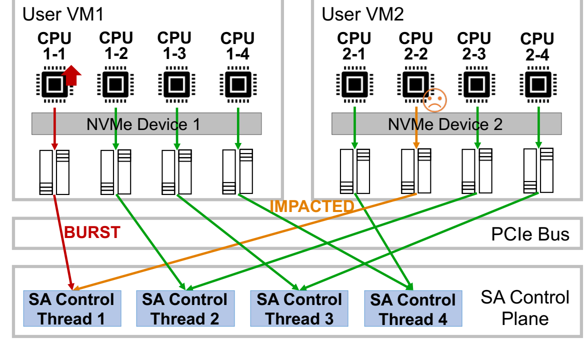


Figure 6: Impact of imbalanced load.

Our existing system addresses this problem by limiting the number of co-located VMs. Under a simplified model, assume a VM has the probability p to burst in a time interval, and we have N VMs on the compute node. We know that VMs will experience performance interference if k out of N VMs burst concurrently. The chance of performance interference is:

$$P(X \geq k) = 1 - \sum_{i=0}^{k-1} \binom{N}{i} (1-p)^{N-i} p^i \quad (1)$$

If we want to limit this probability to a small number, we have to limit the number of VMs that can co-locate on a compute node, which hurts the overall resource utilization. In production, over 95% of the compute nodes have 32 or fewer VMs allocated, which helps us maintain our SLA during 99% of the time. Indeed, although we have made this number small enough, we still observe performance interference occasionally. Figure 5 shows one such incident. During half an hour, multiple tenants were bursting and the software limit on xDPU was frequently triggered resulting in throttling. Although Victim1 and Victim2 ran below their base-level BPS steadily, they both observed many unexpected millisecond-scale average latency spikes during that time.

Observation 2: Inter-thread load imbalance and intra-thread resource contention are the major sources of performance interference on xDPU. In a VM, each vCPU corresponds to a queue pair. These queue pairs are mapped to an SA control thread in a round-robin manner by FPGA on xDPU. However, this mapping cannot adapt to dynamic workload in real time and is prone to cause load imbalance among SA control threads. Figure 6 gives an example. CPU 1-1 of VM 1 and CPU 2-2 of VM 2 are both mapped to SA control thread 1. When CPU 1-1 is creating a burst I/O stream, control thread 1 becomes congested. I/Os from CPU 2-2 experience a high queuing delay even if other control threads are idle. In short, uneven I/O intensity per vCPU created by users and static vCPU to SA control thread mapping through FPGA together lead to inter-thread load imbalance.

Bursting from a single thread in a VM is a common pattern of many I/O intensive applications because they are generally

Table 2: Parallel execution support of popular databases. Non-modifying ops refer to operations that do not change any database records.

	PE support	Enabled by default
Oracle	✓	✗
MySQL	✗	✗
SQL server	non-modifying ops	✓
PostgreSQL	non-modifying ops	✓
MongoDB	✓	✓
Redis	✗	✗
Elasticsearch	✓	✓
Db2	✓	✗
SQLite	non-modifying ops	✗
Access	✗	✗

built under the assumption that host CPU is not the bottleneck. In Table 2, we surveyed parallel execution support of the top 10 database management systems in terms of popularity [31]. Relational databases either do not fully support parallel subqueries on multiple CPUs or do not enable parallel execution by default. Redis is also known for its single-threaded design.

Our online experience also confirms this phenomenon. Figure 7 shows the I/O intensity per vCPU for 1000 randomly sampled 4-core VMs in a minute in a production cluster. Over 80% of the total I/Os are from the most I/O-intensive core.

Besides inter-thread load imbalance, intra-thread resource contention is another cause of unexpected performance degradation. Processing an I/O consumes CPU cycles in the SA control plane to construct the packet header and interconnect bandwidth in the SA data plane to transmit the actual data. When there is a resource contention, an I/O has to wait in queues until the resources are available.

We observe there are two typical cases that an I/O stream is impacted as an undesired result. First, a burst tenant with a high I/O parallelism has a significantly better chance of acquiring the resources than a base-level tenant. In Figure 8a, when we increase the I/O parallelism of a burst tenant which creates a mixture of 4KB to 128KB I/Os in the background, serial write I/Os from the base-level tenant are also queued up and average latency increases sharply due to HoL blocking.

Second, various CBS product offerings are available to customers. Through our measurement and analysis, different CBS product offerings differ in their capability on competing SA resources since they have significantly different I/O processing pipelines. For example, in Figure 8b, we start a ProductA disk and let it burst in the background, and no matter how much I/O parallelism we add to the ProductB disk, it cannot reach a similar level of throughput to ProductA.

Summary of implications. Based on our observations, we draw a few important implications for designing BurstCBS:

- The performance bottleneck is generally on compute nodes rather than backend servers and devices.
- The burst capability of cloud block storage that we must support is a main trigger of this bottleneck.

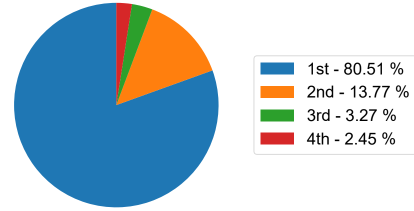
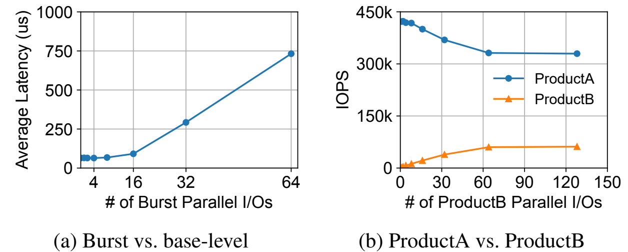


Figure 7: Distribution of I/Os from each vCPU of 4-core VMs (1st is the busiest core, 4th is the least busy core).



(a) Burst vs. base-level

(b) ProductA vs. ProductB

Figure 8: Resource competition on a control thread.

- The root causes are load imbalance among control threads and resource competition within a thread.

4 BurstCBS Overview

BurstCBS is designed and implemented as a standalone system package on xDPU (Figure 9). BurstCBS consists of three key components: high-performance queue scaling, burstable I/O scheduler, and vectorized I/O cost estimator.

High-performance queue scaling (§5.1). We rely on xDPU hardware features to balance I/O distribution among SA control threads. However, it creates an extra challenge for SA control threads to achieve high-performance I/O processing. We propose a two-tier memory pool where BurstCBS moves free buffers between the shared pool and queue-dedicated pools for efficient use of the limited memory.

Burstable I/O scheduler (§5.2). Burst capability with performance isolation requires non-uniform and dynamic resource allocation among multiple co-located tenants. We design a burst-capable I/O scheduler that periodically runs on every SA control thread for resource allocation. It allows each tenant to burst when possible while keeping performance interference among tenants within an acceptable range.

Vectorized I/O cost estimator (§5.3). The key to allocating the right amount of resources to tenants is an accurate estimation of the resource consumption of each I/O. SA manages multiple resources including CPU cycles and interconnect bandwidth. Any of these resources can become the bottleneck under various I/O patterns. We design a vectorized I/O cost estimator that decouples the estimation of each resource.

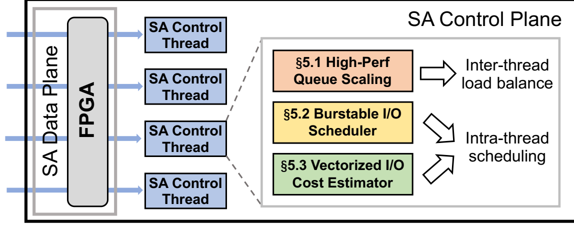
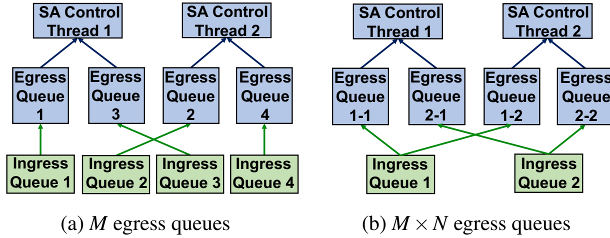


Figure 9: BurstCBS overview.



(a) M egress queues (b) $M \times N$ egress queues

Figure 10: 1 : 1 binding vs. 1 : N binding.

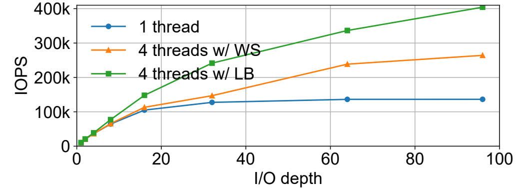


Figure 11: Throughput comparison of different approaches. WS refers to the work stealing prototype, and LB means that each thread receives an equal amount of load.

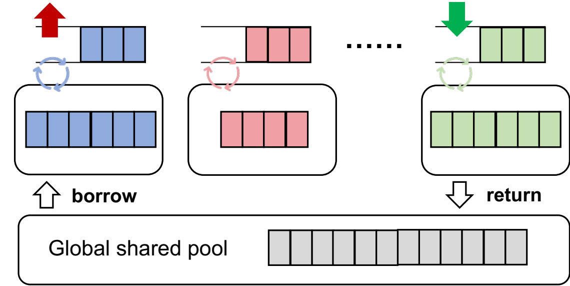


Figure 12: Two-tier memory pool.

5 BurstCBS Design

The goal of BurstCBS is to address the two aforementioned issues: (*i*) lack of load balancing among control threads; (*ii*) lack of resource scheduling among tenants. In this section, we describe the design choices we make for each system module, as well as the considerations behind them.

5.1 High-Performance Queue Scaling

When we design the queue scaling mechanism among threads, there are two key requirements. First, load balancing must be achieved with low overhead to avoid hurting latency and throughput. Second, the mechanism should provide near-perfect load balancing to avoid further thread synchronization.

Today's DPUs are in the early stages and evolving fast. Unlike NICs supporting Receive Side Scaling (RSS) [32], there are no ASIC-based scaling solutions between host and DPU cores. Instead, DPUs provide programmable hardware on the datapath so that users can implement custom logic. For example, NVIDIA BlueField-3 consists of a set of embedded RISC-V cores named datapath accelerator (DPA) [23], and our xDPU has FPGA as the equivalent. FPGA is capable of performing lookup operations at a very high rate, which resembles a programmable switch that controls packets through match-action tables, making it an attractive candidate for offloading logic such as load balancing and rate control [33].

Evolution of load balancing on xDPU. Due to the limitation of FPGA resources on early versions of xDPU, it does not support load balancing by any means. We first make a compromise on the software side by creating one egress queue for every ingress queue, and equally assign the egress queues to all the SA control threads (Figure 10a). Assuming we have many ingress queues (one ingress queue per vCPU) and the throughput of each ingress queue only varies in a narrow range, we expect to see near-perfect load balancing.

This design works relatively well with our non-bursting CBS product types. After we launch burstable disks, we start to observe frequent load imbalance between control threads. Although we can dynamically adjust queue binding, it takes a few seconds to drain inflight I/Os and reconfigure I/O queues, making it impossible to handle transient bursts.

For early versions of xDPU, we have explored two potential software-based approaches to mitigate this issue: (*i*) designating a thread as a centralized dispatcher [34] and (*ii*) allowing idle threads to steal I/Os from others [14, 35]. However, both approaches create additional overhead that we cannot bear. They require intensive messaging between threads, which occupies a significant amount of time on wimpy cores. Figure 11 shows a 35% throughput loss if we switch to a work stealing prototype which we develop using DPDK's lockless ring buffer [36] with a reasonable level of batching.

The newest version of xDPU adds support for load balancing by allowing mapping one ingress queue to multiple egress queues (Figure 10b). We leverage this capability to realize queue scaling to multiple threads. Although no system assumptions are broken with multiple egress queues, it significantly changes how we manage DPU memory.

Two-tier memory pool for fast I/O processing. In the early years of SA development, we kept a shared pool of buffers for I/O processing because DPU memory was very limited. When an I/O arrives, I/O metadata which is required for packet header generation is written to a buffer retrieved from the memory pool. When we take the leap to support mil-

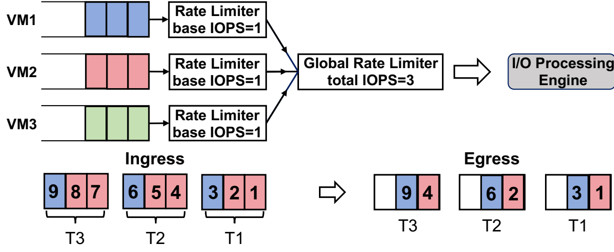


Figure 13: Lack of burst support for BaseCBS.

lions of IOPS per node, we discover that a lot of CPU cycles are wasted on writing disk/queue specific metadata into the buffers. To achieve the best I/O performance (i.e., latency and maximum IOPS), we make a design choice to let each queue maintain its own memory pool, so every memory buffer can be prefilled and will not be overwritten.

Adding extra queues makes each memory pool have even fewer buffers, which means that the number of inflight I/Os a queue can support becomes very limited. Not having enough buffers diminishes the burst maximum we can support for a disk. To remedy this loss, we move to a two-tier memory pool design (Figure 12). Each queue still keeps its own memory pool with a few dedicated buffers, but we add a global shared pool. When a queue experiences increased I/O depth, it acquires extra buffers on demand from the global pool and prefills them with its disk/queue specific metadata. It keeps the extra buffers in its own memory pool and only returns them when the burst terminates. With this design, we avoid slow I/Os caused by repetitive metadata filling to the buffers but keep memory allocation flexible enough.

5.2 Burstable I/O Scheduler

With load imbalance among threads addressed, we next focus on resource scheduling within a thread. There are two essential requirements guiding our design of the scheduler. First, a tenant should be able to use its base-level provisioning (i.e., base-level IOPS and BPS) with bounded latency no matter how other co-located tenants behave. Second, a tenant should be able to burst, but not exceed its burst provisioning. Every tenant should have an equal chance and ability to burst when they share the same burst provisioning and I/O pattern. Note that providing each tenant an equal fraction of the resources is a *non-requirement* for BurstCBS.

Base-level performance guarantee. Achieving base-level performance consists of two implications: (i) a tenant can achieve base-level IOPS/BPS with enough I/O parallelism; (ii) average read/write latency is bounded for a tenant if it is within its base-level IOPS/BPS. The latency guarantee is particularly important because it also determines the maximum throughput that an application can achieve if synchronous system calls are mostly used. In Figure 8a, we show that the latency of a victim tenant starts increasing dramatically with more parallelism of other tenants on a compute node. The reason is that a large amount of queuing delay is added when a hardware/software bottleneck is hit.

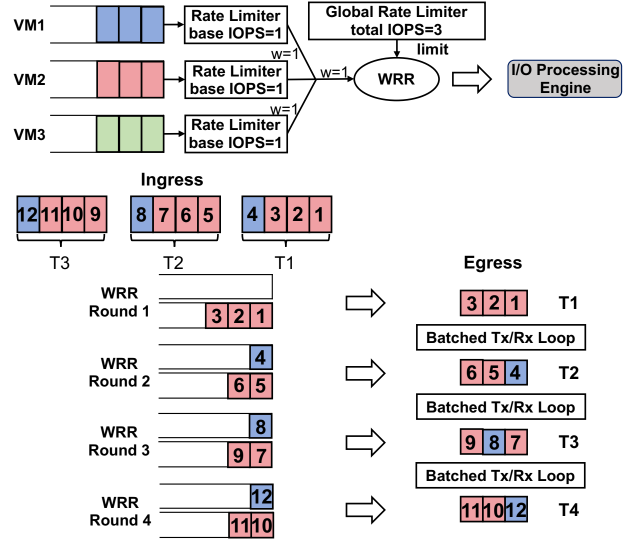


Figure 14: Queuing delay for WildCBS.

These two requirements intuitively translate to a group of rate controllers that limit the admission rate to prevent congestion, which resembles Gimbal [11]. We call this design BaseCBS, which provides performance isolation for base-level performance. There are two major differences between BaseCBS and Gimbal. First, Gimbal strives for black-box SSD congestion avoidance, so it limits the number of inflight I/Os. On the contrary, we try to avoid congestion on DPU. When an inflight I/O is being processed by the backend servers and devices, it does not consume any resources on DPU. Therefore, BaseCBS limits the I/O admission rate instead. Second, Gimbal enforces strict fair sharing of resources among tenants to achieve absolute fairness while we need each tenant to get a share in proportion to its purchased base-level provisioning as a cloud provider.

Bounded burst support. Although BaseCBS provides a strong base-level performance guarantee, it does not provide burst support because each tenant has a static throughput limit. In Figure 13, even if VM3 is completely idle with no incoming I/O, and admitting an additional I/O will not result in congestion, VM2 is only allowed to process one I/O per window. To this end, BaseCBS is only applied to some of the legacy non-burstable CBS disks.

An easy modification that lets this design work is to assign only part of the total resource Res_{base} to BaseCBS while keeping the rest Res_{burst} in a shared pool for potential bursts. However, this design does not work out for CBS products. On the one hand, this design limits the burst capability we can provide on a compute node to Res_{burst}. On the other hand,

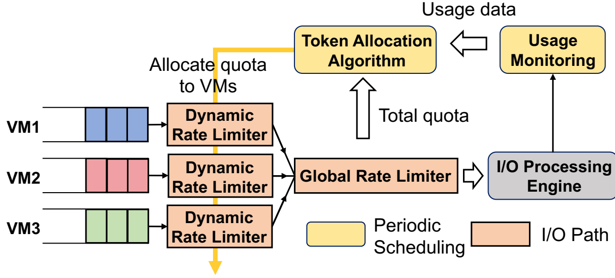


Figure 15: Burstable I/O scheduler workflow.

keeping separate and smaller resource pools directly leads to lower resource utilization, which would force us to provide CBS products at a higher price.

As Figure 4 shows, many of the tenants run way below their base-level IOPS/BPS. An ideal design is to harvest the idle base-level resources to support bursts and return the resources when they are needed. Therefore, bounded burst support essentially requires a work-conserving fair queuing scheduler with a per-tenant rate limiter which enforces provisioning limits (WildCBS). To this end, we use a weighted round-robin (WRR) scheduler, a classic work-conserving fair queuing scheduler that iterates through all ingress queues and processes requests in proportion to their weights.

Figure 14 shows an example of three tenants that are provisioned 1 base IOPS and 3 burst IOPS. 1 base IOPS per tenant is achieved by having a global rate limiter of 3 IOPS and each tenant is weighted equally in WRR scheduling. In this example, VM1 runs at base-level usage, VM2 tries to burst to 3 IOPS, and VM3 is completely idle. Due to the work-conserving nature of WRR, WildCBS is able to fully utilize idle resources from VM3, and the base-level IOPS of VM1 is still guaranteed. WildCBS is integrated into the existing version of SA to enable burst capability in production.

However, a side effect of WildCBS is that a VM that runs at or below base-level usage observes a significantly higher latency while the bursting VM is barely impacted. We observe that if some VMs dispatch I/Os at a much higher rate than others, they can quickly consume all the resource budget, leaving the rest of the VMs to wait until the next window. In Figure 14, VM2 causes all I/Os from a base-level tenant VM1 to delay by a time window. It becomes very common when the bursting VMs employ an extremely high I/O parallelism.

Burstable I/O scheduler. To remedy inadequacies of BaseCBS and WildCBS, we leverage resource usage history to instruct dynamic rate limiting. Figure 15 shows the design of our burstable I/O scheduler (BIOS). BIOS actively collects usage data and allocates resources in proportion to user demands. It provides strong protection on base-level performance by (i) enforcing the total resource allocation limit and (ii) resuming base-level provisioning as soon as it discovers insufficient resource allocation to under-utilizing tenants.

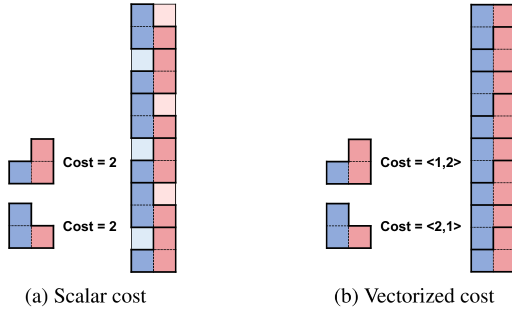
Algorithm 1 BIOS algorithm

1: **procedure** RUN_SCHEDULING( )
2: $unused = total_alloc - reserved$
3: **for** all tenants $i = 1..n$ **do**
4: **if** $status_i$ = burst or $throttle_i > 0$ **then**
5: $alloc_i \leftarrow res_base_i$
6: **else**
7: $alloc_i \leftarrow alloc_hist_i \times \alpha$
8: $unused \leftarrow unused - alloc_i$
9: **for** all tenants $i = 1..n$ **do**
10: $w_i = \frac{usage_i + throttle_i \times weight_throttle}{\sum usage_i + \sum throttle_i \times weight_throttle}$
11: $alloc_i \leftarrow min(alloc_i + unused \times w_i, res_burst_i)$
12: **if** $alloc_i > res_base_i$ **then**
13: burst_tenants.append(tenant$_i$)
14: **function** THROTTLE_IO(tenant$_k$, io)
15: **if** $alloc_k \geq$cost(io) **then**
16: $alloc_k \leftarrow alloc_k -$cost(io)
17: **else**
18: tenant$_b$ ←power_of_two_choices(burst_tenants)
19: **if** $alloc_b \geq$cost(io) **then**
20: $alloc_b \leftarrow alloc_b -$cost(io)
21: **else if** reserved≥cost(io) **then**
22: reserved←reserved−cost(io)
23: **else**
24: **return** true
25: **return** false

Algorithm 1 depicts how resources are allocated. The algorithm first allocates base-level provisioning to all tenants unless they were below base-level and did not consume all resources in the last window (lines 4-5). Otherwise, the algorithm tries to lower their allocated resources to allow others to burst (line 7). In the second round of allocation, it allocates the remaining resources in proportion to the weighted sum of consumed and throttled I/Os in the last window, but it cannot exceed burst provisioning (lines 10-11). We add a higher weight to throttled I/Os because they may have more follow-up I/Os. The running time of this algorithm only increases linearly with the number of tenants. This is important because (i) there is not enough headroom on wimpy cores for a complex algorithm, and (ii) we may have hundreds of tenants running on the same compute node in extreme cases.

Fast base-level performance recovery. A scheduling algorithm that relies on historical statistics needs to tolerate bad predictions. The consequence of harvesting idle base-level resources is that if a tenant suddenly starts dispatching I/Os after being idle for a long time, it may observe high queuing delay and inadequate IOPS, which breaks our commitment on base-level performance guarantee.

A fast recovery mechanism is added as compensation before the algorithm catches its mis-prediction in the next scheduling cycle. As shown in Algorithm 1 lines 18-20, a base-level tenant that runs out of resources first tries to re-



(a) Scalar cost (b) Vectorized cost

Figure 16: Scalar cost vs. vectorized cost.

claim the extra resources from bursting tenants. Ideally, we should reclaim resources from the tenant with the most remaining resources, which involves sorting all bursting tenants. However, applying sorting on a per-I/O basis hurts I/O performance significantly. We adopt power of two choices as an alternative to eliminate sorting on the I/O processing path.

This mechanism could fail to take effect if bursting tenants quickly consume all the allocated resources, leaving nothing left for a base-level tenant to reclaim. Therefore, we add another layer of protection of a shared resource pool (Algorithm 1 lines 21-22). The intuition behind this optimization is that, through our online measurements, it is unlikely multiple base-level tenants would resume their usage at the same time. So we reserve a small amount of pooled resources that are just enough for two tenants to resume base-level provisioning. It diminishes a little of burst maximum we can support, but greatly helps us guarantee base-level performance.

5.3 Vectorized I/O Cost Estimator

BIOS is a proper framework for serving a mixture of base-level and burst tenants. A key assumption of BIOS is that consumed resource per I/O is known. However, IOPS and BPS that an SA control thread on DPU can saturate are dynamic subject to hardware specification, software implementation, system configuration, and I/O pattern. To avoid overloading SA, it is necessary to estimate the I/O cost accurately. Previous storage systems focus on I/O cost estimation of SSDs [10, 11, 17]. Because manufacturers of SSDs reveal limited design details of their products, existing work estimates SSD I/O cost by profiling each device with synthetic workloads.

For CBS, the bottlenecks are on xDPU and SA, rather than SSDs. Based on our online measurement, we identify three major bottlenecks in xDPU and SA. (*i*) CPU: it takes a few microseconds to process an I/O on an SA control thread on average, so a control thread can handle a few hundreds of thousands of IOs per second at maximum. (*ii*) Interconnect: SA is able to use 100Gbps NIC bandwidth and ~100Gbps DMA bandwidth. (*iii*) Software: the read/write bandwidth of SA is also constrained by a software rate limiter which protects other non-storage services from resource starvation.

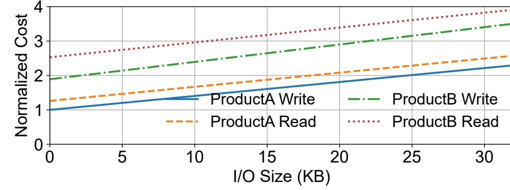


Figure 17: Normalized CPU cost estimation of four I/O types.

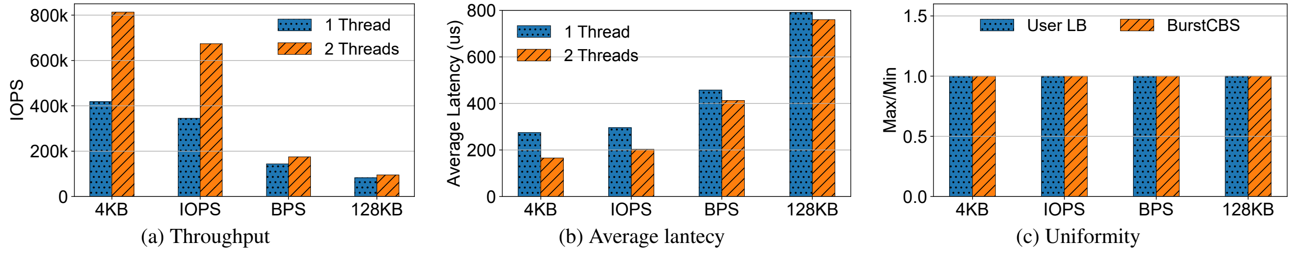
Table 3: Effectiveness of estimation adjustment.

	WildCBS	**BurstCBS w/o adjustment**	**BurstCBS with adjustment**
Read Lat (us)	1096.89	848.57	248.97
Write Lat (us)	868.79	749.57	270.49

Furthermore, the relative ratios across consumed resources are divergent on a per I/O basis. Our experiments show that a 4KB write I/O consumes 8× higher CPU time per byte compared to a 128KB write I/O, while they always consume the same egress bandwidth per byte. Therefore, the bottlenecks are CPU and egress bandwidth for 4KB and 128KB write I/Os respectively. This result implies that it is necessary to model the cost of different I/O types independently.

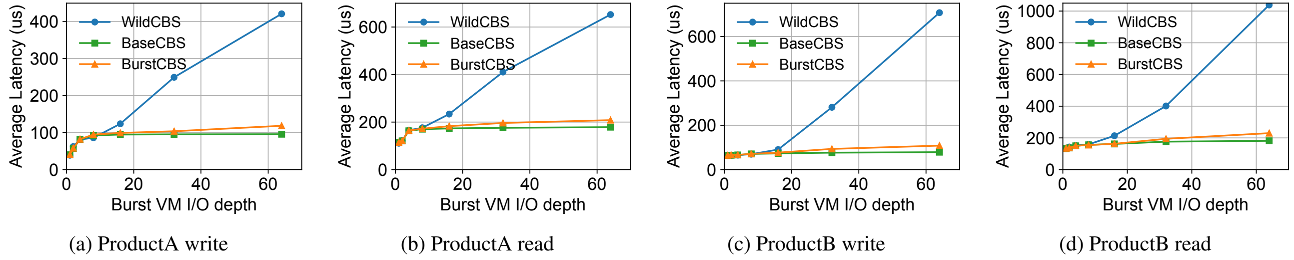
If we describe I/O cost as a scalar, it creates resource underutilization. In Figure 16, a scalar cost must be given the value of the most consumed resource, which unnecessarily leaves other resources idle, limiting our ability to burst. In contrast, if we decouple the costs of different resource types, higher resource utilization can be achieved without breaching the latency target. Therefore, we describe the cost of an I/O as a vector of 4 dimensions: *CPU time*, *ingress*, *egress*, and *software limit*. Ingress, egress, and software limits are shared among all threads. We simply divide the global limit by the number of threads to get the per-thread limit. Out of them, only CPU time requires profiling. We derive the CPU time of (product_type, rw, size) tuple from the maximum IOPS it can achieve. We only profile I/O sizes from 4KB to 16KB, because the bottleneck is no longer on CPU beyond 16KB. We fit the observed values into a linear model so that we can estimate CPU time for all sizes. Figure 17 shows the normalized estimation of different I/O types on the newest xDPU. ProductA and ProductB are both burstable CBS product classes. Compared to ProductB, ProductA is further optimized for higher throughput and lower latency by adopting advanced hardware features and optimized software implementation.

Unpredictable misestimation handling. Although vectorized cost estimator accurately reflects I/O cost in common cases, there are many circumstances that we cannot foresee and integrate into our cost model in advance. A typical misestimation happens when SA takes a different path for I/O processing when the FPGA of xDPU experiences transient hardware failures. During uncommon failures, SA enters a



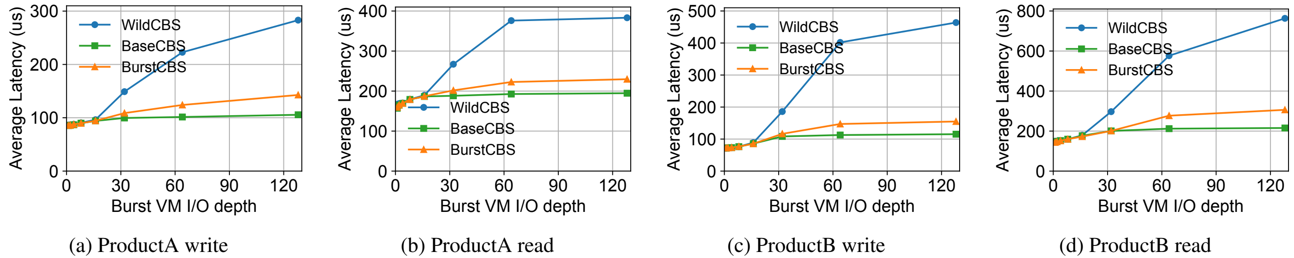
(a) Throughput (b) Average lantecy (c) Uniformity

Figure 18: Effectiveness and overhead of FPGA-based load balancing with high-performance queue scaling.



(a) ProductA write (b) ProductA read (c) ProductB write (d) ProductB read

Figure 19: I/O latency (I/O depth=1) with a BPS-intensive burst VM in the background.



(a) ProductA write (b) ProductA read (c) ProductB write (d) ProductB read

Figure 20: I/O latency (I/O depth=1) with an IOPS-intensive burst VM in the background.

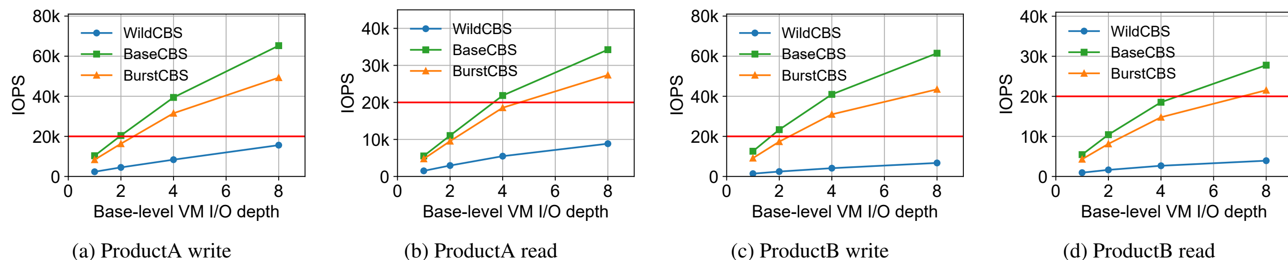
heavy error handling branch which retries failed I/Os multiple times until they reach the configured timeout and generates error logs for further investigation. Doing so at a per I/O basis consumes more CPU resources and bandwidth than a normal code path usually does. Based on our observation, the amortized cost per successful I/O can be doubled when SA experiences such unexpected failures.

To alleviate this impact, we introduce a delay-based cost adjustment mechanism. BurstCBS is anchored to a target delay. When the target delay is breached, we gradually increase the cost. And we reduce the cost when the delay drops below the target. Note that backend time is excluded from this delay. The reason is that SSD is notorious for its high tail latency [37–39], which may wrongly trigger the cost adjustment mechanism. To allow the cost adjustment mechanism to react quickly with only a few data points, irrelevant outliers should be avoided. In Table 3, we show a case that SA keeps detecting FPGA failures. BurstCBS without the cost adjustment mechanism mis-estimates I/O cost and admits more I/Os than its capacity, which results in high latency on the base-level tenant. Adding cost estimation adjustment significantly reduces the latency by admitting the right amount of I/Os.

6 Evaluation

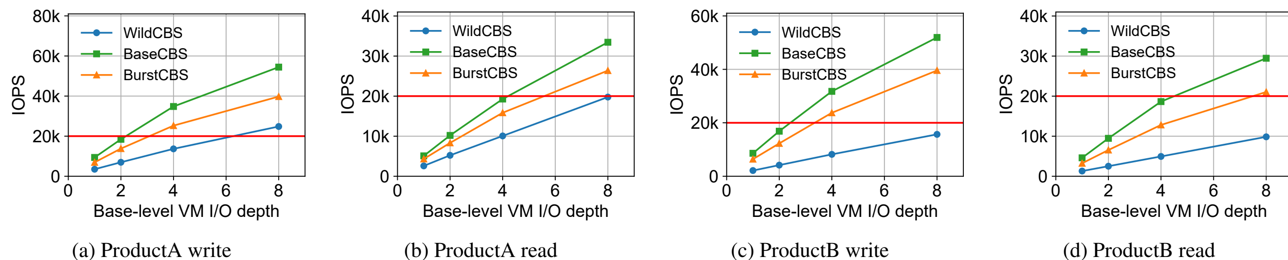
In this section, we evaluate the performance of BurstCBS. Our main baseline is WildCBS which combines WRR scheduling and per-tenant rate limiters. Currently, WildCBS is the most widely deployed version in our production clusters. We also compare BurstCBS with BaseCBS which is a variant of Gimbal and provides strong performance isolation between tenants. All of the experiments are conducted on a compute node with the newest version of xDPU.

Our experiments run two different FIO [40] workloads which we typically use at Alibaba Cloud to make sure our products can adapt to different usage patterns. One is IOPS-intensive and contains a mix of 4KB-16KB I/Os. This workload consists of small I/Os which resembles the I/O pattern of many transactional databases. The other is BPS-intensive



(a) ProductA write (b) ProductA read (c) ProductB write (d) ProductB read

Figure 21: IOPS with a BPS-intensive burst VM (I/O depth=64) in the background.



(a) ProductA write (b) ProductA read (c) ProductB write (d) ProductB read

Figure 22: IOPS with a IOPS-intensive burst VM (I/O depth=128) in the background.

and contains a mix of 4KB-128KB I/Os. Most of the I/Os are within this range in our production environment. We include both ProductA and ProductB I/Os in each workload.

We mainly evaluate three aspects of BurstCBS: thread load balancing with high-performance queue scaling (§6.1), latency (§6.2) and IOPS (§6.3) of base-level tenants with bursting neighbors, and overall resource utilization at burst (§6.4). We also evaluate effectiveness of the fast base-level throughput recovery mechanism (§6.5) and the scalability of BIOS (§6.6). At last, we perform a set of database experiments to evaluate how BurstCBS performs under real use cases (§6.7), and collect results from a node that serves an internal database service (§6.8).

6.1 Inter-thread Load Balancing

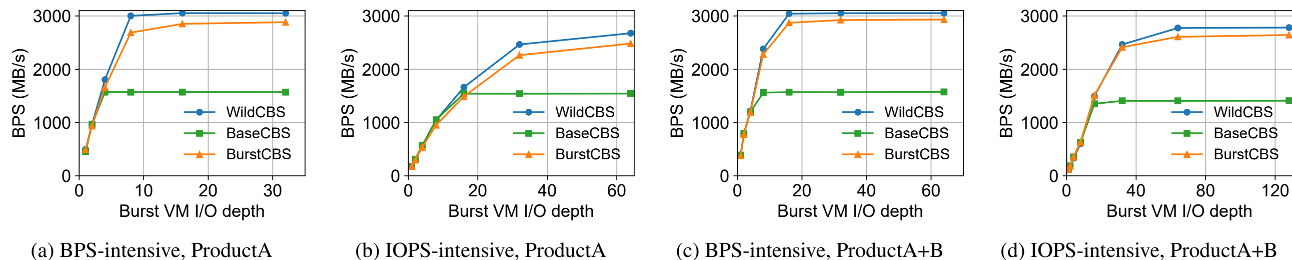
We first evaluate inter-thread load balancing to understand the effectiveness and overhead of FPGA-based load balancing with high-performance queue scaling. Figure 18a shows the maximum write IOPS that a tenant can achieve with one or two SA control threads behind the FPGA load balancer. We observe near-linear scaling on pure 4KB workload and IOPS-intensive workload because the bottleneck is on the SA control threads which run on CPU cores when a large amount of small I/Os are being processed. Scaling on BPS-intensive workload and pure 128KB workload is limited because the NIC is already congested while the SA control threads are not fully occupied. In Figure 18b, we repeat the experiment with I/O depth unchanged and compare average latency, from which we can draw a similar conclusion.

To show how well the load is balanced among multiple threads, in Figure 18c, we compare BurstCBS with the case that a user intentionally dispatches I/Os equally to each ingress queue (by using multiple vCPUs which are mapped to different queues) through FIO. We start six SA control threads in total and take the ratio of maximum to minimum values of throughput across threads to reflect uniformity. The results demonstrate that they are almost equivalent in terms of inter-thread load balancing.

6.2 Base-level Tenant Latency

A main goal of BurstCBS is to keep average latency under SLO for tenants running below their base-level IOPS/BPS. In this experiment, we run an I/O stream with different I/O depths from a background VM. While the background VM is running, we start a 4KB I/O stream of depth 1 on the victim VM and observe its average latency. In Figure 19, we use a read/write BPS-intensive I/O workload in the background VM with I/O depths from 1 to 64, and we record the average latency of four types of I/Os. We evaluate both read and write I/Os from two burstable CBS product classes, ProductA and ProductB. With the current implementation, ProductA is faster than ProductB, and write is faster than read. Overall, average latency is reduced by 68%–85% compared to WildCBS, and is very close to BaseCBS which shows the ideal latency we can possibly achieve with strong isolation.

Figure 20 shows the results on the IOPS-intensive workload in the background. The latency reduction ranges from 40% to 66% which is slightly less than that of the BPS-intensive workload. This difference is because the IOPS-intensive work-



(a) BPS-intensive, ProductA (b) IOPS-intensive, ProductA (c) BPS-intensive, ProductA+B (d) IOPS-intensive, ProductA+B

Figure 23: Overall resource utilization during a burst.

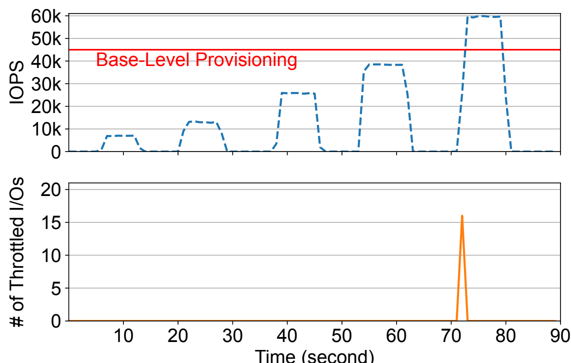


Figure 24: Responsiveness to sudden tenant activation.

load shifts bottleneck to the CPU cores. When the CPU cores are congested, the performance gap between BaseCBS and WildCBS is not as significant.

6.3 Base-level Tenant Throughput

Another critical requirement for BurstCBS is that a tenant should be able to reach its base-level IOPS with a relatively small I/O depth. In this experiment, we set the base-level IOPS of the victim VM to 20k, and we expect it to reach that IOPS on a 4KB I/O stream within I/O depth 8. We again validate the effectiveness of BurstCBS on both BPS-intensive workload and IOPS-intensive workload with all four different I/O types. Figure 21 and Figure 22 show that BurstCBS can achieve the desired IOPS for all the cases, while WildCBS at I/O depth 8 fails to meet our goal for seven out of eight cases, and the IOPS is as low as 4,000. Similar to §6.2, there is a smaller performance gap between BaseCBS and WildCBS on the IOPS-intensive workload, which leaves limited space for us to optimize. And ProductB read I/O is the most costly operation out of the four I/O types, so BurstCBS barely reaches 20k IOPS in Figure 22d, which is the lowest.

6.4 Burst Resource Utilization

Although protecting base-level performance is our first priority, we also seek for high resource utilization during bursts. In

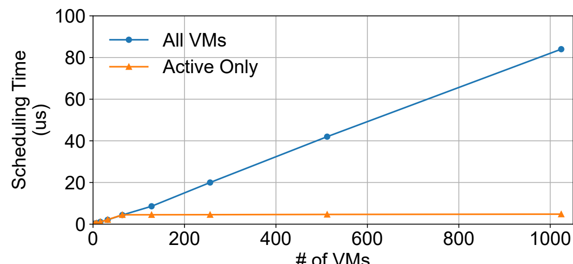


Figure 25: Scheduler scalability with number of VMs.

this experiment, the available bandwidth is limited to 3GB/s, which is the maximum that an SA control thread can handle. We start two VMs and let one VM burst as much as possible while keeping the other VM running at I/O depth 2 which produces an I/O stream below its base-level provisioning. In Figure 23, the results demonstrate that BurstCBS loses about 5%–8% throughput compared to WildCBS, which meets our expectation because we keep 5% of the total resources in the shared pool for fast recovery. BaseCBS enforces fair resource distribution at all times, which limits the resources that the burst VM can use to half of the limit. Note that the maximum bandwidth cannot be achieved with the IOPS-intensive workload due to the extra CPU overhead incurred by small I/Os. And mixing ProductA and ProductB I/Os slightly improves resource utilization because ProductA and ProductB use different polling loops and idle loops waste CPU cycles.

6.5 Responsiveness to Sudden Activation

We next evaluate the fast base-level provisioning recovery mechanism. A tenant should be able to recover its base-level provisioning seamlessly even when the resources are lent to other tenants. In this experiment, we create a VM with base-level provisioning of 45k IOPS. We run I/O streams of depths 1-16 on this VM. Before we run each stream, we let the VM stay idle for a few seconds to make sure that its resource allocation drops to zero. When we run an I/O stream, the metric we record is the number of throttled I/Os. We expect to see no throttled I/Os when an I/O stream is within the base-level provisioning. In Figure 24, we do not observe any

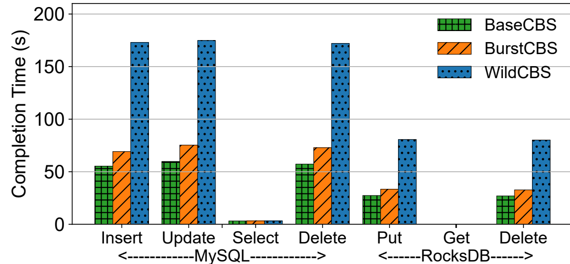


Figure 26: Completion time of 100k DB operations.

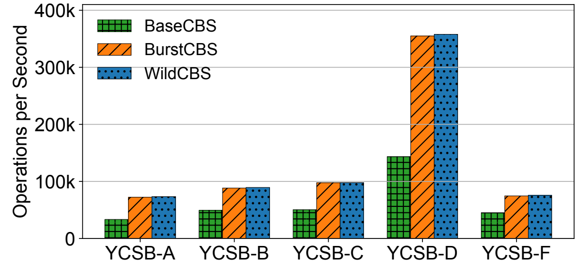


Figure 27: YCSB throughput.

throttled I/Os until we kick off a 60k IOPS stream which already makes the VM a burst tenant.

6.6 Scalability

Scalability of the scheduler is also our concern because if the periodic scheduler runs for too long, successive I/Os will experience high latency and it will also hurt overall throughput by occupying too many CPU cycles. In this experiment, we solely run the scheduler on xDPU and vary the number of VMs from 4 to 1024. 1024 is the theoretical maximum number of VMs on a compute node in any near future. In Figure 25, the scheduling time increases linearly with the number of VMs and is always below 100μs. We introduce an additional optimization that removes a VM from scheduling after it becomes idle for a while. We run the experiment again with 64 active VMs at maximum. It takes less than 5μs to run the scheduler once no matter how many VMs are there in total.

6.7 Application Performance

Transactional databases are one of the targeted use cases of burstable CBS. We evaluate both a SQL database (MySQL [41]) and a NoSQL database (RocksDB [42]) to validate its improvement on real-world use cases. Following our production specification, we provision 16 I/O-burstable VMs with 100k base-level IOPS and 200k burst IOPS on a compute node. The corresponding base-level BPS and burst BPS we provision are 1400MB/s and 2800MB/s respectively. Each VM has 8 vCPUs, 16GB memory, a 40GB ProductB virtual disk as the OS disk, and another 1TB ProductB virtual disk as the database data disk.

We first evaluate the latency of DB operations. For the MySQL experiment, we install MySQL 8.0 on one of the VMs. We develop a simple C program that connects to the MySQL database, executes each type of operation 100k times, and records the execution time. For RocksDB, we write a C++ program and again run put/get/delete 100k times with the sync flag enabled to force an immediate disk I/O per write request. When we execute the programs, we let the other 15 VMs burst by starting a BPS-intensive workload of I/O depth 32 on each of them. In Figure 26, for all write operations (insert, update, put, and delete), we observe about 60% latency reduction. Databases extensively use cache for read operations (select and get). Therefore, we do not observe significant improvement on read operations in this experiment.

We next evaluate the performance improvement brought by burst capability. In this experiment, we focus on YCSB [43] over RocksDB which creates higher throughput than MySQL with the same compute power. We leave the other 15 VMs idle and set up eight RocksDB instances to better leverage burst capability. For each DB instance, we create 10 million 1KB entries and execute 1 million operations. As shown in Figure 27, 2$\times$ burst on BurstCBS results in 1.7$\times$-2.5$\times$ throughput improvement on various workloads over BaseCBS, and the results are close to those of WildCBS.

6.8 Practical Benefits

We last confirm that BurstCBS improves our database user experience. Here, we deploy BurstCBS to a production compute node that serves the internal Relational Database Service (RDS) and evaluate performance. RDS creates VM instances with CBS disks and manages databases on the VMs for users. A typical RDS VM instance comes with two data disks: one ProductB disk as the main storage, and one ProductA disk as a buffer pool extension. Previously, RDS has noticed neighbor interference and informed us.

In this experiment, an RDS VM instance is started and preloaded with 100 tables of 10 million rows. Another VM is used to connect to the RDS instance through a virtual private network and run single-threaded sysbench [44]. In Figure 28, 8 or 16 burst tenants of I/O depth 32 run in the background. The distribution of I/O sizes and the read/write ratio follow the same pattern in production [7]. The results show that average query latency is reduced by up to 83%. Note that RDS read operations can also trigger disk writes because the ProductA disk is used as a buffer pool extension.

This experiment is also run against a more fluctuant trace for 30 minutes. A 30-minute second-scale monitoring history of a burst disk in production is amplified to the scale of 10GB/s to generate the I/O trace. The trace is replayed in the background, and sysbench is used to record average query latency. In Figure 29, while WildCBS creates latency spikes of 20-50ms, BurstCBS is able to keep it under 10ms.

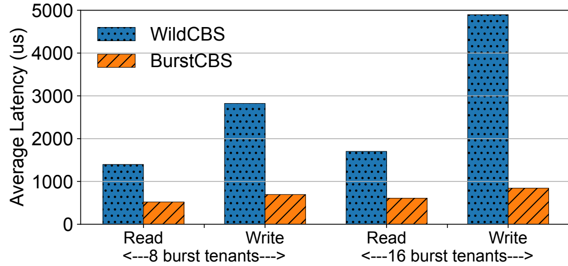


Figure 28: RDS query latency with burst neighbors.

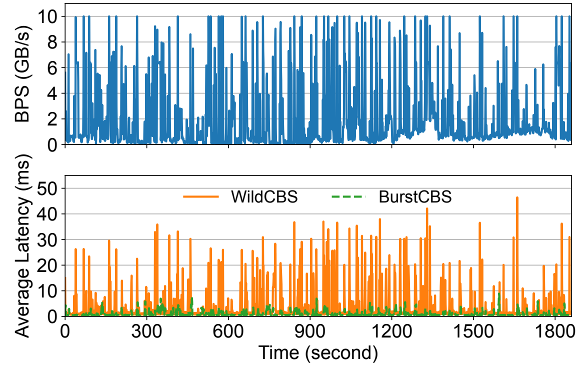


Figure 29: RDS query latency with replayed production background traffic.

7 Discussion

We discuss a number of future improvements on our roadmap that will further enhance BurstCBS.

Co-optimization with user OS kernel. Rate control at SA requires a significant amount of DPU memory to buffer the requests and may eventually cause out-of-memory. Limiting I/O rate on the user side can mitigate this issue. Furthermore, newer Linux kernel supports NVMe WRR [45], a feature that allows a user to prioritize certain I/Os, which helps protect the performance of critical I/Os when SA is congested.

Automated cost profiling. We currently maintain three different versions of xDPU and many more system configurations in our production environment. It brings a heavy operational burden if we need to manually profile I/O cost every time we make a software/configuration update. Instead, we are developing an automated I/O cost profiler which profiles I/O cost offline at system bootstrap and adjusts cost adaptively online.

Inter-server scheduling. The best way to handle congestion is always to avoid it in the first place. Once we detect multiple co-located VMs often burst at the same time, we can signal the control plane of VM instances to migrate them when possible. Because the time to migrate a VM ranges from a few seconds to several minutes and it causes temporary unavailability, we still rely on BurstCBS to handle short-term congestion.

8 Related Work

Cloud storage systems. There is a large body of work on cloud storage systems [5–9, 16, 17, 46–52]. Despite the different interfaces these systems expose (e.g., block store, object store), most of them are distributed systems to meet the scale of cloud. Existing research mainly focuses on the backend system design. Tectonic [51] and Pangu [52] provide unified backend storage for a large number of tenants and different use cases. To overcome the inefficiency of traditional in-kernel network stacks, RDMA [5, 6, 48] and other kernel bypassing network stacks [7, 8] are deployed. BurstCBS instead focuses on burst capability support of cloud block storage. We reveal and tackle the major challenges to achieve both extreme burst and base-level performance protection.

Resource sharing for storage systems. Previous work has explored how to achieve work-conserving scheduling and/or fair sharing for storage systems [11,17,53–62]. Many of them also involve estimating per I/O cost for efficient resource scheduling. However, previous work mainly targets the bottleneck of storage media (e.g., HDD and SSD), while the bottleneck we encounter is on DPU. A highly related system is Gimbal [11] which designs a fair queuing scheduler for performance isolation on the server side of disaggregated storage with DPUs. BurstCBS distinguishes itself by supporting dynamic bursts and addressing unique challenges (i.e., load imbalance and cost estimation) on client-side DPUs.

9 Conclusion

This paper presents BurstCBS, a hardware-software co-designed storage I/O scheduling system that achieves inter-thread load balancing and intra-thread resource scheduling. BurstCBS applies three techniques: a high-performance queue scaling mechanism, a burstable I/O scheduler, and a vectorized I/O cost estimator. We implement and evaluate BurstCBS on xDPU-based servers. We show that BurstCBS provides base-level performance protection while allowing tenants to burst as much as possible.

Acknowledgments. We thank our shepherd Geoffrey M. Voelker and the anonymous reviewers for their valuable feedback. We also thank Erci Xu for his precious comments. This work was supported by the National Key Research and Development Program of China under the grant number 2022YFB4500700, National Natural Science Foundation of China under the grant numbers 62172008 and 62325201, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), and Alibaba Cloud through Alibaba Research Intern Program. Xin Jin is the corresponding author. Junyi Shu, Xuanzhe Liu and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] "Amazon EBS features." https://aws.amazon.com/ebs/features/, 2023. Retrieved Nov 15, 2023.

[2] "Azure managed disk types." https://learn.microsoft.com/en-us/azure/virtual-machines/disks-types/, 2023. Retrieved Nov 15, 2023.

[3] "Google storage options." https://cloud.google.com/compute/docs/disks, 2023. Retrieved Nov 15, 2023.

[4] "Alibaba Cloud ECS Limits." https://www.alibabacloud.com/help/en/ecs/product-overview/limits, 2023. Retrieved Nov 15, 2023.

[5] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, R. Chow, J. Cohen, M. Elhaddad, V. Ette, I. Figlin, D. Firestone, M. George, I. German, L. Ghai, E. Green, A. Greenberg, M. Gupta, R. Haagens, M. Hendel, R. Howlader, N. John, J. Johnstone, T. Jolly, G. Kramer, D. Kruse, A. Kumar, E. Lan, I. Lee, A. Levy, M. Lipshteyn, X. Liu, C. Liu, G. Lu, Y. Lu, X. Lu, V. Makhervaks, U. Malashanka, D. A. Maltz, I. Marinos, R. Mehta, S. Murthi, A. Namdhari, A. Ogus, J. Padhye, M. Pandya, D. Phillips, A. Power, S. Puri, S. Raindel, J. Rhee, A. Russo, M. Sah, A. Sheriff, C. Sparacino, A. Srivastava, W. Sun, N. Swanson, F. Tian, L. Tomczyk, V. Vadlamuri, A. Wolman, Y. Xie, J. Yom, L. Yuan, Y. Zhang, and B. Zill, "Empowering Azure Storage with RDMA," in *USENIX NSDI*, 2023.

[6] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu, "When Cloud Storage Meets RDMA," in *USENIX NSDI*, 2021.

[7] R. Miao, L. Zhu, S. Ma, K. Qian, S. Zhuang, B. Li, S. Cheng, J. Gao, Y. Zhuang, P. Zhang, R. Liu, C. Shi, B. Fu, J. Zhu, J. Wu, D. Cai, and H. H. Liu, "From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud," in *ACM SIGCOMM*, 2022.

[8] L. Zhu, Y. Shen, E. Xu, B. Shi, T. Fu, S. Ma, S. Chen, Z. Wang, H. Wu, X. Liao, Z. Yang, Z. Chen, W. Lin, Y. Hou, R. Liu, C. Shi, J. Zhu, and J. Wu, "Deploying User-space TCP at Cloud Scale with LUNA," in *USENIX ATC*, 2023.

[9] W. Zhang, E. Xu, Q. Wang, X. Zhang, Y. Gu, Z. Lu, T. Ouyang, G. Dai, W. Peng, Z. Xu, S. Zhang, D. Wu, Y. Peng, T. Wang, H. Zhang, J. Wang, W. Yan, Y. Dong, W. Yao, Z. Wu, L. Zhu, C. Shi, Y. Wang, R. Liu, J. Wu, J. Zhu, and J. Wu, "What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store," in *USENIX FAST*, 2024.

[10] A. Klimovic, H. Litz, and C. Kozyrakis, "ReFlex: Remote Flash ≈ Local Flash," in *ACM ASPLOS*, 2017.

[11] J. Min, M. Liu, T. Chugh, C. Zhao, A. Wei, I. H. Doh, and A. Krishnamurthy, "Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs," in *ACM SIGCOMM*, 2021.

[12] J. Shu, R. Zhu, Y. Ma, G. Huang, H. Mei, X. Liu, and X. Jin, "Disaggregated RAID Storage in Modern Datacenters," in *ACM ASPLOS*, 2023.

[13] J. Hwang, Q. Cai, A. Tang, and R. Agarwal, "TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10," in *USENIX NSDI*, 2020.

[14] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal, "Rearchitecting Linux Storage Stack for µs Latency and High Throughput," in *USENIX OSDI*, 2021.

[15] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash Storage Disaggregation," in *EuroSys*, 2016.

[16] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes, "Building An Elastic Query Engine on Disaggregated Storage," in *USENIX NSDI*, 2020.

[17] T. Heo, D. Schatzberg, A. Newell, S. Liu, S. Dhakshinamurthy, I. Narayanan, J. Bacik, C. Mason, C. Tang, and D. Skarlatos, "IOCost: Block IO Control for Containers in Datacenters," in *ACM ASPLOS*, 2022.

[18] "Understanding Burst vs. Baseline Performance with Amazon RDS and GP2." https://aws.amazon.com/blogs/database/understanding-burst-vs-baseline-performance-with-amazon-rds-and-gp2/, 2023. Retrieved Nov 15, 2023.

[19] "Azure managed disk bursting." https://learn.microsoft.com/en-us/azure/virtual-machines/disk-bursting, 2023. Retrieved Nov 15, 2023.

[20] "Alibaba Cloud block storage performance." https://www.alibabacloud.com/help/en/ecs/user-guide/block-storage-performance, 2023. Retrieved Nov 15, 2023.

[21] "AWS Nitro System." https://aws.amazon.com/ec2/nitro/, 2023. Retrieved Nov 15, 2023.

[22] "A Detailed Explanation about Alibaba Cloud CIPU." https://www.alibabacloud.com/blog/a-detailed-explanation-about-alibaba-cloud-cipu_599183, 2023. Retrieved Nov 15, 2023.

[23] "Nvidia BlueField-3." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2023. Retrieved Nov 15, 2023.

[24] "Intel Infrastructure Processing Unit." https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html, 2023. Retrieved Nov 15, 2023.

[25] "Marvell LiquidIO 3." https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf, 2023. Retrieved Nov 15, 2023.

[26] "AWS burstable performance instances." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html, 2023. Retrieved Nov 15, 2023.

[27] "Azure B-series burstable virtual machine sizes." https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable, 2023. Retrieved Nov 15, 2023.

[28] "Google Shared-core VMs." https://cloud.google.com/compute/docs/general-purpose-machines#sharedcore, 2023. Retrieved Nov 15, 2023.

[29] "Alibaba Cloud Burstable Instances." https://www.alibabacloud.com/help/en/ecs/user-guide/overview-5, 2023. Retrieved Nov 15, 2023.

[30] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms," in *ACM ASPLOS*, 2023.

[31] "DB-Engines Ranking." https://db-engines.com/en/ranking, 2023. Retrieved Nov 15, 2023.

[32] "Introduction to Receive Side Scaling." https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, 2022. Retrieved Nov 15, 2023.

[33] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo, "Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing," in *USENIX NSDI*, 2022.

[34] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency," in *USENIX NSDI*, 2019.

[35] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks," in *ACM SOSP*, 2017.

[36] "DPDK Ring Library." https://doc.dpdk.org/guides/prog_guide/ring_lib.html, 2023. Retrieved Nov 15, 2023.

[37] H. Li, M. L. Putra, R. Shi, X. Lin, G. R. Ganger, and H. S. Gunawi, "IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage," in *ACM SOSP*, 2021.

[38] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," *ACM Transactions on Storage*, 2017.

[39] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments," in *USENIX FAST*, 2016.

[40] "Flexible I/O Tester." https://github.com/axboe/fio, 2023. Retrieved Nov 15, 2023.

[41] "MySQL." https://www.mysql.com/, 2023. Retrieved Nov 15, 2023.

[42] "RocksDB." https://rocksdb.org/, 2023. Retrieved Nov 15, 2023.

[43] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *ACM Symposium on Cloud Computing*, 2010.

[44] "sysbench." https://github.com/akopytov/sysbench, 2024. Retrieved Apr 15, 2024.

[45] K. Joshi, K. Yadav, and P. Choudhary, "Enabling NVMe WRR support in Linux Block Layer," in *USENIX HotStorage Workshop*, 2017.

[46] R. Lu, E. Xu, Y. Zhang, F. Zhu, Z. Zhu, M. Wang, Z. Zhu, G. Xue, J. Shu, M. Li, and J. Wu, "Perseus: A Fail-Slow Detection Framework for Cloud Storage Systems," in *USENIX FAST*, 2023.

[47] S. Zhou, E. Xu, H. Wu, Y. Du, J. Cui, W. Fu, C. Liu, Y. Wang, W. Wang, S. Sun, X. Wang, B. Feng, B. Zhu, X. Tong, W. Kong, L. Liu, Z. Wu, J. Wu, Q. Luo, and J. Wu, "SMRSTORE: A Storage Engine for Cloud Object Storage on HM-SMR Drives," in *USENIX FAST*, 2023.

[48] Q. Li, Y. Gao, X. Wang, H. Qiu, Y. Le, D. Liu, Q. Xiang, F. Feng, P. Zhang, B. Li, J. Dong, L. Tang, H. H. Liu, S. Liu, W. Li, R. Miao, Y. Wu, Z. Wu, C. Han, L. Yan, Z. Cao, Z. Wu, C. Tian, G. Chen, D. Cai, J. Wu, J. Zhu, J. Wu, and J. Shu, "Flor: An Open High Performance RDMA Framework Over Heterogeneous RNICs," in *USENIX OSDI*, 2023.

[49] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield, "Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3," in *ACM SOSP*, 2021.

[50] Q. Li, L. Chen, X. Wang, S. Huang, Q. Xiang, Y. Dong, W. Yao, M. Huang, P. Yang, S. Liu, Z. Zhu, H. Wang, H. Qiu, D. Liu, S. Liu, Y. Zhou, Y. Wu, Z. Wu, S. Gao, C. Han, Z. Luo, Y. Shao, G. Tian, Z. Wu, Z. Cao, J. Wu, J. Shu, J. Wu, and J. Wu, "Fisc: A Large-scale Cloud-native-oriented File System," in *USENIX FAST*, 2023.

[51] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, S. S. P, M. Shuey, R. Wareing, M. Gangapuram, G. Cao, C. Preseau, P. Singh, K. Patiejunas, J. Tipton, E. Katz-Bassett, and W. Lloyd, "Facebook's Tectonic Filesystem: Efficiency from Exascale," in *USENIX FAST*, 2021.

[52] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu, H. Wang, S. Liu, L. Chen, Z. Wu, H. Qiu, D. Liu, G. Tian, C. Han, S. Liu, Y. Wu, Z. Luo, Y. Shao, J. Wu, Z. Cao, Z. Wu, J. Zhu, J. Wu, J. Shu, and J. Wu, "More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba," in *USENIX FAST*, 2023.

[53] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional Allocation of Resources for Distributed Storage Access," in *USENIX FAST*, 2009.

[54] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, "Multi-Queue Fair Queuing," in *USENIX ATC*, 2019.

[55] H. Lu, B. Saltaformaggio, R. Kompella, and D. Xu, "VFair: Latency-Aware Fair Storage Scheduling via per-IO Cost-Based Differentiation," in *ACM Symposium on Cloud Computing*, 2015.

[56] M. Nanavati, J. Wires, and A. Warfield, "Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage," in *USENIX NSDI*, 2017.

[57] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler," in *USENIX FAST*, 2012.

[58] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs," in *USENIX ATC*, 2013.

[59] D. Shue and M. J. Freedman, "From Application Requests to Virtual IOPs: Provisioned Key-Value Storage with Libra," in *EuroSys*, 2014.

[60] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A Software-Defined Storage Architecture," in *ACM SOSP*, 2013.

[61] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance insulation for shared storage servers," in *USENIX FAST*, 2007.

[62] J. Min, C. Zhao, M. Liu, and A. Krishnamurthy, "eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs," in *USENIX OSDI*, 2023.