Journal of Grid Computing (2026) 24:18
https://doi.org/10.1007/s10723-026-09838-5

RESEARCH

Weight-based Disk I/O Scaling for Serverless Containers

Óscar Castellanos-Rodríguez · Roberto R. Expósito · Jonatan Enes · Juan Touriño

Received: 30 September 2025 / Accepted: 8 June 2026

Abstract Disk bandwidth is a critical resource for I/O-intensive applications that must transfer large volumes of data to and from persistent storage. Most multi-tenant infrastructures efficiently allocate CPU and memory resources to concurrent workloads, but typically lack mechanisms for allocating I/O bandwidth. As a result, users often resort to exclusive node reservations to avoid disk contention, which can lead to under-utilisation of other node resources if not fully exploited. Another common issue is that users do not know the exact resource requirements of their applications. Even when this is known, applications rarely maintain peak resource usage throughout their entire execution, resulting in wasted resources that could otherwise benefit other users. Today, many users prefer cloud serverless platforms because of their ease of use and flexible billing. However, these platforms have inherent limitations and may not be suitable for workloads with specific requirements. In this paper, we present a serverless scaling mechanism that dynamically adjusts disk I/O bandwidth for containerised applications by scaling their allocation up or down based on real-time usage and configurable weights. In addition, the system incorporates automatic extension management capabilities for virtual disk devices, such as logical volumes. Our approach can be integrated with other serverless scaling mechanisms, such as CPU and memory management, to provide a comprehensive resource scaling solution. The experimental results have shown significant performance improvements, with overall runtime reductions of up to 53% for concurrent I/O-intensive workloads compared to running them without serverless capabilities.

Keywords Serverless computing · Disk I/O scaling · Containers · Hot-extension

Ó. Castellanos-Rodríguez (✉) · R. R. Expósito · J. Enes · J. Touriño
Universidade da Coruña, CITIC, Computer Architecture Group, A Coruña, Spain
e-mail: oscar.castellanos@udc.es

R. R. Expósito
e-mail: roberto.rey.exposito@udc.es

J. Enes
e-mail: jonatan.enes@udc.es

J. Touriño
e-mail: juan.tourino@udc.es

1 Introduction

Clusters and servers are computing resources widely used by researchers in their daily work, as they usually need access to infrastructure to develop and run their applications. Many organisations have moved to cloud services, particularly serverless platforms [1–3], due to their ease of use, avoidance of hardware acquisition and maintenance, and flexible billing models. However, there are still users with specific needs that require on-premises infrastructures, for example to comply with data privacy regulations, in which storage location and potential data leakage are critical concerns.

Published online: 27 June 2026

Both approaches have their benefits and drawbacks. On-premises infrastructures tend to be more restrictive in terms of resource allocation, as they typically rely on resource managers such as SLURM [4] and OpenPBS [5], which only support static allocation mechanisms (e.g., CPU, memory, storage) and/or exclusive node access [6]. This lack of dynamic allocation or serverless-like management often leads to resource underutilisation, as most applications rarely consume all of their allocated resources throughout the entire execution. In addition, these resource managers typically lack support for I/O bandwidth allocation, so even if applications reserve significant amounts of other resources, they may still have to share disk access equally with other concurrent workloads. This constraint is particularly critical for I/O-intensive workloads, where performance can be severely degraded. As a result, many users often request exclusive access to nodes whenever possible. However, this practice not only exacerbates resource underutilisation when users do not fully use the requested nodes, but also increases queue times in highly contended systems because it is harder to fully release nodes than to partially release them.

On the other hand, serverless cloud infrastructures come with their own issues [7, 8], such as regional restrictions, long-term pricing concerns, compliance with private data regulations, and the difficulty or even impossibility of setting up and running applications with specific requirements. For example, Big Data workloads often require network communication between nodes to transfer data, but many serverless platforms do not support such inter-node communication because they typically follow the Function-as-a-Service model (FaaS) [9], which is designed for simple functions executed within restricted instances. Other issues include the runtime constraints often imposed by these platforms, which may be incompatible with the long-running nature of many workloads. On the positive side, serverless platforms typically offer dynamic storage allocation compared to on-premises infrastructures, but this storage is often network-based, which can have a negative impact on I/O-intensive workloads.

In this paper, we present a serverless-like scaling mechanism that dynamically allocates I/O bandwidth to workloads running in containerised environments. The allocated bandwidth is scaled up or down based on actual consumption and configurable weights to allow prioritising target workloads while maximising overall disk usage. Moreover, our mechanism has been integrated with an existing serverless platform [10] that manages CPU and memory, further enhancing serverless capabilities and providing a comprehensive resource scaling solution for containerised workloads. In addition, virtual devices (e.g., logical volumes) are supported, allowing their I/O bandwidth to be transparently managed and scaled, just like regular block devices. Since virtual devices can typically be resized, our approach also includes mechanisms for automatically managing their extension, particularly by supporting online resizing operations (i.e., hot-extensions). These operations are designed to minimise interference with running containers while effectively leveraging the additional capacity and/or performance provided. The main contributions of this paper to the state of the art are as follows:

- A serverless-like and weight-based mechanism that automatically and dynamically manages and scales the disk I/O bandwidth allocated to containers.
- An automated system for managing hot-extensions of virtual disk devices, further enhancing the provided disk I/O scaling capabilities.
- An experimental evaluation to measure the benefits of disk I/O scaling when executing I/O-intensive workloads compared to running them without serverless capabilities.

The remainder of the paper is organised as follows. Section 2 introduces key background concepts relevant to this paper, while Section 3 discusses related work. The implemented features are detailed in Section 4, and Section 5 evaluates the performance obtained when executing both single and concurrent workloads. Finally, Section 6 concludes the paper.

2 Background

This section introduces the main concepts relevant to this work, such as container-based virtualisation (Section 2.1) and an overview of serverless computing (Section 2.2). Finally, Section 2.3 presents some key concepts related to disk I/O.

2.1 Container Virtualisation

Containers are isolated virtual environments that rely on OS-level virtualisation as they share the OS kernel

in which they are hosted. This means that containers, unlike virtual machines, are very lightweight to deploy and have low overhead, making them an attractive solution for deploying multiple services at scale. From the OS perspective, containers are basically groups of processes, which makes them compatible with cgroups [11], a Linux kernel feature that allows for limiting, accounting for, and isolating the resource usage of processes. Therefore, cgroups enables vertical and dynamic scaling of container resources, potentially adjusting them based on actual usage, thus laying the foundation for serverless-like approaches.

2.2 Serverless Computing and FaaS

Serverless computing allows users to run workloads without managing or configuring the underlying infrastructure, delegating maintenance, resource allocation, and demand-driven scaling to the service provider. Such scaling is key to the serverless paradigm, as it allows users to access resources elastically and transparently, getting more resources as they need them and paying only for what they use. On the provider side, resources need to be scaled up appropriately to meet the user demands, and scaled down to ensure fair billing and reallocate underutilised resources to other users or applications. Virtualisation is the most extended approach to allow users to share a physical infrastructure while providing isolated environments to ensure privacy and security. The resources allocated to these environments can also be dynamically adjusted to meet scaling requirements. For example, Amazon Web Services (AWS) [12] offers Fargate [13] as a serverless service to deploy containers via a container orchestration platform.

Several implementations of the serverless computing model exist, with FaaS [9] currently being the most prevalent. Typical FaaS services allow users to run workloads by simply submitting their code, usually triggering the actual execution based on specific events or requests. This makes these services well suited for simple tasks, processing functions for a data stream, or stateless services with scaling requirements, such as web servers. However, this focus on simplicity often makes them unsuitable for more complex workloads, such as Big Data or HPC workloads, which require inter-node communication and exhibit high runtime and resource consumption, features that are typically constrained in current serverless platforms.

2.3 Disk I/O Management

Disk I/O is one of the main limiting factors when executing data-intensive workloads. User-level applications issue I/O operations via OS system calls (e.g., read, write), which are handled by the kernel in the most optimal way. To do this, queues are used to hold pending requests, while different algorithms are used to schedule them efficiently [14]. For example, the most basic one is First Come First Served (FCFS), where requests are served in order, while Shortest Seek Time First (SSTF) prioritises requests closest to the current disk position to reduce movement between disk sectors. Different implementations of these scheduling algorithms are integrated into the I/O stack of current operating systems, often allowing the user to change the default selection. Common I/O schedulers in Linux systems include [15]: 1) the ‘noop’ scheduler, which does not reorder requests and essentially implements FCFS; 2) the ‘deadline’ scheduler, which assigns deadlines to I/O requests to prevent starvation; and 3) the Completely Fair Queueing (CFQ) scheduler, which maintains per-process queues and allocates time slices to each queue to promote fairness. Although some of these schedulers have been deprecated in recent versions of Linux, newer ones with similar design principles have emerged [16]. For example, the ‘none’ scheduler, despite its simplicity as a replacement for ‘noop’, is currently widely used for SSD disks due to its low overhead as these disks lack mechanical arms and have minimal seek times [17]. Similarly, the Budget Fair Queueing (BFQ) scheduler was introduced as an alternative to CFQ, offering fair sharing based on bandwidth rather than on time slices.

As mentioned in Section 2.1, Linux cgroups allows to control and limit the resources allocated to processes and, by extension, to containers. To limit disk I/O usage, cgroups can directly throttle the read and write bandwidth and/or IOPS that each process can consume from each block device (e.g., physical disk or logical volume). However, the cgroups version determines how this management is performed. cgroups v1 applies throttling only to I/O operations performed directly on disk, i.e., bypassing the Linux buffer cache commonly

used to speed up I/O operations and avoid slower disk access. To do this, it uses the block I/O controller, known as 'blkio' [18]. cgroups v2 [19] overcomes this limitation and also applies throttling to I/O operations performed on the buffer cache, using the I/O controller (referred to as 'io').

3 Related Work

On the one hand, it is worth mentioning previous work related to I/O resource control at the system level. Many works focused on I/O throttling to prioritise specific processes or applications. In [20], the authors proposed techniques to address I/O congestion in MapReduce tasks by modifying the Apache Hadoop framework [21] and managing I/O streams, while in [22] the authors proposed I/O throttling to improve MPI-IO performance. IO QoS [23] is an I/O scheduler embedded in the Linux kernel designed to guarantee QoS in cloud environments. More recent work, such as GIFT [24], aims to ensure fairness between competing applications in HPC systems through a coupon-based mechanism and usage pattern analysis. Authors in [25] proposed a solution to dynamically provision and control I/O data rate for cluster managers such as YARN [26] and Mesos [27]. FastResponse [28] is another kernel-level approach to throttle and reduce disk contention on SSD devices and improve the performance of latency-sensitive workloads. Similarly, [29] targets SSD devices by leveraging insights from their internal architecture and machine learning models to predict workload performance and allocate applications to disks where their Service-Level Objectives (SLOs) can be met. It is also worth noting the Linux built-in command called IONice [30], which allows setting priorities for processes during disk operations, similar to the "nice" command, although it requires either the CFQ or BFQ schedulers. Finally, IOCost [31] is a cgroups-based mechanism integrated into the Linux kernel designed for containerised environments which enables application prioritisation through weights. Unlike approaches based on simpler metrics such as IOPS or transferred bytes, IOCost distributes I/O by estimating the device occupancy of requests. However, in standard kernels, IOCost only supports cgroups v2, and its effective configuration requires careful parameter selection and tuning to achieve well-balanced results in practice.

On the other hand, some cloud platforms provide serverless storage capabilities. For example, AWS offers options such as S3 and EFS [32]. S3 is an object-based system that provides a scalable and cost-effective solution, albeit with limited performance, while EFS is a network file system that automatically scales its capacity based on actual usage, offering higher performance than S3 in some scenarios. These solutions are primarily designed for use within cloud instances such as EC2 [33], Fargate [13] or Lambda [34], although some options support on-premises deployments via Storage Gateway [35]. However, both storage options are network-based, so their I/O performance can be suboptimal, especially for data-intensive workloads [36, 37]. In addition, these cloud services do not provide I/O bandwidth scaling, so the bandwidth available per instance is typically limited to prevent contention between users and to ensure QoS, which means that even during periods of low contention the bandwidth cannot be fully exploited. Several approaches aim to mitigate the I/O bottlenecks of serverless cloud environments. For example, StarShip [38] and Pocket [39] leverage different storage backends to distribute I/O traffic. En4S [40] proposes an object-based storage system aimed at meeting SLOs across tenants through a scalable scheduling framework that coordinates multiple storage instances. Glider [41] introduces stateful near-data computation within cloud storage services to minimise data transfers. Cloudburst [42] is a stateful FaaS platform that combines a global database with local caches to share data between instances without compromising I/O performance.

In summary, I/O bandwidth is a critical resource and throttling-based mechanisms can help prioritise some workloads and/or improve overall performance when running concurrent applications. Low-level I/O control approaches such as BFQ or IOCost can provide good performance, but they typically lack higher-level mechanisms to assess contention at workload deployment and therefore cannot fully exploit underutilised storage devices. Serverless platforms, on the other hand, usually provide scheduling abstractions to users, but they are generally limited when it comes to managing disk bandwidth as another shared resource. To the best of our knowledge, combining serverless-like I/O bandwidth

management with infrastructure- and contention-aware deployment capabilities remains largely unexplored.

4 Implementation

This section outlines the implementation details of this work. Section 4.1 describes the disk I/O scaling mechanism, Section 4.2 addresses the management of hot-extensions in virtual devices, and Section 4.3 details the integration of our scaling mechanism into a serverless platform.

4.1 Disk I/O Scaling

The first step in performing the I/O scaling is to determine the available disk bandwidth in the infrastructure. This is necessary not only for the allocation itself, but also to assess contention when deploying new containers and, if possible, to select less congested nodes. To do so, a lightweight benchmark using FIO [43] is executed to measure the maximum read and write bandwidth for both sequential and random access on a single disk (or all disks, if multiple are available). To ensure measurements closely reflect the underlying hardware performance, the Linux buffer cache is bypassed during these tests using the *O_DIRECT* hint. However, the system administrator can arbitrarily configure the maximum bandwidth values for the disk(s) if desired.

4.1.1 Application Deployment

Users can configure applications to take advantage of the disk I/O scaling mechanism, optionally specifying maximum and/or minimum read and write bandwidths. Applications can then be started on a user-defined number of containers, which are automatically deployed on the infrastructure. Each container binds and mounts one disk on its corresponding node. The I/O bandwidth of each container is dynamically scaled up or down based on its actual usage, while respecting the previously configured maximum and minimum limits. When a new application is launched, its containers are automatically deployed by default on the least congested nodes to minimise resource contention. If no free nodes are available, new containers are bound to unused disks if possible, or to the least congested ones otherwise.

Figure 1 shows an example of container allocation for two applications considering disk contention. The initial scenario consists of a single node with one disk, where a previously deployed application was already running (*App0*), which is made up of a single container that is allocated 800 MiB/s of disk bandwidth. At a later time, a second application (*App1*) is launched using two containers and also requesting 800 MiB/s of disk bandwidth, evenly distributed between both containers (i.e., 400 MiB/s each). Since there are currently no free nodes or available disks, both applications and the three containers must now share the already allocated disk, resulting in full utilisation of the estimated maximum bandwidth (1,600 MiB/s in this example). In this scenario, where multiple containers compete for the same disk, the available I/O bandwidth is dynamically distributed in real time based on actual usage. For example, if two containers share a single disk and one of them is not performing any I/O operations at a given moment, its allocated bandwidth is scaled down so that the other container can take advantage of the full disk bandwidth. Additionally, even though the example scenario illustrates allocation over a physical disk, devices such as logical volumes that may group several underlying disks can be used equivalently, provided that they are exposed as block devices to the host operating system.

4.1.2 I/O Scaling via Cgroups

The cgroups kernel feature provides a pseudo file system located at the */sys/fs/cgroups* path, where each process (or container) has its own directory with its cgroups control files. Our I/O bandwidth scaling operates by dynamically modifying the corresponding cgroups files responsible for disk I/O throttling for read and write operations. These files require the identification of the block device mounted by the container using its 'major' and 'minor' numbers, along with the allowed bandwidth in terms of bytes. In cgroups v1, the corresponding I/O bandwidth files are *blkio.throttle.read_bps_device* and *blkio.throttle.write_bps_device* for read and write operations, respectively, with the syntax: *<major>:<minor> <bw limit>*. In contrast, cgroups v2 uses a single file called *io.max* to control both bandwidth and IOPS limits for read and write operations, with the syntax: *<major>:<minor> rbps=<read bw> wbps=<write bw> riops=<read IOPS> wiops=<write IOPS>*. Any parameters not explicitly set are automatically

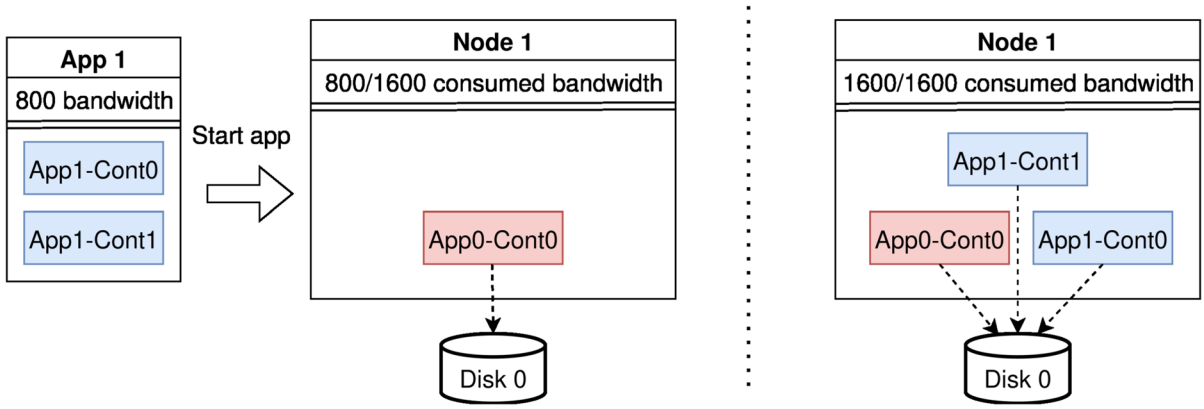


Fig. 1 Example of container allocation under disk contention

treated as unlimited, so that only the required ones need to be defined (in our case, read and write bandwidth).

In both versions of cgroups, the limits set at any given moment are applied to the associated processes in real time. Our scaling mechanism takes advantage of this feature by continuously monitoring the actual container I/O usage and dynamically adjusting the limits accordingly, scaling up when bottlenecks occur and scaling down when underutilisation is detected. Figure 2 illustrates the scaling mechanism applied to a container performing write operations, configured with a maximum and minimum bandwidth of 500 and 50 MiB/s, respectively. The container is initially allocated around 350 MiB/s of bandwidth, but as actual usage decreases, our mechanism scales down to 250 MiB/s to prevent overprovisioning. Later, when usage increases

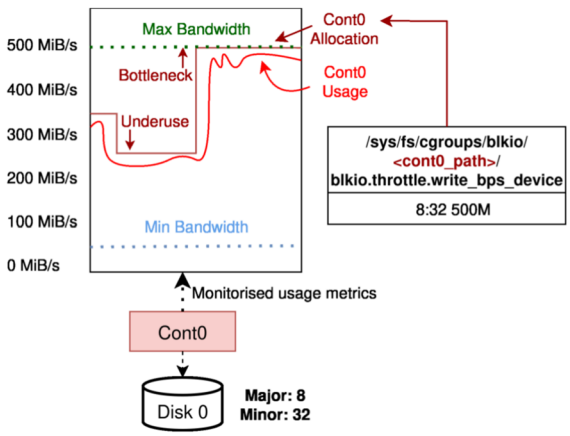


Fig. 2 Evolution of allocated bandwidth for a single container, highlighting periods of underutilisation and bottlenecks

again and the allocated bandwidth becomes a bottleneck, the mechanism scales up to the maximum available bandwidth to accommodate the higher load. Each scaling operation requires updating the corresponding cgroups file to enforce the new limit (e.g., the cgroups v1 write throttle depicted in the figure).

More specifically, our scaling mechanism operates by classifying the resource state observed within a given time window as a bottleneck, underutilisation, or regular utilisation, and then by scaling resources accordingly based on how many windows exhibit these conditions. This reactive approach inherently incurs overhead, as bandwidth is only allocated or released after a sustained demand is detected. As a result, workloads with high bandwidth requirements may experience slowdowns, particularly bursty workloads issuing many short I/O requests. Several parameters govern the allocation policy and the responsiveness of the scaling mechanism, allowing configurations that range from aggressive to conservative. The most influential parameters include the amount of bandwidth added during a scale-up operation, the number of windows exhibiting bottleneck or underutilisation conditions required to trigger scaling actions, and the window delay and window time lapse, which together define the temporal characteristics of the observation window. The window delay specifies how far into the past the analysed window begins, whereas the window time lapse defines its length. An aggressive configuration employs larger bandwidth increments, fewer windows required to trigger scaling actions, and shorter window delays and durations to react quickly to changes in demand. However, such configurations increase the risk of hysteresis,

an undesirable phenomenon in which resources oscillate rapidly, leading to system instability. For example, if resources are scaled immediately without relying on delayed observation windows, recently increased bandwidth allocations may be scaled down again before the workload has time to adapt, resulting in repeated and unnecessary scaling operations. Consequently, it is important to balance the responsiveness of aggressive configurations against the improved stability and hysteresis mitigation offered by more conservative configurations.

The I/O scaling approach is straightforward with sequential I/O operations, but random I/O is more challenging. Random I/O operations typically achieve lower throughput than sequential ones, but for the same amount of data transferred, they impose a higher load due to requiring a larger number of IOPS, especially for small block sizes. To address this, our mechanism relies on two strategies, explained below.

Sequential-to-random factor adjustment The random read and write bandwidths estimated during the initial disk benchmarking (Section 4.1) are used to determine the relative cost of random I/O compared to sequential access. For example, if a disk provides maximum bandwidths of 450 and 150 MiB/s for sequential and random reads, respectively, random access can be estimated to be three times as costly. Accordingly, the effective bandwidth is scaled by multiplying the random-access fraction by the corresponding factor (i.e., 3:1 in this example). This factor-based strategy improves disk occupancy estimation and supports more effective workload placement on less congested disks or nodes. In the previous example, a 150 MiB/s sequential read would be considered to use a third of the disk's capacity, whereas a 150 MiB/s random read would be treated as fully saturating it ($150 \times 3 = 450$).

To distinguish between sequential and random I/O, our approach relies on the observed block size as a practical heuristic. In most storage workloads, sequential operations typically employ larger request sizes (tens of KiB or more), whereas random workloads tend to issue much smaller blocks. This correlation allows us to infer access patterns using only readily available cgroups counters (transferred bytes and IOPS) without inspecting logical block addresses or maintaining detailed traces, which would add significant overhead. Our approach does not capture spatial locality directly: small sequential requests may be misclassified as random, and large random requests may be classified as sequential. Nevertheless, for our purpose of estimating the relative cost of random versus sequential access and scaling bandwidth accordingly, block size provides a lightweight and sufficiently accurate indicator. The threshold distinguishing "small" from "large" I/O requests can be tuned empirically to reflect the characteristics of the underlying hardware and workload. A threshold that is too large may classify many sequential requests as random, whereas a threshold that is too small may have the opposite effect. By default, this threshold is configured at 64 KiB. Algorithm 1 presents the pseudocode used to classify I/O in real time as sequential or random, by estimating the block size from continuously monitoring the cgroups counters for issued IOPS and transferred bytes. The block size for each interval is computed from the difference between consecutive samples, dividing the change in transferred bytes by the change in IOPS (see lines 9-16). For instance, if 10,000 bytes and 1,000 IOPS are recorded at time 0, and 50,960 bytes and 1,010 IOPS at time 1, the estimated block size is 4 KiB, as shown in (1). In this example, I/O will be classified as random according to lines 20-21. Since the algorithm runs periodically, changes in an I/O workload's pattern, particularly in block size, are detected and handled by applying the appropriate scaling rules.

$$\text{Block size} = \frac{50{,}960 - 10{,}000}{1{,}010 - 1{,}000} = 4{,}096 \text{ bytes} \quad (1)$$

The sampling interval determines the trade-off between responsiveness and accuracy in block size estimation. An excessively long interval may fail to capture rapid workload changes and misclassify the current pattern by mixing recent requests with older ones within the same sample. Conversely, an excessively short interval may induce hysteresis in workloads exhibiting mixed I/O behaviours, causing frequent oscillations between sequential and random classifications rather than accurately capturing the dominant access pattern. Moreover, very short sampling intervals may erroneously classify the workload as null (i.e., indicating an idle disk) if no I/O operations occur during a given interval.

Fairness enforcement Random I/O typically suffers from starvation when executed concurrently with sequential workloads. This occurs because sequential I/O efficiently leverages disk resources (e.g., queues,

Algorithm 1 Block size estimation and I/O classification

Require: *INTERVAL* ▷ Sampling period
Require: *THRESHOLD* ▷ Block size threshold
Require: $read\_bytes()$ ▷ cgroups counter for transferred bytes
Require: $read\_iops()$ ▷ cgroups counter for issued IOPS
Ensure: $block\_size[t]$ ▷ Estimated average block size for each sampling interval t
Ensure: $io\_type[t]$ ▷ Classification of the interval t as NULL, SEQUENTIAL or RANDOM

```
previous_bytes ← read_bytes()
previous_iops ← read_iops()
previous_time ← current_time()

while true do                          ▷ Loop every INTERVAL seconds
    current_bytes ← read_bytes()
    current_iops ← read_iops()
    current_time ← current_time()
    delta_bytes ← current_bytes − previous_bytes
    delta_iops ← current_iops − previous_iops

    if delta_iops > 0 then
        block ← delta_bytes/delta_iops
    else
        block ← 0                      ▷ No operations in this interval
    end if

    if block == 0 then
        type ← NULL
    else if block < THRESHOLD then
        type ← RANDOM
    else
        type ← SEQUENTIAL
    end if
    output (current_time, block, type)

    previous_bytes ← current_bytes
    previous_iops ← current_iops
    previous_time ← current_time
end while
```

buffers) while issuing relatively few operations, whereas random I/O usually generates a large number of small requests, each incurring significant per-request overhead. As a result, when both run concurrently, sequential I/O continues to utilise disk resources efficiently, while random I/O now faces even longer waiting times per request, leading to severe performance degradation. To mitigate this, a second implemented strategy ensures that random I/O allocations are never scaled down below a guaranteed minimum. Without this safeguard, sequential streams could otherwise consume most of the available bandwidth, even in the presence of random workloads, making the factor-based adjustment insufficient to guarantee fairness. By explicitly reserving bandwidth, concurrent sequential and random workloads share the disk capacity more equitably. For example, consider two containers sharing a single disk, one executing a sequential workload and the other a random workload. Each container is initially allocated half of the available bandwidth, ensuring that the random workload has a fair share. As the sequential usage fluctuates, its allocation can be scaled down or up accordingly, while the random allocation may also increase or decrease but never below its guaranteed minimum (which may be adjusted if the number of concurrent containers changes), preserving fairness and preventing starvation. Ideally, in this scenario, both workloads would achieve an average bandwidth close to half of the total disk bandwidth. However, this target is often difficult to attain for random workloads due to their inherently lower throughput. The resulting I/O bandwidth distribution can be further modulated using the weight-based approach described in the following section. This enables scenarios in which sequential workloads are prioritised, achieving higher throughput at the potential expense of starving random workloads. Conversely, random workloads may be prioritised to improve fairness, at the cost of reducing the bandwidth available to sequential workloads.

4.1.3 Weight-based Balancing

Another mechanism implemented is a weight-based balancing system for distributing disk bandwidth between running containers. The read and/or write bandwidth of each container can be assigned a custom weight, with a default value of 1 (the lowest possible weight). Our balancing system will then allocate I/O bandwidth proportionally according to these weights. For example, when two containers with equal weights run concurrently, they receive approximately the same bandwidth allocation. Conversely, a container with twice the weight of another receives roughly double the bandwidth. The weight-based balancing mechanism allows system administrators to assign weights to users' containers according to custom policies, and enables users to specify relative priorities among their own containers. This mechanism only works under contention; that is, if a container is not accessing the disk heavily, others may get most of the available bandwidth regardless of their assigned weights. It is important to note that this weight-based management is implemented externally to the cgroups' native weight

system, ensuring broad applicability and not depending on specific schedulers, as some cgroups weight implementations rely on underlying schedulers such as BFQ or CFQ. In practice, we recommend using a simple scheduler such as 'none', since our scaling operates at a higher level. This avoids potential conflicts and overhead introduced by weighted schedulers such as BFQ.

Figure 3 illustrates our weighted mechanism with two containers performing sequential write operations, configured with a 3:1 ratio in favour of the first container (*Cont0*). Initially, allocated bandwidths are about 275 and 225 MiB/s for *Cont0* and *Cont1*, respectively. When *Cont0* reaches a bottleneck, the balancer enforces the weight ratio, scaling up its allocation to 375 MiB/s while reducing *Cont1's* to 125 MiB/s. Later, as *Cont1* underutilises its share, its allocation is scaled down and the freed bandwidth is reallocated to *Cont0*. Finally, when *Cont1* reaches a bottleneck, the weight ratio is enforced again, scaling *Cont0* down and *Cont1* up accordingly (i.e., 375:125 MiB/s). The weight-based balancing mechanism can also be applied to random I/O by enforcing the weight ratio and preventing the scaling down of random loads. For example, in Fig. 3, if *Cont1* had been performing random writes, the scale down operation triggered by underutilisation would not have occurred, thereby preserving the weight ratio for most of the execution. Similar to the scaling mechanism itself, the balancing system operates over an independent time window to monitor the current resource state and exposes configurable parameters that govern its behaviour. In particular, the window delay and time lapse can be adjusted to enable more aggressive settings, using smaller values to enforce weight ratios more rapidly, or more conservative settings, with larger values to mitigate hysteresis.

Algorithm 2 Weight-based balancing mechanism with containers sharing a disk

Require: *WINDOW_TIMELAPSE* ▷ Polling period
Require: *THRESHOLD* ▷ Usage threshold

while true **do** ▷ Loop every *WINDOW_TIMELAPSE* seconds
  $total\_allocated\_bw \leftarrow 0$
  $weight\_sum \leftarrow 0$
  $weight\_read\_sum \leftarrow 0$
  $weight\_write\_sum \leftarrow 0$
  $read\_participants \leftarrow list$
  $write\_participants \leftarrow list$

  for container in containers **do** ▷ Look for containers actively using I/O
    $is\_read\_consumer \leftarrow container.read\_usage / container.read\_allocation > threshold$
    $is\_write\_consumer \leftarrow container.write\_usage / container.write\_allocation > threshold$

    if $is\_read\_consumer$ **then**
      $total\_allocated\_bw \leftarrow total\_allocated\_bw + container.read\_allocation$
      $weight\_sum \leftarrow weight\_sum + container.read\_weight$
      $weight\_read\_sum \leftarrow weight\_read\_sum + container.read\_weight$
      $read\_participants.add(container)$
    end if
    if $is\_write\_consumer$ **then**
      $total\_allocated\_bw \leftarrow total\_allocated\_bw + container.write\_allocation$
      $weight\_sum \leftarrow weight\_sum + container.write\_weight$
      $weight\_write\_sum \leftarrow weight\_write\_sum + container.write\_weight$
      $write\_participants.add(container)$
    end if
  end for

  if $weight\_sum > 0$ **then**
    $read\_distribution \leftarrow total\_allocated\_bw * (weight\_read\_sum / weight\_sum)$
    $write\_distribution \leftarrow total\_allocated\_bw * (weight\_write\_sum / weight\_sum)$
    $adjust\_distributions(read\_distribution, write\_distribution)$ ▷ Adjust to comply with the maximum disk bandwidths

    if $weight\_read\_sum > 0$ **then**
      $read\_slice \leftarrow read\_distribution / weight\_read\_sum$
    else
      $read\_slice \leftarrow 0$
    end if
    if $weight\_write\_sum > 0$ **then**
      $write\_slice \leftarrow write\_distribution / weight\_write\_sum$
    else
      $write\_slice \leftarrow 0$
    end if

    $slices\_adjusted \leftarrow false$
    while not slices_adjusted **do** ▷ Ensure each container receives an allocation within its maximum and minimum limits
      for container in read_participants **do**
        $container.read\_new\_allocation \leftarrow adjust\_allocation(container, read\_slice, read\_distribution, slices\_adjusted)$
      end for
      for container in write_participants **do**
        $container.write\_new\_allocation \leftarrow adjust\_allocation(container, write\_slice, write\_distribution, slices\_adjusted)$
      end for

```
51:         end while
52:
53:         for container in read_participants do
54:             read_amount ← container.read_new_allocation − container.read_allocation
55:             scale(container, read_amount)
56:         end for
57:         for container in write_participants do
58:             write_amount ← container.write_new_allocation − container.write_allocation
59:             scale(container, write_amount)
60:         end for
61:     end if
62: end while
```

Algorithm 2 presents the pseudocode of the weight-based balancing mechanism applied to containers sharing the same underlying disk. The algorithm executes periodically at each observation window, polling the current bandwidth allocation and the actual read and write usage of each container. For each container, the algorithm first determines whether it is actively using the disk by comparing the observed usage against a configurable threshold (lines 10-11). Lower threshold values allow containers with lower usage relative to their allocated bandwidth to remain active. The bandwidth allocations of all active containers are then accumulated into a shared pool (total_allocated_bw) and divided into separate read and write pools according to the sum of read and write weights, respectively (lines 28-29), while ensuring that the maximum read and write bandwidth limits of the underlying disk are not exceeded (line 30). Next, the algorithm computes the per-weight read and write bandwidth allocations to be assigned to each container (lines 32–41). These allocations are subsequently adjusted to enforce each container's minimum and maximum bandwidth constraints (lines 43–51). Enforcing these constraints may increase or decrease the pool of distributable bandwidth, potentially requiring multiple iterations until a stable distribution is reached. Finally, each container is scaled based on the difference between its current bandwidth allocation and the newly computed allocation (lines 53–60). Positive differences trigger scale-up operations, whereas negative differences result in scale-down actions.

The balancing system can benefit not only higher-priority containers, but also lower-priority ones, while improving overall completion time. This is because workloads rarely consume a single resource at full capacity throughout their entire execution; instead, they typically alternate between phases that stress different resources. To illustrate this idea, consider a data-intensive workload consisting of three distinct phases: input reading, data processing, and output writing, i.e., following the typical Read-Compute-Write (RCW) workflow pattern commonly used in Big Data and HPC applications. The first and third phases typically exhibit high I/O but low CPU utilisation, while the compute phase generally follows the inverse pattern. When multiple workloads with similar usage patterns run concurrently, they tend to compete for the same resources (e.g., disk when reading or writing data), while other resources that may be needed at later phases remain

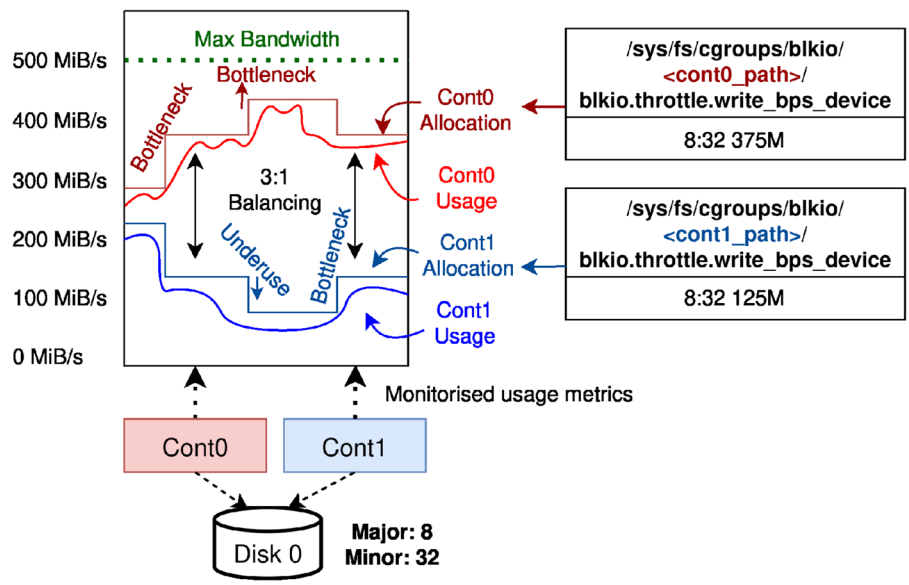


Fig. 3 Evolution of bandwidth over time for two containers, illustrating the impact of weight-based balancing

underutilised. Our weighted balancer enables higher-priority applications to complete I/O-intensive phases more quickly, releasing the congested disk earlier. This, in turn, allows lower-priority applications to fully use the disk while higher-priority ones move on to subsequent execution phases. Figure 4 illustrates a representative example, where two workloads running in separate containers compete for disk access, configured with a 3:1 weight ratio (i.e., the first container has three times the weight of the second). In this figure, which depicts part of the execution, the first workload (blue line) is nearing the end of its lifecycle and entering a writing phase, while the second workload (orange line) remains in a compute phase and will not begin writing until two minutes later. During this time, the first container is able to use over 800 MiB/s of bandwidth until the second container begins to compete for its share. At this point, the weighted balancer enforces the weight ratio and limits the write bandwidth to below 700 MiB/s and 250 MiB/s for the first and second containers, respectively. As a result, the performance of the first workload is not significantly affected by disk contention, and once it finishes at around second 200, the second workload gains full access to the available bandwidth.

4.2 Dynamic Bandwidth Extension

Many infrastructures rely on storage virtualisation rather than on physical disk devices due to its flexibility, which enables storage capacity and performance to be scaled up or down in response to demand. However, such dynamic changes may conflict with our I/O scaling mechanism for two main reasons: variations in available bandwidth and potential interference with

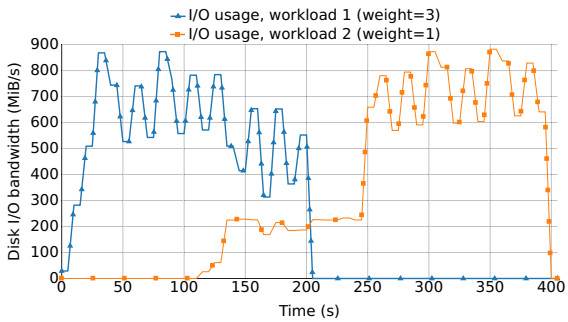


Fig. 4 Disk I/O scaling for applications with different weights

running workloads. To address these challenges, we introduce two key features that enable the integration of virtualised storage solutions into our system. First, we implement an automated and transparent process for provisioning virtual or logical devices from physical disks. System administrators only need to select the disks to be virtualised, and our solution automatically consolidates them and performs benchmarking to characterise their performance. Second, we design a dynamic extension mechanism that allows additional physical disks to be integrated into existing virtual devices. This mechanism determines the appropriate timing for the extension based on current device usage when the operation occurs during active use (i.e., a hot-extension), manages the extension procedure itself, and updates the performance profile of the virtual storage accordingly.

To provision virtual devices, our system relies on the Logical Volume Manager (LVM). Consequently, the procedure involves the following stages: 1) the creation of Physical Volumes (PVs) from the underlying physical disks; 2) the aggregation of these PVs into Volume Groups (VGs); and 3) the creation, formatting, and mounting of the corresponding Logical Volumes (LVs) from the VGs. A RAID 0 configuration is employed to maximise performance. This consolidated approach provides bandwidth close to the aggregated throughput of the underlying physical disks while simplifying disk management. It renders the storage layer transparent to containers and eliminates the need for users to manage multiple disks manually in order to fully exploit the available performance. Figure 5 illustrates this process, showing an example with two containers sharing an LV created from two physical disks, including a zoomed-in diagram of the commands required at each stage ordered from bottom to top. LVs and their available I/O bandwidth can be managed through cgroups in the same way as regular block devices, supporting all the implemented mechanisms (e.g., I/O scaling, weight-based balancing).

The extension process is also implemented using LVM, allowing new physical disks to be added to an existing LV both dynamically and automatically. Although cold-extensions (i.e., when an LV is not in use or not mounted) are also supported, this work focuses on hot-extensions, whose management constitutes the primary contribution of this process and requires additional considerations. The timing of a hot-extension is critical. When performed early, while containers are

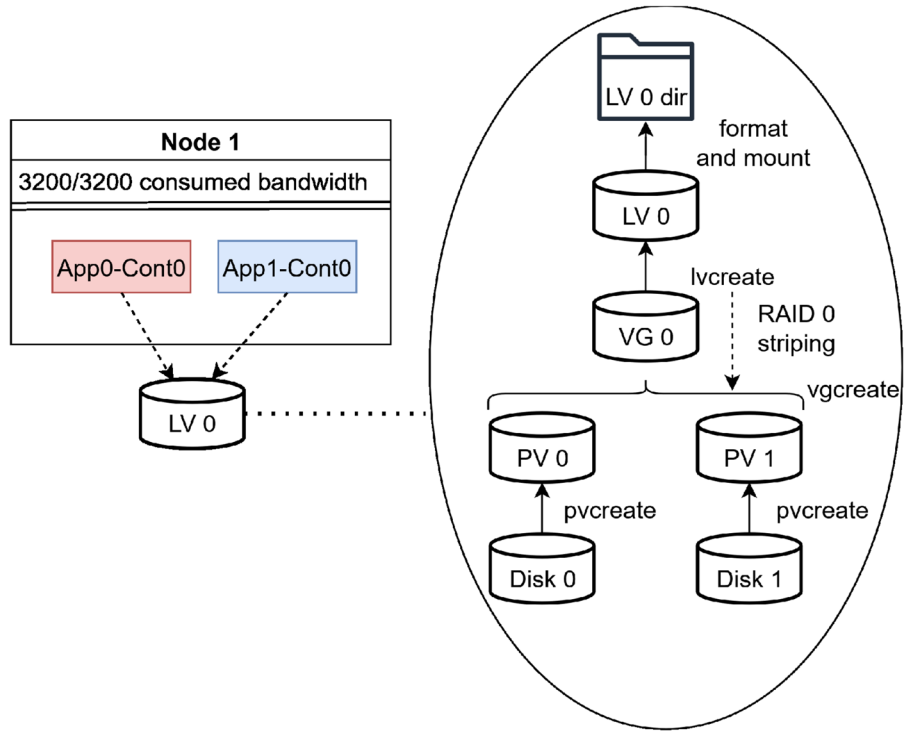


Fig. 5 Example of container allocation highlighting an LV as storage device

actively accessing the LV, it can provide a substantial and seamless increase in available performance. However, it may also interfere with running workloads, as it incurs inherent I/O overhead due to the re-striping of data across the expanded LV. Conversely, if the extension is performed late, when containers are completing execution or have already finished, only future workloads may benefit from the increased performance. To address this trade-off, our system can be configured to defer the hot-extension process for a specified timeout period, waiting until the LV's I/O usage falls below a configurable threshold. This approach minimises disruption to running applications during periods of intensive LV activity and allows the extension to be executed automatically at a potentially more suitable time (e.g., during a CPU-bound phase). If the timeout expires while I/O usage remains high, the system proceeds with the extension regardless, enforcing strict bandwidth limits on running containers to expedite the operation.

Algorithm 3 presents the pseudocode that determines when to trigger an extension, based on periodic polling of LV usage. A short polling period allows the algorithm to react more quickly to changes in usage and potentially trigger the extension sooner, but it introduces lightweight additional overhead from the polling process itself. Conversely, a low usage threshold may prevent the extension from being triggered even under minor I/O activity, whereas a high threshold may trigger the operation while containers are still performing moderate I/O. The bandwidth limit applied to running containers when an extension is forcibly triggered defaults to 10% of the total disk bandwidth and is shared among all active containers, but it remains configurable. The system administrator is responsible for selecting the LVs to be extended with new disks, while optionally configuring the parameters that govern each extension.

Additionally, to apply I/O scaling in accordance with the updated resource configuration and performance metrics, a new estimate of the maximum attainable bandwidth is required. If no containers are actively using the LV, this estimate is obtained by running the FIO benchmark described in Section 4.1. Otherwise, the benchmark is deferred until the LV becomes idle in order to avoid interference with running workloads and to preserve the accuracy of the measurements. During this contention period, an optimistic estimate is employed based on the number of newly added disks and the previously measured LV bandwidth (e.g., a vol-

Algorithm 3 Determining LV extension timing

Require: *LV* ▷ LV to extend
Require: *NEW_DISKS* ▷ New disks to add
Require: *INTERVAL* ▷ Polling period
Require: *THRESHOLD* ▷ Usage threshold
Require: *TIMEOUT* ▷ Timeout period
Require: *LIMIT* ▷ Enforced BW limit
Require: *get_current_usage*() ▷ Current LV usage

current_usage ← *get_current_usage*(*LV*)
initial_time ← *current_time*()
elapsed_time ← 0
limit_applied ← *false*

while *current_usage* > *THRESHOLD* and *elapsed_time* < *TIMEOUT* **do** ▷ Loop every *INTERVAL* seconds
    current_usage ← *get_current_usage*(*LV*)
    elapsed_time ← *current_time*() − *initial_time*
end while

if *current_usage* > *THRESHOLD* **then**
    limit_active_containers_allocation(*LV*, *LIMIT*)
    limit_applied ← *true*
end if

extend_lv(*LV*, *NEW_DISKS*) ▷ Extend the LV

if *limit_applied* **then** ▷ Remove previous restriction if needed
    remove_containers_limit(*LV*)
end if

ume delivering 450 MiB/s is assumed to provide 900 MiB/s after the addition of a second disk).

This hot-extension feature is particularly valuable for workloads deployed on top of complex systems with their own storage management (e.g., cluster managers, distributed file systems). In these scenarios, incorporating new disks into operational nodes, either for storage or bandwidth reasons, typically requires service reboots to make them available, potentially disrupting running workloads and degrading overall performance. Our approach removes this limitation, enabling seamless disk expansion with ideally minimal disruption.

4.3 Integration Into a Serverless Platform

To provide a comprehensive resource scaling solution for containers, the disk I/O scaling mechanism has been integrated into our serverless platform [10], which already supports CPU and memory scaling via cgroups. This platform was selected for its seamless integration with our cgroups-based approach for scaling I/O resources and its ability to vertically scale CPU and memory resources without requiring container restarts or disrupting active jobs, unlike other serverless systems such as OpenWhisk [44] or vHive [45], which are typically more focused on horizontal scaling. Essentially, the platform manages resource allocation following a similar approach to our I/O scaling mechanism, by initially assigning a specific amount of CPU and memory to containers and modifying the corresponding cgroups files accordingly to enforce the allocation. Real-time usage metrics for CPU and memory are continuously collected by the serverless platform and stored in a time-series database. Scaling decisions are then made periodically by comparing resource usage with allocated resources and container-specific dynamic thresholds. If actual usage approaches the upper threshold, the platform attempts to increase the allocation; conversely, if usage is close to the lower threshold, the allocation is reduced. After each scaling operation, thresholds are recalculated for subsequent iterations.

To enable the integration of our I/O scaling mechanism into the serverless platform, the weighted balancer has also been extended to include support for CPU and memory resources, enabling workload prioritisation not only during I/O-intensive phases, but also during compute-bound processing phases. This integrated approach makes it easier to experiment with more complex workloads, while providing a comprehensive evaluation of both the disk I/O scaling mechanism and the overall effectiveness of the serverless platform.

5 Performance Evaluation

The primary aim of this evaluation is to analyse the impact of the disk I/O scaling mechanism when executing data-intensive workloads. By integrating our mechanism into the serverless platform described in Section 4.3, it becomes possible to assess more complex scenarios involving concurrent scaling of multiple resources. All experiments were conducted on single, dedicated Linux-based servers to eliminate external factors, such as network delays or performance fluctuations, that might otherwise affect the results. Specifically, two separate testbeds were used for different sets of experiments. Their main characteristics are summarised in Table 1. In particular, the number of phys-

ical cores (32 and 16, respectively) corresponds to a total capacity of 3,200 and 1,600 CPU shares. It is also worth noting that the first testbed is based on cgroups v1, whereas the second supports cgroups v2, thereby enabling control over both direct and buffered I/O (see Section 2.3). The experiments compare two different scenarios: non-serverless executions running directly on the server (referred to as the standard scenario), and executions using our I/O scaling mechanism within the serverless platform (the serverless scenario). Unless otherwise specified, all experiments use the 'none' disk scheduler.

Several sets of experiments are presented next. Section 5.1 focuses on running single workloads to measure the potential overhead introduced by the I/O scaling mechanism, while also exploring different disk layouts. Section 5.2 evaluates the effectiveness of the I/O throttling mechanism for both consecutive and concurrent workloads, while Section 5.3 compares the performance of oversubscribed workloads running at different levels of resource contention. Section 5.4 evaluates the weight-based balancing mechanism presented in Section 4.1.3. Section 5.5 introduces more complex experiments where mixed computation-I/O workloads are run concurrently to emulate more realistic usage scenarios. Section 5.6 extends this evaluation to a larger number of concurrent containers to assess the scalability of the serverless approach. Section 5.7 evaluates our hot-extension mechanism, measuring the performance impact of dynamically expanding storage capacity and bandwidth during active workload execution. The experiments presented in Sections 5.1 to 5.7 were conducted on the first testbed, where both the standard and serverless scenarios were evaluated using synthetic workloads generated with the FIO tool [43], which allows precise control over I/O load patterns. Python workloads were also executed in selected experiments to introduce CPU-intensive computations and overlap them with I/O activity. As this testbed is based on cgroups v1, FIO was configured to use direct I/O and bypass the Linux buffer cache so that I/O throttling could be enforced. This extensive set of experiments enables an evaluation focused strictly on disk I/O behaviour, avoiding the additional complexity introduced by buffered I/O. Sections 5.1 and 5.7 use the three available physical disks, either individually or in LV layouts, whereas Sections 5.2 to 5.6 are limited to a two-disk LV layout for practical reasons and due to hardware availability constraints. Finally, Section 5.8 evaluates real, non-synthetic workloads on the second testbed, which supports cgroups v2. This allows the assessment of the serverless approach under more realistic conditions.

Each experiment was executed at least five times to ensure the consistency of the observed results. The reported results correspond to representative runs whose outcomes are close to the median of the measured values. As most experiments rely on synthetic workloads, the variability across runs is relatively low. Table 2 presents the main configuration parameters used by the different mechanisms evaluated in this work, as described in Sections 4.1.2 (I/O scaling and block size estimation), 4.1.3 (weight-based balancing), and 4.2 (LV extension). Overall, the selected values represent a relatively aggressive configuration, targeting high performance while limiting excessive hys-

Table 1 Testbeds used in the evaluation

	Testbed 1	Testbed 2
Processors	2x Intel Xeon Silver 4216	2x Intel Xeon E5-2650 v2
Physical cores	32	16
Memory	256 GiB	64 GiB
Disks	3x SATA SSD	1x SATA SSD
cgroups	v1	v2

Table 2 Main configuration parameter values for the mechanisms used in the experiments

Function	Parameter	Value
Scaling	Window delay	10s
	Window time lapse	5s
	Scale-up increment	50%−75%
	Bottleneck windows	1
	Underutilisation windows	6
Block size estimation	Interval	1s
	Threshold	100 KiB
Weight-based balancing	Window delay	10s
	Window time lapse	10s
	Threshold	10%
LV extension	Interval	5s
	Threshold	20%
	Limit	10%

teresis. The most relevant parameters are the scale-up increment, ranging between 50% and 75%, and the number of bottleneck events required to trigger scaling actions (1). The scale-up increment specifies the increase in a container's bandwidth allocation when additional bandwidth is required. Across experiments, the increment varies from 50% of the host disk's total bandwidth, a conservative setting suitable for concurrent workloads sharing I/O, up to 75%, a more aggressive setting useful for single-workload scenarios. Given that a single bottleneck event is sufficient to trigger scaling, once a bottleneck is detected within an observation window, the affected container's bandwidth allocation is increased by the configured increment (50–75%), or by a smaller amount if insufficient bandwidth is available. The last row in Table 2, which reports the LV extension parameters, is only relevant for the experiments presented in Section 5.7.

5.1 Overhead of I/O Scaling on Single Workloads

The first set of experiments consists of running a single FIO workload within a container for a fixed duration of 10 minutes, comparing the standard and serverless scenarios on Testbed 1 (see Table 1). In the standard scenario, this workload is intended to obtain full I/O bandwidth from the server physical disks, providing a baseline to accurately measure the overhead introduced by the I/O scaling mechanism. In the serverless scenario, the container starts with a low bandwidth allocation that is dynamically scaled up when higher demand is detected (i.e., the full bandwidth is not initially allocated). Table 3 reports the average bandwidth achieved in both scenarios across three storage layouts: 1) a single disk; 2) a two-disk LV; and 3) a three-disk LV, with LVs configured in RAID 0. Regarding I/O patterns, read and write operations are evaluated using both sequential and random access patterns.

Analysing the different disk configurations under the standard scenario to establish the baselines, the results show that a single disk provides approximately 520 MiB/s for sequential reads and 430 MiB/s for sequential writes. The two-disk LV achieves around 910 and 840 MiB/s, respectively, while the three-disk LV reaches roughly 1,570 and 1,300 MiB/s, demonstrating the benefits of the RAID 0 level implemented via LVM. Random read performance scales only modestly, increasing from 105 MiB/s to 123 MiB/s and then to 176 MiB/s, while random write decreases from 154 MiB/s to 107 MiB/s before recovering to 139 MiB/s. In the serverless scenario, the results are very similar overall, with maximum overhead of up to 9% across all storage layouts and an average overhead of around 4%, validating the effectiveness of our approach. As the I/O load in these experiments remains almost constant throughout the execution, the overhead introduced by our I/O scaling mechanism is minimal, as only a limited number of scaling operations are actually triggered. Random write performance showed less stability during experimentation, with recurring fluctuations, but their impact is mitigated by the factor adjustment strategy described in Section 4.1.2, which enables random workloads to allocate more bandwidth than they actually consume.

5.2 Dynamic I/O Throttling Capabilities

The goal of these experiments is to verify that our serverless scaling approach can enforce the configured I/O limits and to evaluate the associated overhead. They are conducted on the two-disk LV described previously (see Table 3 for the single-workload results). Figure 6 shows the execution of three consecutive FIO workloads within the same container when different

Table 3 Average bandwidth running a single workload with different disk layouts on Testbed 1

I/O pattern	Scenario	Average read / write bandwidth (MiB/s)		
		Single disk	Two-disk LV	Three-disk LV
Sequential	Standard	524 / 429	911 / 841	1572 / 1303
	Serverless	498 (-5%) / 398 (-7%)	864 (-5%) / 799 (-5%)	1518 (-3%) / 1264 (-3%)
Random	Standard	105 / 154	123 / 107	176 / 139
	Serverless	104 (-1%) / 150 (-3%)	123 (0%) / 97 (-9%)	168 (-5%) / 132 (-5%)

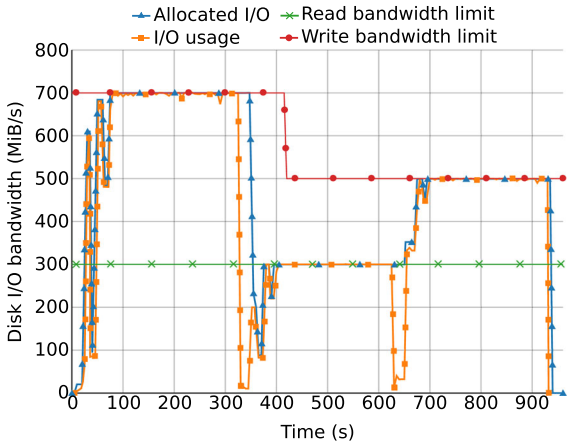


Fig. 6 I/O usage for multiple workloads executed within the same container in the serverless scenario

I/O limits are applied. The first and third workloads perform sequential writes, while the second performs sequential reads. During the writing phases, the write bandwidth of the container is limited to 700 and 500 MiB/s, respectively, while the read bandwidth is limited to 300 MiB/s for the entire execution, affecting only the reading phase. The blue line represents the I/O bandwidth allocated to the container and the orange line depicts the actual usage. The green and red horizontal lines show the enforced read and write limits, respectively. Note that the write limit is reduced to 500 MiB/s around the 400-second mark, just after the start of the reading phase, to demonstrate the dynamic nature of our approach. It can be seen that the allocated and used bandwidth consistently stay below the corresponding limits: up to 700 MiB/s during the first writing phase (ending at second 320), up to 300 MiB/s during the reading phase (between seconds 320 and 620), and up to 500 MiB/s during the final writing phase (from second 620 to the end). A small number of scaling operations between phases (approximately five per phase) prevents the bandwidth from reaching the configured limit immediately at the start of each phase. Consequently, each phase experiences a slight overhead relative to the set limit, resulting in an average overhead of approximately 8%.

Figure 7 presents the execution of three concurrent sequential write workloads, each running in a separate container and configured with different I/O limits: 400, 250, and 100 MiB/s, respectively. The blue and orange lines represent the allocated bandwidth and actual usage, while the enforced I/O limit is shown in green. Compared to the scaling configuration shown in Table 2, this experiment adopts a slightly more conservative setting, with the scale-up increment set to one third of the total available bandwidth. As observed, the I/O limits are correctly enforced across all containers. However, as expected, concurrent execution introduces more overhead. This is mainly due to increased contention for disk access, which makes individual workloads less stable, especially compared to the single-container scenario shown in Fig. 6. As a result, each container achieves lower effective bandwidth and undergoes more frequent I/O scaling operations, contributing to an overall overhead of around 15%.

5.3 Evaluation of Oversubscribed I/O Workloads

These experiments compare the performance of the standard and serverless scenarios when running two concurrent sequential workloads in separate containers on the two-disk LV (see Table 3). Each workload is configured to run for 10 minutes. Here, disk bandwidth is oversubscribed, i.e., the combined bandwidth requested by both containers exceeds the available capacity and must therefore be shared and allocated accordingly. To reflect more realistic conditions, dif-

Fig. 7 I/O usage for three concurrent workloads executed in the serverless scenario with different limits

Table 4 Average bandwidth running two concurrent workloads with different delays

Scenario	Average bandwidth (MiB/s)			
	High-contention (60s delay)		Low-contention (420s delay)	
	Cont1	Cont2	Cont1	Cont2
Standard	477	451	717	712
Serverless	452 (-5%)	445 (-1%)	713 (-1%)	701 (-2%)

ferent startup delays were applied to the second container, as different applications rarely start at exactly the same time. This delay effectively modulates the degree of disk contention during I/O access. Two different cases were analysed: a high-contention scenario characterised by a short delay (60 seconds), resulting in resource sharing for most of the runtime, and a low-contention scenario with a longer delay (420 seconds), where containers compete for disk access only for short periods.

The results for both contention scenarios, where each container can potentially use the full available bandwidth, are summarised in Table 4. The serverless execution continues to introduce very low overheads compared to the standard, non-serverless execution, with values reaching only up to 5%. As expected, the average overhead is slightly lower when resource contention is reduced with a longer delay. Overall, these results further demonstrate that I/O bandwidth is effectively and evenly distributed between containers under both contention levels, even though one container starts execution earlier and allocates resources first. Figure 8 shows bandwidth usage over time for the high-contention serverless scenario. The blue and orange lines represent the allocated and consumed I/O bandwidth for the first container, while the green and red lines correspond to the second container. As expected, the first container initially gets most of the available bandwidth. However, as soon as the second container starts at around second 60, the bandwidth is evenly shared between them. After the first container completes its workload at around second 600, the second container takes advantage of the full bandwidth for the remainder of its execution.

Table 5 presents the results of the serverless execution when conducting the same experiments as before, but now with an I/O limit of 250 MiB/s enforced on only one container at a time, across two separate runs (i.e., first applied to one container, then to the other). In the high-contention case, the aggregated bandwidths for both containers in the serverless execution (612+241=853 and 243+616=859) are comparable to those observed in the standard scenario (477+451=928), with overheads of 8% and 7%, respectively. In addition, the container without limits achieves up to 37% more bandwidth in the serverless scenario, highlighting the benefits of workload prioritisation. The limited container, by contrast, experiences the expected bandwidth reduction as a result of the imposed constraint. In the low-contention case, the container without limits continues to benefit from higher priority, but to a lesser extent. Both the aggregated bandwidths

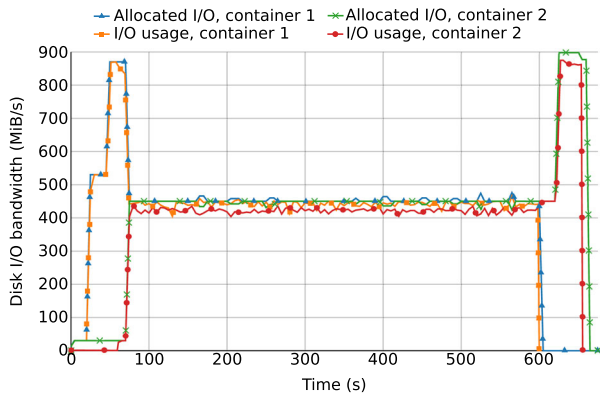


Fig. 8 I/O usage for two concurrent workloads executed in the high-contention serverless scenario

Table 5 Average bandwidth running two concurrent workloads with different delays and bandwidth limits

Scenario	Average bandwidth (MiB/s)			
	High-contention (60s delay)		Low-contention (420s delay)	
	Cont1	Cont2	Cont1	Cont2
Standard	477	451	717	712
Serverless	612 (+28%)	241 [max=250] (-47%)	765 (+7%)	245 [max=250] (-66%)
Serverless	243 [max=250] (-49%)	616 (+37%)	242 [max=250] (-66%)	765 (+7%)

and the bandwidth obtained by the second container suffer a significant reduction, which is again expected given the enforced limit. These results suggest that workload prioritisation can be beneficial under high contention by mitigating interference from co-located workloads, particularly when there is significant overlap. In contrast, lower overlap (e.g., due to long startup delays) tends to reduce aggregated bandwidth when limits are imposed. Under low contention, while it may still alleviate penalties during short contention periods, it may not be optimal overall. A weighted approach, introduced in Section 4.1.3 and evaluated in the following section, could provide better performance and resource efficiency.

Finally, Fig. 9 illustrates the high-contention execution in the serverless scenario when the I/O limit (250 MiB/s) is applied to the second container. The colour scheme follows that of Fig. 8, with a new purple line indicating the limit imposed on the second container. This figure shows that the first container can use almost 900 MiB/s until the second container starts and claims its bandwidth share. Despite disk contention, the first container maintains a significant share of the available bandwidth (around 610 MiB/s), experiencing considerably less performance degradation due to concurrency compared to the results shown in Fig. 8, thereby reinforcing the findings presented in Table 5.

5.4 Evaluation of Weight-based Balancing

In this section, our weight-based balancing system described in Section 4.1.3 is evaluated under concurrent sequential and random I/O workloads. For comparison, we evaluate a third scenario that mirrors the standard setup, except that the 'none' scheduler used in previous experiments is replaced by a weight-based scheduler, BFQ. Storage is provided by the two-disk LV (see Table 3), and workloads are configured to run for 10 minutes. Table 6 presents baseline results for single workloads in terms of average bandwidth, reporting both the overhead introduced by BFQ and the overhead of our serverless scaling approach. For sequential I/O, both impacts are negligible. However, for random I/O, BFQ degrades performance by up to 74%, whereas the serverless mechanism maintains low overhead.

Table 7 reports the results obtained when running two concurrent containers executing sequential read and write workloads. A useful objective metric for evaluating how effectively weights are enforced is the

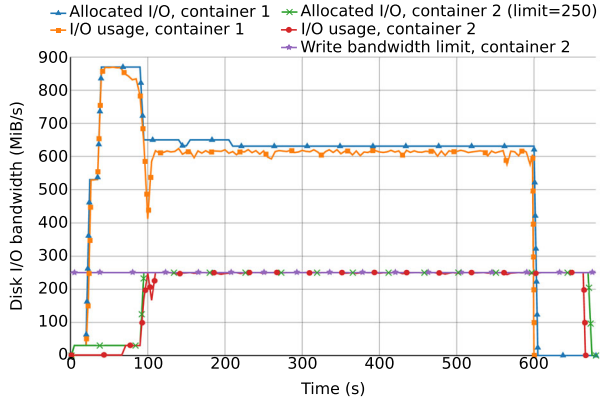


Fig. 9 I/O usage for two concurrent workloads executed in the high-contention serverless scenario with a limit

Table 6 Average bandwidth running a single workload to compare the serverless scenario with different disk schedulers

I/O pattern	Scenario	Average read/write bandwidth (MiB/s)
Sequential	Standard	911 / 841
	BFQ	893 (-2%) / 837 (0%)
	Serverless	864 (-5%) / 799 (-5%)
Random	Standard	123 / 107
	BFQ	39 **(-68%)** / 28 **(-74%)**
	Serverless	123 (0%) / 97 (-9%)

Notes: The bolded values are used to visually highlight the most relevant results in each experiment and to facilitate the interpretation of the evaluation results

percentage accuracy with which the achieved average bandwidths approximate the target weight ratio. This metric is computed using (2), where A and B denote the workloads, b represents the achieved average bandwidth, and w the configured weight:

$$\text{Weight ratio accuracy}_{A:B} = \frac{min(\frac{b_A}{b_B}, \frac{w_A}{w_B})}{max(\frac{b_A}{b_B}, \frac{w_A}{w_B})} \quad (2)$$

With equal weights (or no explicit weights in the standard scenario, as the none scheduler does not support weighting), all scenarios achieve an even distribution of bandwidth for both read and write operations, approaching the ideal 1:1 ratio with accuracies between 94% and 99%. BFQ introduces negligible overhead only for writes (-3%), while the serverless scenario adds slight overhead to both reads (-7%) and writes (-3%). When using a 4:1 ratio, BFQ enforces weights very accurately, providing ratios of 700/180=3.9 (98% accuracy) for reads and 658/190=3.5 (88% accuracy) for writes. The serverless mechanism achieves slightly less precision: 649/189=3.4 (85% accuracy) and 607/193=3.2 (80% accuracy). In both scenarios, overall performance is mostly preserved relative to the standard scenario.

Table 8 presents the results for two concurrent containers executing mixed sequential and random read and write workloads, with the first container performing sequential I/O and the second performing random I/O. In these experiments, the performance disparity between the workloads becomes more pronounced. For mixed reads, the random workload is nearly starved in the standard scenario (around 5 MiB/s). BFQ does not alleviate this issue when using equal weights and can even further reduce random bandwidth, although it slightly increases aggregated throughput. The serverless mechanism, by contrast, enforces fairness: sequential bandwidth is reduced by 33%, lowering total throughput by 30%, but random read performance increases from 5 to 23 MiB/s (+360%). This demonstrates that our fairness enforcement effectively mitigates starvation, albeit at the cost of overall perfor-

Table 7 Average bandwidth running two concurrent sequential workloads to compare the serverless scenario with different disk schedulers

I/O	Scenario	Weight ratio	Average bandwidth (MiB/s)		Total I/O (MiB/s)
			Cont1	Cont2	
Read	Standard	–	422	420	842
	BFQ	1:1	421	423	844 (0%)
		4:1	**700**	**180**	880 (+5%)
	Serverless	1:1	392	388	780 (-7%)
		4:1	**649**	**189**	838 (0%)
Write	Standard	–	430	405	835
	BFQ	1:1	401	405	806 (-3%)
		4:1	**658**	**190**	848 (+2%)
	Serverless	1:1	414	397	811 (-3%)
		4:1	**607**	**193**	800 (-4%)

Notes: The bolded values are used to visually highlight the most relevant results in each experiment and to facilitate the interpretation of the evaluation results

Table 8 Average bandwidth running two concurrent sequential and random workloads to compare the serverless scenario with different disk schedulers

I/O	Scenario	Weight ratio (Seq:Rnd)	Average bandwidth (MiB/s)		Total I/O (MiB/s)
			Cont1	Cont2	
Read	Standard	–	667	**5**	672
	BFQ	1:1	768	4	772 (+15%)
		4:1	773	4	777 (+16%)
		1:4	416	**19**	435 (-35%)
	Serverless	1:1	450	**23**	473 (-30%)
		4:1	644	5	649 (-3%)
		1:4	193	**54**	247 (-63%)
Write	Standard	–	250	**1**	251
	BFQ	1:1	372	2	374 (+49%)
		4:1	400	2	402 (+60%)
		1:4	294	4	298 (+19%)
	Serverless	1:1	237	**11**	248 (-1%)
		4:1	203	6	209 (-17%)
		1:4	182	**28**	210 (-16%)

Weight ratio: Seq=sequential; Rnd=random
Notes: The bolded values are used to visually highlight the most relevant results in each experiment and to facilitate the interpretation of the evaluation results

mance. Since random I/O typically achieves significantly lower bandwidth than a concurrent sequential workload, weight ratios are not meaningfully respected in this set of experiments. A more suitable metric for assessing fairness is the closeness of the achieved bandwidth to the ideal fair allocation. With two concurrent containers, the ideal bandwidth corresponds to half of the maximum attainable throughput. Using the results from Table 6 as a baseline, the ideal bandwidth for the sequential workload is approximately 911/2 = 456 MiB/s for reads and 421 MiB/s for writes, while for the random workload it is around 62 MiB/s for reads and 54 MiB/s for writes. Under equal weights, the resulting fairness values for the sequential and random workloads are 146% and 8% in the standard scenario, 168% and 6% with BFQ, and 99% and 37% with the serverless approach. When using a 4:1 ratio favouring sequential I/O, BFQ behaves similarly to the equal-weight case, while the serverless mechanism more closely resembles the standard scenario. When reversing the ratio to favour random I/O, both BFQ and serverless reduce sequential performance accordingly to benefit the random workload. In this case, the serverless mechanism nearly triples random bandwidth compared to BFQ, achieving fairness values of 42% and 87% for the sequential and random workloads, respectively, whereas BFQ attains 91% and 31%. This improvement comes at the cost of a sharper drop in total throughput (–35% for BFQ and –63% for serverless), reflecting the fairness trade-off.

Mixed writes are the most challenging case, exhibiting high instability and steep performance drops. In the standard scenario, sequential writes average 250 MiB/s, whereas random writes reach only 1 MiB/s. BFQ increases sequential throughput while slightly improving random performance. The serverless mechanism maintains sequential throughput very close to the standard case but increases random performance significantly to 11 MiB/s. With a 4:1 ratio, BFQ results remain similar, whereas the serverless mechanism loses performance in both workloads. With a 1:4 ratio, BFQ modestly improves random throughput to 4 MiB/s and raises sequential by 18% (fairness of 70% and 7%, respectively), whereas the serverless scenario reduces sequential throughput by 27% but significantly boosts random to 28 MiB/s (fairness of 43% and 52%). Overall, prioritising sequential writes yields limited benefits due to the inherent performance drop, while prioritis-

ing random writes provides substantial improvements with the serverless mechanism.

In summary, two key insights emerge from this set of experiments. First, the serverless approach distributes bandwidth proportionally to weights without incurring significant overhead, although specialised disk schedulers such as BFQ can achieve more precise and efficient distributions. Second, the serverless fairness strategy for random I/O effectively mitigates starvation and significantly improves random performance, but at the cost of overall performance. Finally, it is worth noting that configuring a disk scheduler such as BFQ for an LV composed of multiple disks is less straightforward, as the scheduler cannot be directly set via the device-mapper interface. It must instead be configured for each underlying physical disk, which increases the risk of misconfiguration.

5.5 Evaluation of Mixed Workloads

These experiments involve running containers with mixed workloads that combine both CPU and I/O activity on the two-disk LV (see Table 3). These workloads follow the RCW pattern mentioned in Section 4.1.3, where FIO is used to implement the reading and writing phases, while the compute phase executes a Python workload that performs CPU-intensive computations on a mathematical function, specifically the factorial. Unlike previous tests that measured average bandwidth over a fixed duration, these experiments execute a fixed amount of work. Therefore, the results allow us to analyse the impact of the serverless scenario on the total runtime of mixed workloads. In the standard scenario, containers are allocated CPU resources based on the number of concurrent workloads. For example, if a single container is running, it can potentially use all available CPU cores on the server (32), but if two containers run concurrently, each is limited to half of the available cores. This is because standard executions are conducted on a multi-user shared infrastructure, where resources must be requested by users via the SLURM scheduler. As a result, concurrent workloads cannot share CPU resources and must be statically partitioned. This setup is typical in infrastructures shared by multiple users (e.g., clusters and supercomputers) and orchestrated by resource managers such as SLURM. Furthermore, as mentioned in Section 1, SLURM does not support disk I/O bandwidth splitting and throttling, or dynamic CPU allocation. In contrast, our I/O scaling mechanism, integrated with the serverless platform (see Section 4.3), overcomes these limitations by dynamically allocating both CPU and disk bandwidth to each container in real time according to actual demand. This thereby allows each container to potentially use full capacity.

Three types of mixed workloads were evaluated: 1) "regular", which implements the RCW pattern using sequential I/O for both reading and writing phases; 2) "bursty", which is composed of multiple shorter instances of the regular workload and is intended to evaluate performance under highly variable resource consumption; and 3) "random", which implements the RCW pattern like "regular" but uses random I/O operations instead. Table 9 summarises the workload configuration for the different phases of the three workload types. The regular and random workloads execute a single iteration of the specified load, whereas the bursty workload executes three consecutive iterations. Each workload reads from and writes to multiple files, generating a total I/O load corresponding to the sizes reported in the table. During the compute phase, all workloads spawn 64 parallel processes, each calculating the specified factorial, thereby saturating the CPU resources.

Table 10 presents the results obtained by running these workloads in isolation, each within a single container, to assess the overhead introduced by our approach. The results are reported separately for each phase, along with the aggregated I/O time (i.e., reading + writing phases) and total workload runtime. As shown, the regular workload incurs the lowest overheads, with a 10% increase in CPU, aggregated I/O, and total runtimes. In contrast, the bursty workload has the worst results, with a higher increase in both CPU

Table 9 Configuration of mixed workloads

	Workload		
	Regular	Bursty	Random
Iterations	1	3	1
Read	150 GiB	54 GiB	18 GiB
Compute (factorial)	5×10^6	2.5×10^6	5×10^6
Write	150 GiB	60 GiB	16 GiB
I/O block size	1 MiB	1 MiB	4 KiB

Table 10 Runtimes for mixed workloads executed separately

Workload	Scenario	Runtime (seconds)				
		Read	Compute	Write	Total I/O (R+W)	Total
Regular	Standard	192	273	198	390	663
	Serverless	214 (+11%)	299 (+10%)	215 (+9%)	429 (+10%)	728 (+10%)
Bursty	Standard	209	252	243	452	704
	Serverless	256 (+22%)	315 (+25%)	301 (+24%)	557 (+23%)	872 (+24%)
Random	Standard	217	278	187	404	682
	Serverless	228 (+5%)	307 (+10%)	228 (+22%)	456 (+13%)	763 (+12%)

and I/O metrics, as more frequent scaling operations have a negative impact on performance, as expected. Finally, the random workload exhibits intermediate results between the regular and bursty cases. In summary, serverless executions show reduced performance compared to their non-serverless counterparts, due to the overheads introduced by dynamic scaling. This effect is amplified by the shorter phase durations in these experiments, which increase the relative impact of scaling operations, particularly for random write operations. However, as shown in the following subsections, these overheads can potentially be mitigated and performance improved in concurrent workload scenarios where resource reallocation across containers becomes possible. Nevertheless, our results suggest that such overheads are generally not a significant concern, except for burstier workloads, which represent the worst-case scenario for dynamic resource scaling. This limitation is common to scaling mechanisms in general, where the only practical solution is to rely on resource overprovisioning.

In the remainder of this section, we present experiments that combine and concurrently execute mixed workloads in both standard and serverless scenarios. We also analyse the performance impact of applying our weighted balancer to both CPU and I/O resources in the serverless scenario.

5.5.1 Concurrent Regular Workloads

Table 11 shows the results obtained when running two regular workloads concurrently in separate containers with different startup delays and weights. More specifically, delays of 0 and 180 seconds are used to represent high- and low-contention cases, respectively. The first conclusion that can be drawn is that serverless execution consistently reduces total runtimes compared to the standard scenario in almost all experiments, demonstrating its ability to efficiently manage CPU and disk resources for concurrent mixed workloads. Under high contention (i.e., 0-second delay) and equal weight assignment to both containers, the serverless execution essentially mirrors the standard scenario, where workloads synchronously compete for resources. In this case, the serverless approach yields slight improvements for the first container: 6% for both aggregated I/O and total runtime, while the second container experiences a negligible overhead in total runtime (only 1%). This is because the first container always starts earlier, even in the 0-delay experiment, allowing it to allocate bandwidth before the second container starts, after which the balancing system reallocates resources to ensure a fair distribution. This small advantage for the first workload allows it to complete the reading phase and start the compute phase just a little earlier, further benefiting from being allocated CPU cores first. Increasing the weight assigned to the first container under high contention (weight ratios of 2:1 and 4:1) significantly benefits its performance, reducing reading, compute, and writing times by up to 24%, 40%, and 36%, respectively. These improvements lead to total runtime reductions of up to 34%. The second container, by contrast, experiences a degradation in read performance, which is expected due to the uneven disk bandwidth allocation. However, it still shows improvements in the subsequent compute and writing phases, as well as in aggregated I/O and total runtime, albeit to a lesser extent (total runtime is reduced by up to 13% for the 4:1 weight ratio). These improvements, despite the unbalanced distribution introduced by the assigned

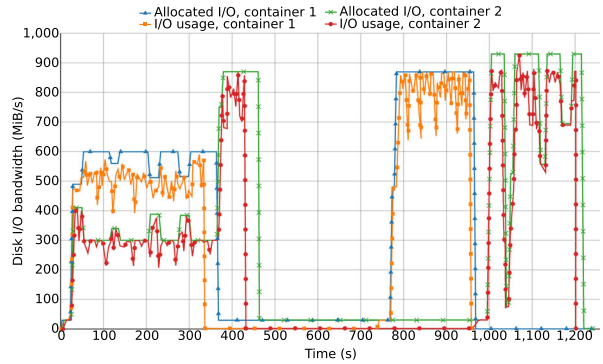
Table 11 Runtimes for concurrent execution of two regular mixed workloads with different delays

Delay	Scenario	Workload	Weight ratio	Runtime (seconds) Read	Compute	Write	Total I/O (R+W)	Total
0s	Standard	Regular 1	–	369	554	345	714	1268
		Regular 2		391	538	364	755	1293
	Serverless	Regular 1	1:1	369 (0%)	523 (-6%)	299 (-13%)	668 **(-6%)**	1191 **(-6%)**
		Regular 2		410 (+5%)	565 (+5%)	329 (-10%)	739 (-2%)	1304 **(+1%)**
		Regular 1	2:1	334 (-9%)	400 (-28%)	221 **(-36%)**	555 (-22%)	955 (-25%)
		Regular 2		420 (+7%)	541 (+1%)	234 **(-36%)**	654 (-13%)	1195 (-8%)
		Regular 1	4:1	279 **(-24%)**	333 **(-40%)**	222 **(-36%)**	501 (-30%)	834 **(-34%)**
		Regular 2		462 (+18%)	435 (-19%)	224 **(-38%)**	686 (-9%)	1121 **(-13%)**
180s	Standard	Regular 1	–	197	554	220	417	971
		Regular 2		201	538	266	467	1005
	Serverless	Regular 1	1:1	231 (+17%)	423 (-24%)	211 (-4%)	442 (+6%)	865 **(-11%)**
		Regular 2		253 (+26%)	414 (-23%)	224 (-16%)	477 (+2%)	891 **(-11%)**
		Regular 1	2:1	220 (+12%)	346 (-38%)	216 (-2%)	436 (+5%)	782 **(-19%)**
		Regular 2		269 (+34%)	406 (-25%)	217 (-18%)	486 (+4%)	892 (-11%)
		Regular 1	4:1	233 (+18%)	342 (-38%)	215 (-2%)	448 (+7%)	790 **(-19%)**
		Regular 2		279 (+39%)	408 (-24%)	221 (-17%)	500 (+7%)	908 (-10%)

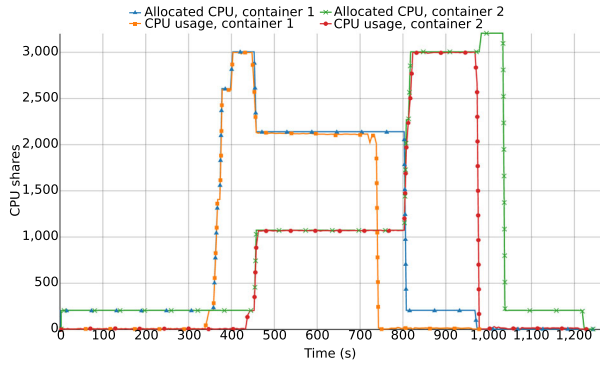
Notes: The bolded values are used to visually highlight the most relevant results in each experiment and to facilitate the interpretation of the evaluation results

weights, are attributed to reduced contention and more efficient resource usage, as discussed in Section 4.1.3. In low-contention cases (i.e., 180-second delay), disk contention is already minimal in the standard scenario, so some overhead is expected when introducing resource scaling. However, containers in the standard scenario are limited to allocating half of the total available cores due to the static allocation policy enforced by SLURM, regardless of the actual contention level. In contrast, the serverless scenario is not subject to this constraint, allowing for a more flexible and efficient use of resources overall, especially during compute phases. When containers run with equal priority (i.e., identical weights), the low-contention results show that, despite an overhead when reading, improvements during the compute and writing phases result in an overall performance gain of 11% for both containers. The unexpected improvement when writing is again due to the inherent advantage of the first container, which allocates resources earlier, thereby desynchronising the writing phases of the workloads. Increasing the weight for the first container further benefits its workload, achieving up to a 19% reduction in total runtime, while the performance of the second container remains largely unaffected. This more modest improvement compared to the high-contention executions is reasonable, as the balancer only affects those scenarios where workloads are actively competing for shared resources.

To illustrate a representative example from this set of experiments, Fig. 10 shows the I/O and CPU usage over time for two regular mixed workloads executed in the serverless scenario with no startup delay and a 2:1 weight ratio. The blue and orange lines represent the allocation and usage, respectively, of I/O bandwidth (subfigure a) and CPU (subfigure b) for the first container, while the green and red lines show the same metrics for the second container. As shown in the I/O usage plot, the first container receives more I/O bandwidth until second 370 (600 MiB/s vs. 300 MiB/s) due to the assigned weights, even though both start simultaneously. As a result, the first workload completes its reading phase faster and starts to compute around second 330 (see the CPU usage plot). From this point, the second container fully uses the available I/O bandwidth and completes its reading phase much faster. By



(a) I/O usage



(b) CPU usage

Fig. 10 Resource usage of two concurrent regular workloads in the serverless scenario without delay and a 2:1 weight ratio

the time the second container enters its compute phase (around second 430), the first has already made significant progress in processing. The 2:1 ratio (around 2,100 CPU shares vs. 1,100 between seconds 450 and 800) allows the first container to finish faster, subsequently freeing up CPU resources. Similarly, when the second container starts its writing phase, the first container has already finished, allowing both to write to disk with-

Table 12 Runtimes for concurrent execution of a regular and a bursty workload without delay

Scenario	Workload	Weight ratio	Runtime (seconds)				
			Read	Compute	Write	Total I/O (R+W)	Total
Standard	Regular	–	265	551	261	526	1077
	Bursty		278	495	291	569	1064
Serverless	Regular	1:1	303 (+14%)	365 (-34%)	355 (+36%)	658 (+25%)	1023 **(-5%)**
	Bursty		428 (+54%)	389 (-21%)	417 (+43%)	845 (+49%)	1234 **(+16%)**
	Regular	2:1	292 (+10%)	351 (-36%)	310 (+19%)	602 (+14%)	953 **(-12%)**
	Bursty		427 (+54%)	421 (-15%)	428 (+47%)	855 (+50%)	1276 **(+20%)**
	Regular	4:1	296 (+12%)	340 (-38%)	319 (+22%)	615 (+17%)	955 (-11%)
	Bursty		371 (+33%)	421 (-15%)	458 (+57%)	829 (+46%)	1250 (+17%)

Notes: The bolded values are used to visually highlight the most relevant results in each experiment and to facilitate the interpretation of the evaluation results

out contention due to the resource desynchronisation caused by the uneven weighted approach.

5.5.2 Concurrent Regular and Bursty Workloads

The results of running a regular and a bursty workload concurrently in separate containers are shown in Table 12. Unlike the previous experiments, only executions with no delay are evaluated, as this represents the worst-case scenario for our serverless approach. The inherent resource fluctuations of the bursty workload create an overall low-contention scenario, so introducing a startup delay simply modulates the overlap between the phases of the workloads. While this may affect specific results, it does not significantly alter the overall findings of these experiments. In the standard scenario, I/O times are not greatly affected by concurrency compared to isolated execution (see Table 10), due to the low disk contention in these experiments. However, compute times remain constrained by the static CPU allocation. At first glance, the results in the serverless scenario show that the total runtime of the first container (the regular workload) always decreases, while that of the second container (bursty) increases. Conversely, I/O times tend to increase for both containers, reflecting the limited benefit of I/O scaling under low disk contention. Without prioritising one container over the other, the first workload experiences some performance loss in I/O, but this is offset by a significant reduction in compute time, resulting in a total runtime reduction of 5%. In contrast, the second workload experiences a more pronounced degradation in I/O performance; despite a reduction in compute time, the total runtime increases by 16%. In fact, this is the worst-case scenario for the serverless approach, as it faces constant scaling up and down due to the bursty workload, along with continuous adjustments of the balancing system to redistribute resources evenly. As a result, the second workload incurs significant overhead while the first benefits only marginally. Increasing the weight assigned to the first container improves its I/O performance, as it no longer has to share bandwidth equally with the second container. Combined with a reduction in compute time, this results in a total runtime reduction of up to 12%. As expected, due to its lower priority, the overall performance of the second container degrades further. Nevertheless, its total runtime only increases from 16% up to 20% compared to the equally weighted case. This higher penalty is offset by the greater gain achieved by the first container.

In summary, these results show that bursty workloads are still a challenge for serverless environments, as overheads increase significantly. Nevertheless, a serverless approach can still offer advantages over the standard scenario, mainly due to resource underutilisation caused by static CPU allocation.

Table 13 Runtimes for concurrent execution of a regular and a random workload without delay

Scenario	Workload	Weight ratio	Runtime (seconds)				
			Read	Compute	Write	Total I/O (R+W)	Total
Standard	Regular	–	238	554	195	433	987
	Random		441	554	181	622	1176
Serverless	Regular	1:1	359 **(+51%)**	475 (-14%)	230 (+18%)	589 (+36%)	1064 (+8%)
	Random		490 **(+11%)**	502 (-9%)	283 (+56%)	773 (+24%)	1275 (+8%)
	Regular	4:1	279 (+17%)	353 (-36%)	216 (+11%)	495 (+14%)	848 (-14%)
	Random		476 (+8%)	446 (-19%)	196 (+8%)	672 (+8%)	1118 (-5%)
	Regular	10:1	271 (+14%)	329 **(-41%)**	235 (+21%)	506 (+17%)	835 **(-15%)**
	Random		471 (+7%)	450 (-19%)	242 (+34%)	713 (+15%)	1163 (-1%)
	Regular	1:4	606 **(+155%)**	516 (-7%)	213 (+9%)	819 (+89%)	1335 **(+35%)**
	Random		474 **(+7%)**	373 (-33%)	240 (+33%)	714 (+15%)	1087 (-8%)
	Regular	1:10	625 **(+163%)**	465 (-16%)	224 (+15%)	849 (+96%)	1314 (+33%)
	Random		429 **(-3%)**	336 (-39%)	203 (+12%)	632 (+2%)	968 **(-18%)**

Notes: The bolded values are used to visually highlight the most relevant results in each experiment and to facilitate the interpretation of the evaluation results

5.5.3 Concurrent Regular and Random Workloads

These experiments, summarised in Table 13, involve running a regular and a random workload concurrently in separate containers. These workloads are executed without delay to simulate a high-contention scenario, while different weight ratios are evaluated to compare prioritisation of sequential versus random I/O. Compared to isolated execution (see Table 10), sequential read performance in the standard scenario is largely unaffected by concurrency, whereas random performance drops sharply. This outcome is expected, as sequential requests dominate disk usage and starve random I/O. As a result, the regular workload finishes its reading phase much earlier, causing desynchronisation, and both workloads complete their writing phases without contention, explaining the absence of overhead in writing times. Compute times mirror those reported in Table 11, where CPU usage is limited to half the allocation due to the lack of dynamic scaling.

In contrast, the serverless scenario with equal weights introduces a significant overhead for the regular workload during reads (+51%), while slightly increasing the random read overhead (+11%). This is consistent with the findings of Section 5.4: fairness enforcement can boost random bandwidth during contention and prevent starvation, but at the cost of overall performance. As a result, both reading phases take longer, as the slowed sequential progress extends the contention period. From a runtime perspective, it may be preferable to let the random workload initially starve, allowing the sequential workload to finish quickly and release the disk. However, this approach is unfair to shorter random jobs or interactive tasks that require a stable amount of bandwidth. The serverless mechanism instead offers a way to mitigate such random I/O starvation by enforcing fairness. Compute times are slightly improved under serverless due to better CPU usage, but writing times increase for both workloads because fairness keeps them synchronised, causing contention during writes. Increasing the weight of the regular workload progressively (4:1 and 10:1) reproduces the prioritisation effect of the standard scenario, albeit with some I/O overhead from scaling. Compute times decrease significantly, up to 41% for the prioritised container, and overall runtime reductions are achieved (up to 15%), mainly due to improved CPU usage. Reversing the weights (1:4 and 1:10) to prioritise the random workload has the opposite effect. Sequential reading times increase sharply (+155% and +163%), while random reads increase slightly with 1:4 (+7%) and decrease with 1:10 (–3%). Consequently, the random workload completes its reading phase before the regular workload at the cost of severely degrading sequential read performance. This creates desynchronisation, exacerbated by the uneven CPU distribution. Ultimately, the random workload achieves a runtime reduction of up to 18%, while the regular workload experiences an increase of up to 35%.

To sum up, these experiments highlight the trade-offs between fairness and performance in concurrent sequential and random I/O. Sequential streams

Table 14 Average runtimes for concurrent execution of regular mixed workloads across an increasing number of containers, each starting with a 30-second delay

# Cont	Scenario	Average runtime (seconds)				
		Read	Compute	Write	Total I/O (R+W)	Total
1	Standard	85	147	87	172	319
	Serverless	92	168	102	194	362 **(+13%)**
2	Standard	124	293	121	245	538
	Serverless	148	289	121	269	558 (+4%)
4	Standard	226	586	177	403	989
	Serverless	243	469	169	412	881 (-11%)
8	Standard	441	1176	283	724	1900
	Serverless	434	892	212	646	1538 **(-19%)**

Notes: The bolded values are used to visually highlight the most relevant results in each experiment and to facilitate the interpretation of the evaluation results

naturally starve random workloads, which can optimise overall runtimes by desynchronising resource usage, but can lead to unfair scenarios for shorter or interactive random I/O jobs. The weighted serverless approach provides greater flexibility: prioritising sequential workloads reproduces the efficient behaviour of the standard scenario, while prioritising random workloads sacrifices overall bandwidth but prevents starvation of random I/O and enhances fairness during contention.

5.6 Scalability Evaluation

The goal of the following experiments is to assess the scalability of the serverless approach as the number of concurrent workloads (i.e., containers) increases, since previous experiments were limited to at most three containers. To this end, both the standard and serverless scenarios are evaluated while progressively increasing the number of concurrent containers up to eight. In both scenarios, each container executes a regular RCW workload with a 30-second staggered start delay. To simplify the analysis, all containers in the serverless scenario are configured with equal weights. Storage is provided by the same two-disk LV used throughout the evaluation (see Table 3). To keep execution times within reasonable bounds, the regular workload configuration (see Table 9) is reduced to 60 GiB of read I/O, 60 GiB of write I/O, and a factorial computation of 3.5×10^6.

The results obtained are summarised in Table 14, which reports the average runtime of each phase across all concurrent containers using the standard scenario as the baseline. With a single container, the serverless scenario incurs some minor overhead, amplified by the relatively short workload phases for these experiments compared to previous ones. Specifically, aggregated I/O, compute, and total runtime each show an overhead of around 13%. When increasing the number of concurrent containers, average reading times initially rise, but later tend to decrease, while compute and writing times consistently improve. This reflects a desynchronisation effect, where the first containers benefit from earlier resource allocation, combined with improved CPU utilisation. The effect becomes more pronounced as the number of containers grows: in the eight-container case, compute and writing times are reduced by about 25% on average, resulting in a 19% total runtime reduction.

Figure 11 shows a heatmap of the eight-container executions, illustrating the duration of individual phases for each container. It also reports the start time (in sec-

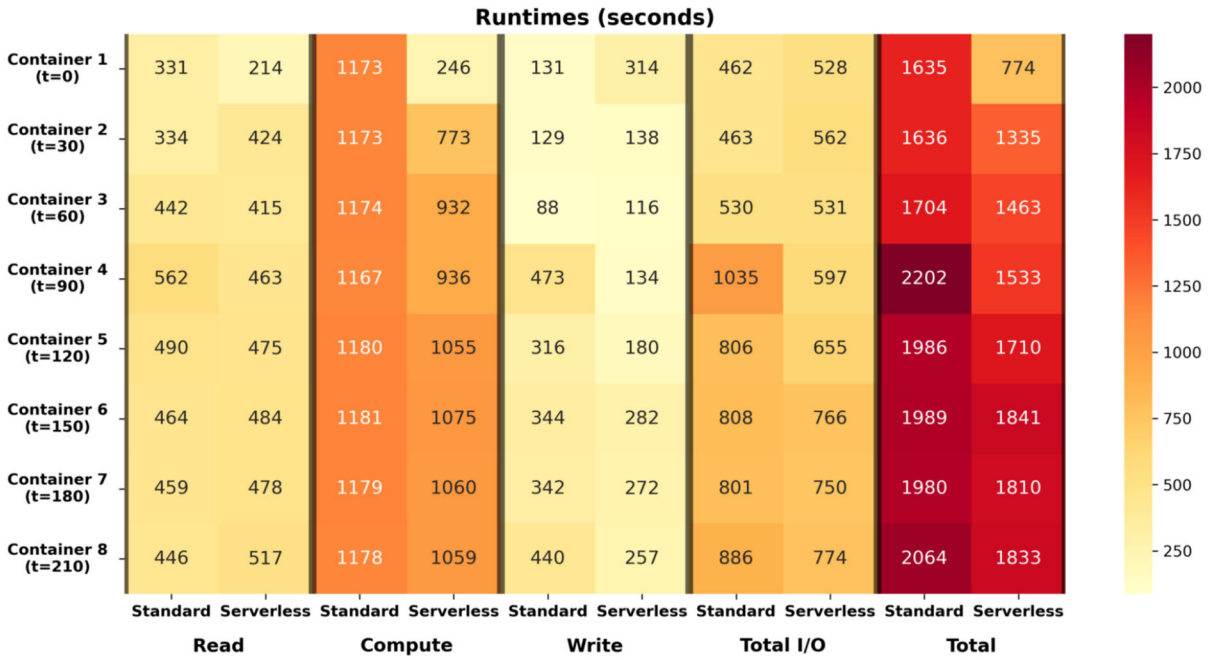


Fig. 11 Phase durations for the regular workload across eight concurrently running containers

onds) of every container, highlighting that even when the last container begins execution ($t = 210$), the first is still in its reading phase, although only for a few seconds in the serverless scenario. Overall, the standard scenario exhibits irregular I/O times: the first three containers achieve better read and significantly better write performance than later ones, as they start earlier and face less contention during reads. In contrast, later containers must compete with both earlier starters and delayed arrivals. Container 4 is the most penalised, positioned in the middle. The advantage of the first three containers allows them to write with reduced contention (or none at all for container 3), while later containers suffer from intense contention during writes. In the serverless scenario, the first container substantially reduces reading and compute times, but this comes at the cost of a significant increase in writing time, as now it partially overlaps with the reading phases of delayed containers. Even so, it achieves a significant 53% reduction in total runtime. For subsequent containers, reading times remain close to the standard scenario, but all reduce their compute times since the first container is no longer competing for CPU resources. Write performance also improves for most containers (up to 72%), with some exceptions: the second container, which faces minor scaling overheads despite no contention, and the third, which now experiences some contention compared to the standard scenario. Overall, all containers reduce their total runtimes by 7% to 53%, yielding the average reduction of 19% reported in the last row of Table 14.

In summary, these experiments demonstrate that the combined serverless approach (CPU and I/O) scales effectively to larger numbers of concurrent containers. Moreover, performance can sometimes improve due to better CPU utilisation and desynchronisation effects, where early starters gain a temporary resource advantage.

5.7 Evaluation of Hot-extension

The hot-extension mechanism described in Section 4.2 is evaluated by running a single regular workload within a container that initially relies on a single-disk LV as underlying storage. During execution, the LV is dynamically extended by adding two new disks during the reading phase, resulting in a three-disk LV (see Table 3 for the expected performance). Three different cases of hot-extension are tested: 1) early immediate extension, performed during the reading phase at approximately second 120; 2) late immediate extension, also during the reading phase, but at around second 300; and 3) deferred extension, triggered at around second 120 but automatically postponed until our system detects a drop in I/O usage (i.e., after the reading phase is complete, thus potentially overlapping with the compute phase). The regular workload was also executed under the standard scenario, where hot-extension is not available, using the initial single-disk LV to establish a baseline. The configuration parameters for the serverless approach, including those specifically related to LV extension, are reported in Table 2. For these experiments, the regular workload I/O configuration is increased to 294 GiB of read I/O and 171 GiB of write I/O, while the factorial computation remains at 5×10^6.

Table 15 shows the results obtained. In the early immediate case, the LV extension finishes during the reading phase, temporarily limiting its performance but increasing the available bandwidth for the remainder of the phase, resulting in a modest 8% reduction in reading time. Compute time is unaffected, aside from a 12% scaling overhead, and writing time decreases significantly by 60%, yielding a 29% reduction in aggregated I/O and a 21% reduction in total runtime. In the late immediate case, the extension is also completed during the reading phase, but too late to provide any ben-

Table 15 Hot-extension results in different scenarios

Scenario	Runtime (seconds)				
	Read	Compute	Write	Total I/O (R+W)	Total
Standard	612	268	414	1026	1294
Serverless (early immediate)	562 **(-8%)**	299 (+12%)	167 (-60%)	729 (-29%)	1028 **(-21%)**
Serverless (late immediate)	700 **(+14%)**	306 (+14%)	177 (-57%)	877 (-15%)	1183 **(-9%)**
Serverless (deferred)	634 **(+4%)**	301 (+12%)	172 (-58%)	806 (-21%)	1107 **(-14%)**

Notes: The bolded values are used to visually highlight the most relevant results in each experiment and to facilitate the interpretation of the evaluation results

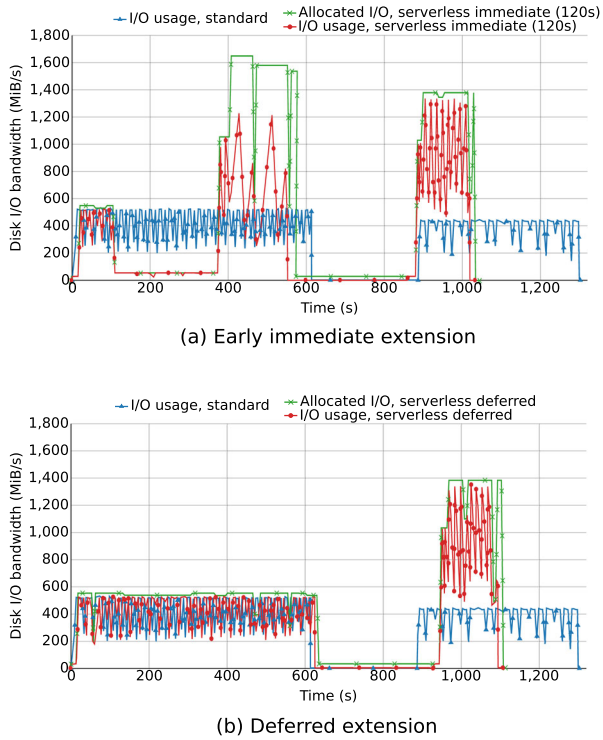


Fig. 12 I/O usage running a single regular workload in different scenarios while hot-extending the underlying LV

efit, and instead causes a 14% degradation in read performance. Compute and writing times remain largely unchanged compared to the early extension case, resulting in a total runtime reduction of around 9%. In contrast, in the deferred case, reading and compute times are only affected by scaling overheads compared to the standard scenario, while writing time improves similarly to the immediate cases, resulting in a 21% and 14% reduction in aggregated I/O and total runtime, respectively. In other words, by avoiding interference with the reading phase, the extension overlaps with the compute phase, significantly benefiting the writing phase and providing a balanced trade-off between the two immediate cases.

Figure 12 depicts two hot-extension executions: early immediate extension (subfigure a) and deferred extension (subfigure b). It shows the I/O usage for the standard scenario (blue line) alongside the bandwidth allocation and I/O usage for the serverless scenario (green and red lines). In the standard scenario, the workload is constrained by the maximum read and write bandwidth of the single-disk LV: around 520 and 430 MiB/s, respectively (see Table 3), resulting in the slowest execution due to the inability to hot-extend the LV in this scenario. In the early immediate case (subfigure a), the container initially reads from the single-disk LV at maximum speed (520 MiB/s). At around second 120, the extension process begins, during which our scaling mechanism temporarily limits the bandwidth of the container. Once the extension is complete at around second 380, the restriction is lifted and the container benefits from the increased available read bandwidth (up to 1,600 MiB/s), speeding up the reading phase, which finishes at around second 560. After the compute phase, the writing phase takes advantage of the increased write bandwidth available (up to 1,300 MiB/s). In the deferred case (subfigure b), the reading phase is limited to the single-disk LV, as in the standard scenario, because the extension is delayed until a drop in I/O usage is detected, which occurs at around second 630 when the compute phase begins. As a result, the extension process overlaps without affecting the workload, allowing the writing phase to be significantly accelerated, similar to the early immediate case.

In summary, these results show that the hot-extension mechanism provides significant improvements when workloads exploit the increased bandwidth for significant portions of their I/O time. Immediate extension,

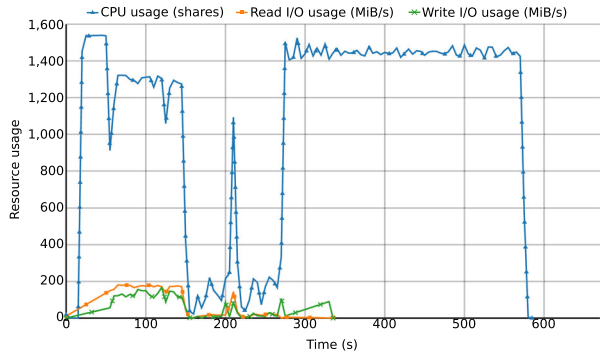


Fig. 13 Resource usage of the TSBS benchmark in the standard scenario without concurrency

despite its temporary interference, can be beneficial when applied early in I/O stages, but may lead to performance degradation when applied later. Deferred extension, by waiting for a more opportune moment, provides a safer and more consistent strategy, especially when workload duration is uncertain. Consequently, the decision-making algorithm presented in Section 4.2 adds flexibility and robustness by automatically considering LV usage together with configurable thresholds and timeouts to determine when the extension should be performed.

5.8 Evaluation with Real Workloads

This section concludes the evaluation by executing real-world workloads instead of synthetic ones, in order to assess the I/O scaling mechanism under real conditions. Since these applications typically rely on buffered I/O rather than direct I/O, cgroups v2 support is required for our mechanism to effectively control their I/O activity. Therefore, this evaluation is conducted on Testbed 2 (see Table 1), which provides cgroups v2 support. The SSD disk available in this node delivers approximately 700 MiB/s of sequential read bandwidth, 400 MiB/s of sequential write bandwidth, 120 MiB/s of random read bandwidth, and 50 MiB/s of random write bandwidth. The configuration parameters used for the scaling mechanism and its related components are largely consistent with those presented in Table 2, with certain adjustments aimed at making the scaling behaviour more conservative and reducing hysteresis effects. In particular, smaller scale-up increments (approximately 20%) were employed, together with an increased number of underutilisation windows (12) required to trigger scale-down operations.

The applications evaluated are the Time Series Benchmark Suite (TSBS) [46] and the TeraSort benchmark. TSBS evaluates the performance of a MongoDB database [47] by performing data insertion operations followed by query execution. TeraSort measures system performance through a large-scale data sorting benchmark, executed using Apache Spark [48] as the data processing engine. Both workloads are I/O-

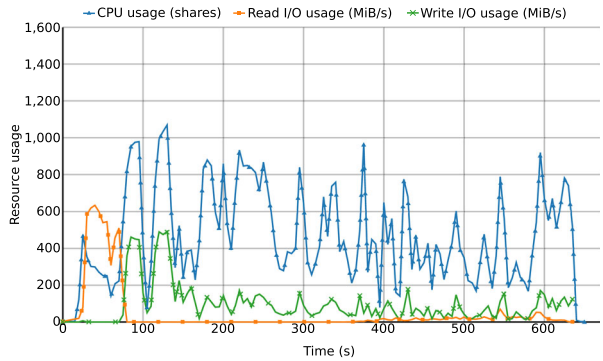


Fig. 14 Resource usage of the TeraSort benchmark in the standard scenario without concurrency

intensive: TSBS inserts an input dataset of approximately 30 GiB and subsequently executes a set of queries, whereas TeraSort operates on an input dataset of 25 GiB. TSBS exhibits a two-phase execution pattern: an initial insertion phase with moderate to high CPU utilisation, moderate read activity, and high write activity, followed by a query phase characterised by high CPU utilisation and negligible I/O activity. In contrast, TeraSort typically comprises three phases: a short initial phase with high read activity, followed by a second phase with moderate CPU usage and high write activity, and a final phase with moderate CPU usage, high write activity, and low read activity.

Figures 13 and 14 illustrate the non-concurrent executions under the standard scenario, highlighting on the Y-axis the CPU usage (blue lines) and I/O activity (orange and green lines) exhibited by both applications. Figure 13 shows that the insertion phase of TSBS accounts for roughly half of the total runtime (around 250 seconds), with the query phase occupying the remainder. Figure 14 indicates that the first and second phases of TeraSort each last around 50–60 seconds, whereas the third phase dominates the overall execution time.

Table 16 reports the total runtimes of the benchmarks executed in isolated containers (i.e., without concurrency) under the standard and serverless scenarios. A third scenario, using BFQ, is included as a weighted baseline for comparison with the serverless approach. The results indicate that the overhead remains limited for TSBS under both weight-based approaches. For TeraSort, BFQ reduces the runtime by 17%, whereas the serverless configuration increases it by 9%. To complement these results, Fig. 15 presents a box plot illustrating the distribution of runtimes across repeated runs for each workload and scenario. Consistent with the tabulated results, the observed overheads for TSBS relative to the standard scenario remain limited across repetitions. For TeraSort, BFQ generally achieves improved runtimes, whereas the serverless configuration represents the worst-case outcome. Overall, performance remains relatively stable, with only minor variability between runs and no substantial dispersion across scenarios.

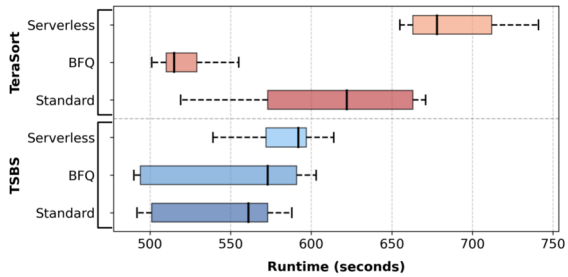


Fig. 15 Runtime distribution for real workloads executed separately

Table 17 presents the results obtained when both benchmarks run concurrently, without start-time delays, in separate containers. Under equal-weight configurations, both BFQ and serverless introduce moderate overhead for TSBS (36% and 25%, respectively), whereas TeraSort remains close to the standard scenario and even exhibits a slight improvement for BFQ.

Table 16 Runtimes for real workloads executed separately

Workload	Scenario	Runtime (seconds)
TSBS	Standard	561
	BFQ	573 (+2%)
	Serverless	592 (+6%)
TeraSort	Standard	622
	BFQ	515 **(-17%)**
	Serverless	678 (+9%)

Notes: The bolded values are used to visually highlight the most relevant results in each experiment and to facilitate the interpretation of the evaluation results

Table 17 Runtimes for concurrent execution of real workloads without delay

Scenario	Workload	Weight ratio	Runtime (seconds)
Standard	TSBS	–	1325
	TeraSort		1411
BFQ	TSBS	1:1	1805 **(+36%)**
	TeraSort		1297 (-8%)
	TSBS	8:1	1528 (+15%)
	TeraSort		1425 (+1%)
Serverless	TSBS	1:1	1654 **(+25%)**
	TeraSort		1522 (+8%)
	TSBS	8:1	658 **(-50%)**
	TeraSort		1598 **(+13%)**

Notes: The bolded values are used to visually highlight the most relevant results in each experiment and to facilitate the interpretation of the evaluation results

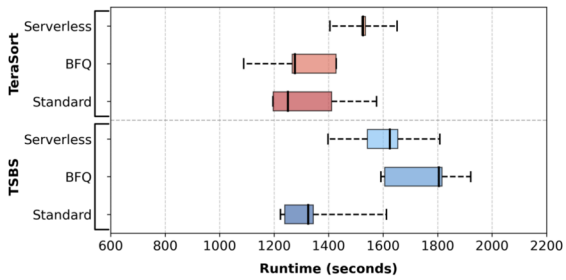


Fig. 16 Runtime distribution for real workloads executed concurrently without prioritisation

This behaviour is explained by the more demanding I/O profile of TSBS, which continuously consumes both read and write bandwidth during its insertion phase. In contrast, TeraSort primarily issues write operations during its longest phase, resulting in a less balanced and more sequential I/O demand profile. BFQ and the serverless approach were also evaluated using a weight ratio of 8:1 in favour of TSBS. This aggressive prioritisation aims to complete the insertion phase of TSBS as early as possible, thereby freeing disk resources. It also allows its CPU-bound query phase to overlap with the largely I/O-bound execution of TeraSort. Under this 8:1 configuration, BFQ mitigates the previously observed overhead for TSBS while maintaining performance for TeraSort comparable to the standard scenario. However, despite the applied weight ratio, BFQ does not yield a noticeable performance advantage for TSBS as one might expect, likely due to its complex and mixed I/O profile. In contrast, our serverless approach significantly reduces the TSBS runtime by 50%, while introducing only a minor overhead for TeraSort (13%). The limited impact on TeraSort, despite its lower priority, is consistent with its execution profile: although its I/O bandwidth is constrained during the insertion phase

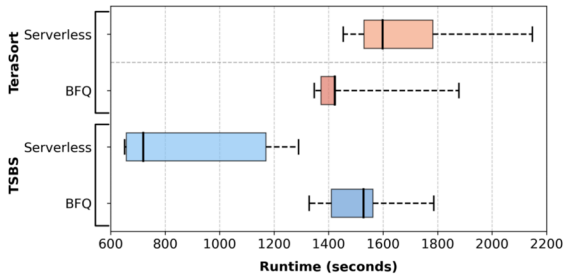


Fig. 17 Runtime distribution for real workloads executed concurrently with a 8:1 weight ratio favouring TSBS

of TSBS, it can proceed without disk contention once TSBS transitions to its CPU-bound query phase. These improvements under the serverless scenario stem from stricter I/O prioritisation through dynamic bandwidth allocation, combined with aggressive CPU prioritisation aligned with the specified weight ratio.

Figures 16 and 17 present the box plots of the runtime distributions for the concurrent experiments under the equal-weight and 8:1 configurations, respectively. Overall, the results appear more dispersed and exhibit greater variability around their median values compared with the isolated experiments. Under equal weights (see Fig. 16), the standard scenario generally yields the lowest runtimes for TSBS, while maintaining completion times similar to those of TeraSort. BFQ delivers strong performance for TeraSort, occasionally surpassing the standard scenario, albeit at the cost of increased runtime for TSBS. The serverless scenario introduces some overhead for both workloads, yet it results in a more balanced performance distribution, with TSBS and TeraSort exhibiting closer runtimes and thus more accurately resembling the proportional balance observed in the standard scenario. Under the 8:1 ratio (Fig. 17), BFQ maintains similar runtimes for both workloads despite the prioritisation. In contrast, the serverless configuration consistently delivers substantially better performance for TSBS, while avoiding severe performance degradation for TeraSort.

In summary, these results demonstrate that the serverless mechanism effectively manages the execution of real-world workloads, including those with complex I/O profiles and dependence on buffered I/O. The observed overheads are comparable to those recorded for synthetic workloads using direct I/O, indicating consistent behaviour across diverse evaluation scenarios. Furthermore, the weighted approach, which combines CPU and I/O prioritisation, achieves substantial performance improvements over I/O-exclusive mechanisms such as BFQ.

6 Conclusions

I/O bandwidth scaling can be a valuable feature for user-shared infrastructures to mitigate disk contention when data-intensive applications are executed concurrently. However, existing cluster resource managers do not support this capability. In addition, many users have moved to cloud serverless infrastructures for their

ease of use, to avoid the costs and maintenance associated with on-premises hardware, and to benefit from the pay-as-you-go billing model. Nevertheless, current FaaS-based serverless offerings may not be well suited for long-running, complex distributed workloads. While cloud providers typically offer serverless storage solutions that can scale on demand, disk I/O remains a limitation because such storage is often provided over the network. Moreover, I/O bandwidth is usually limited per cloud instance to prevent contention, rather than being dynamically scaled to meet demand.

In this paper, we have presented a serverless-like scaling mechanism for disk I/O bandwidth allocation that dynamically adjusts the bandwidth available to containers based on their actual usage. Our approach not only enables monitoring of actual usage —potentially allowing users to be charged solely for the bandwidth they consume— but also ensures that containers receive the required allocation according to their disk usage. In addition, read and/or write limits can be set to throttle certain workloads and ensure QoS for others. Dedicated strategies for managing random I/O are also included to enhance fairness and prevent starvation when random and sequential workloads execute concurrently. A weighted balancer has been implemented to prioritise specific workloads, thereby reducing contention on shared disks and increasing overall resource usage. A related feature is the ability to automatically manage extensions of virtual devices, particularly hot-extensions, enabling running containers to seamlessly benefit from increased storage performance while minimising the performance impact of the extension process. Finally, our I/O scaling mechanism has been integrated into an existing serverless platform that also supports CPU and memory scaling, thus providing a comprehensive resource scaling solution. Accordingly, the weighted balancer has been extended to support CPU and memory as well.

An extensive experimental evaluation was conducted by running I/O-intensive workloads across various scenarios to assess the capabilities of our I/O scaling approach and compare the average bandwidth and runtime with those of non-serverless executions. The results have revealed four main conclusions: 1) the overhead introduced by the serverless execution is minimal for relatively stable workloads, but can degrade performance for burstier loads, as expected due to the scaling operations; 2) the resource deconges-tion achieved by the scaling, particularly when assigning different weights to containers, is key to improving runtimes in highly contended scenarios; 3) fairness enforcement for workloads performing random I/O mitigates starvation when running concurrently with sequential ones, with the trade-off of overall performance; and 4) the hot-extension mechanism is a powerful feature that can significantly enhance the performance of running workloads with minimal interference. Experimental results show runtime reductions of up to 53% for concurrent workloads, and fairness levels of up to 87% for random workloads.

Future work includes exploring oversubscription techniques for scaling, enabling containers to temporarily exceed their configured maximum or fall below their minimum bandwidth limits, thereby improving overall disk usage. Another promising direction is to enhance the weighted balancer by adding automatic and dynamic weight adjustment based on workload consumption patterns. For example, bursty workloads could be temporarily penalised to mitigate the overhead introduced by frequent scaling operations.

Acknowledgements This work was partially conducted during a research stay at ETH Zurich under the mentorship of Prof. Gustavo Alonso, whose support and hospitality are gratefully acknowledged.

Author contributions Óscar Castellanos-Rodríguez: Conceptualization, Investigation, Software, Writing the original draft.
Roberto R. Expósito: Conceptualization, Resources, Supervision, Writing, review and editing.
Jonatan Enes: Resources, Supervision, Writing, review and editing.
Juan Touriño: Funding acquisition, Supervision, Writing, review and editing.

Funding Information Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work was supported by grants PID2022-136435NB-I00 and TED2021-129177B-I00, funded by the Ministry of Science, Innovation and Universities of Spain, MICIU/AEI/10.13039/501100011033 (PID2022 also funded by "ERDF A way of making Europe", EU, and TED2021 by "NextGenerationEU"/PRTR). It was also supported by the Consolidation Program of Competitive Reference Groups of Xunta de Galicia (ref. ED431C 2025/33). CITIC, as a centre accredited for excellence within the Galician University System and a member of the CIGUS Network, receives subsidies from the Department of Education, Science, Universities, and Vocational Training of Xunta de Galicia. Additionally, it is co-financed by the EU through the FEDER Galicia 2021-27 operational program (ref. ED431G 2023/01). O. Castellanos is also supported by Xunta de Galicia (predoctoral fellowship ED481A 2022/067). Funding for open access charge: Universidade da Coruña/CISUG.

Data Availability No datasets were generated or analysed during the current study.

Material Availability Not applicable.

Code Availability The source code of the mechanism presented in this paper is available at https://github.com/oscar-castellanos/ServerlessIOScaling.

Declarations

Competing interests The authors declare no competing interests.

Ethics approval and consent to participate Not applicable.

Consent for publication All authors have read and approved the final manuscript and consent to its publication.

References

1. Castro, P., Ishakian, V., Muthusamy, V., Slominski, A.: The rise of serverless computing. Commun. ACM **62**(12), 44–54 (2019)
2. Baldini, I., et al.: Serverless computing: Current trends and open problems. Research Advances in Cloud Computing, pp. 1–20. Springer, Singapore (2017)
3. Kounev, S., et al.: Serverless computing: What it is, and what it is not? Commun. ACM **66**(9), 80–92 (2023)
4. Yoo, A.B., Jette, M.A., Grondona, M.: Slurm: Simple Linux Utility for Resource Management. In: Proceedings of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'03), pp. 44–60 (2003). Seattle, WA, USA
5. The Linux Foundation: OpenPBS. https://www.openpbs.org/. [Visited Mar 2026] (2025)
6. SchedMD LLC: Consumable resources in Slurm. https://slurm.schedmd.com/cons_tres.html. [Visited Mar 2026] (2023)
7. Schleier-Smith, J., et al.: What serverless computing is and should become: The next phase of cloud computing. Commun. ACM **64**(5), 76–84 (2021)
8. Hellerstein, J.M., et al.: Serverless computing: One step forward, two steps back. In: Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR'19), pp. 1–9 (2019). Asilomar, CA, USA
9. Lynn, T., Rosati, P., Lejeune, A., Emeakaroha, V.: A preliminary review of enterprise serverless cloud computing (Function-as-a-Service) platforms. In: Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom'17), pp. 162–169 (2017). Hong Kong, China
10. Castellanos-Rodríguez, O., Expósito, R.R., Enes, J., Taboada, G.L., Touriño, J.: Serverless-like platform for container-based YARN clusters. Futur. Gener. Comput. Syst. **155**, 256–271 (2024)
11. The Linux Kernel Organization: Control groups. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html. [Visited Mar 2026] (2026)
12. Amazon.com, Inc.: Amazon Web Services (AWS). https://aws.amazon.com. [Visited Mar 2026] (2026)
13. Amazon.com, Inc.: AWS Fargate service. https://aws.amazon.com/fargate. [Visited Mar 2026] (2026)
14. Seltzer, M., Chen, P., Ousterhout, J.: Disk scheduling revisited. In: Proceedings of the Winter 1990 USENIX Technical Conference, pp. 313–323 (1990). Washington, DC, USA
15. Axboe, J.: Linux block IO - present and future. In: Proceedings of the 2004 Linux Symposium (OLS'04), pp. 51–61 (2004). Ottawa, ON, Canada
16. Contributors to the Ubuntu documentation wiki: Linux I/O Schedulers. https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers. [Visited Mar 2026] (2019)
17. Whitaker, C., Sundar, S., Harris, B., Altiparmak, N.: Do we still need IO schedulers for low-latency disks? In: Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'23), pp. 44–50 (2023). Boston, MA, USA
18. The Linux Kernel Organization: Control groups - Block IO Controller. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/blkio-controller.html. [Visited Mar 2026] (2026)
19. The Linux Kernel Organization: Control group v2. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html. [Visited Mar 2026] (2015)
20. Ma, S., Sun, X.-H., Raicu, I.: I/O throttling and coordination for MapReduce. Technical report, Illinois Institute of Technology (2012)
21. The Apache Software Foundation: Apache Hadoop. https://hadoop.apache.org. [Visited Mar 2026] (2024)
22. Tsujita, Y., Hori, A., Kameyama, T., Uno, A., Shoji, F., Ishikawa, Y.: Improving collective MPI-IO using topology-aware stepwise data aggregation with I/O throttling. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia'18), pp. 12–23 (2018). Chiyoda, Tokyo, Japan
23. Wu, Y., Jia, B., Qi, Z.: IO QoS: A new disk I/O scheduler module with QoS guarantee for cloud platform. In: Proceedings of the 2012 Fourth International Symposium on Information Science and Engineering (ISISE'12), pp. 441–444 (2012). Shanghai, China
24. Patel, T., Garg, R., Tiwari, D.: GIFT: A coupon based throttle-and-reward mechanism for fair and efficient I/O bandwidth management on parallel storage systems. In: Pro-

ceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20), pp. 103–119 (2020). Santa Clara, CA, USA
25. Do, N.H., Van Do, T., Farkas, L., Rotter, C.: Provisioning input and output data rates in data processing frameworks. J. Grid Comput. **18**(3), 491–506 (2020)
26. Vavilapalli, V.K., et al.: Apache Hadoop YARN: Yet Another Resource Negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC'13). Santa Clara, CA, USA. Article 5, pages 16 (2013)
27. Hindman, B., et al.: Mesos: A platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11), pp. 295–308 (2011). Boston, MA, USA
28. Liu, M., et al.: Towards low-latency I/O services for mixed workloads using ultra-low latency SSDs. In: Proceedings of the 36th ACM International Conference on Supercomputing (ICS'22). Virtual Event. Article 13, pages 12 (2022)
29. Jaliminche, L.N., Chakraborttii, C.N., Choi, C., Litz, H.: Enabling multi-tenancy on SSDs with accurate IO interference modeling. In: Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC'23), pp. 216–232 (2023). Santa Cruz, CA, USA
30. Axboe, J.: IONice. https://linux.die.net/man/1/ionice. [Visited Mar 2026] (2026)
31. Heo, T., et al.: IOCost: Block IO control for containers in datacenters. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22), pp. 595–608 (2022). Lausanne, Switzerland
32. Amazon.com, Inc.: AWS storage services. https://aws.amazon.com/products/storage. [Visited Mar 2026] (2026)
33. Amazon.com, Inc.: AWS EC2. https://aws.amazon.com/ec2. [Visited Mar 2026] (2026)
34. Amazon.com, Inc.: AWS Lambda. https://aws.amazon.com/lambda. [Visited Mar 2026] (2026)
35. Amazon.com, Inc.: AWS Storage Gateway. https://aws.amazon.com/storagegateway. [Visited Mar 2026] (2026)
36. Roy, R.B., Patel, T., Tiwari, D.: Characterizing and mitigating the I/O scalability challenges for serverless applications. In: Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC'21), pp. 74–86 (2021). Virtual Event
37. Klimovic, A., Wang, Y., Kozyrakis, C., Stuedi, P., Pfefferle, J., Trivedi, A.: Understanding ephemeral storage for serverless analytics. In: Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18), pp. 789–794 (2018). Boston, MA, USA
38. Roy, R.B., Tiwari, D.: StarShip: Mitigating I/O bottlenecks in serverless computing for scientific workflows. In: Proceedings of the ACM on Measurement and Analysis of Computing Systems **8**(1), 29 (2024)
39. Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., Kozyrakis, C.: Pocket: Elastic ephemeral storage for serverless analytics. In: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18), pp. 427–444 (2018). Carlsbad, CA, USA
40. Xie, M., Qian, C., Litz, H.: En4S: Enabling SLOs in serverless storage systems. In: Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC'24), pp. 160–177 (2024). Redmond, WA, USA
41. Barcelona-Pons, D., García-López, P., Metzler, B.: Glider: Serverless ephemeral stateful near-data computation. In: Proceedings of the 24th International Middleware Conference (Middleware'23), pp. 247–260 (2023). Bologna, Italy
42. Sreekanti, V., et al.: Cloudburst: Stateful functions-as-a-service. Proceedings of the VLDB Endowment **13**(12), 2438–2452 (2020)
43. Axboe, J.: FIO - Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html. [Visited Mar 2026] (2025)
44. Djemame, K., Parker, M., Datsev, D.: Open-source serverless architectures: An evaluation of Apache OpenWhisk. In: Proceedings of the 13th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'13), pp. 329–335 (2020). Leicester, UK
45. Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., Grot, B.: Benchmarking, analysis, and optimization of serverless function snapshots. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21), pp. 559–572 (2021). Virtual Event
46. Timescale, Inc.: Time Series Benchmark Suite (TSBS). https://github.com/timescale/tsbs. [Visited Mar 2026] (2024)
47. MongoDB, Inc.: MongoDB. https://www.mongodb.com. [Visited Mar 2026] (2026)
48. Zaharia, M., et al.: Apache Spark: A unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)