DAREDEVIL: Rescue Your Flash Storage from Inflexible Kernel Storage Stack

Junzhe Li
The University of Hong Kong

Ran Shu
Microsoft Research

Jiayi Lin
The University of Hong Kong

Qingyu Zhang
The University of Hong Kong

Ziyue Yang
Microsoft Research

Jie Zhang
Peking University
Zhongguancun Laboratory

Yongqiang Xiong
Microsoft Research

Chenxiong Qian*
The University of Hong Kong

Abstract

Existing kernel storage stacks for NVMe SSDs struggle to address performance interference between I/O requests from tenants with different SLAs, leading to the multi-tenancy issue. Addressing this requires separating their I/O requests within the NVMe I/O queues (NQs). However, our analysis reveals that the static CPU core-NQ bindings of current storage stacks restrict their flexibility to achieve this goal.

We propose DAREDEVIL, a novel kernel storage stack, which addresses this issue by decoupling the static bindings and allowing full connectivity between cores and NQs. Therefore, it grants multi-tenancy control the flexibility to freely route requests among NQs according to their SLAs. Moreover, it incorporates multi-tenancy-aware scheduling on NQs to facilitate efficient request routing. Our evaluation shows that DAREDEVIL can reduce I/O request latency by up to 3-170× compared to current kernel storage stacks, while maintaining comparable throughput.

CCS Concepts: • **Software and its engineering → Operating systems**; • **Information systems → Direct attached storage**.

Keywords: Storage systems, Solid-state drives, Linux kernel

ACM Reference Format:
Junzhe Li, Ran Shu, Jiayi Lin, Qingyu Zhang, Ziyue Yang, Jie Zhang, Yongqiang Xiong, and Chenxiong Qian. 2025. DAREDEVIL: Rescue Your Flash Storage from Inflexible Kernel Storage Stack. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3689031.3717482

*Corresponding author

EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717482

1 Introduction

Modern cloud servers host diverse I/O services for various processes, ranging from interactive web applications that intermittently fetch web pages to deep learning training workloads that periodically checkpoint model states [40, 89]. In this paper, such processes that require I/O services from the server are referred to as *tenants*.

Tenants have specific performance requirements for I/O services, according to which, they can be classified into two categories: latency-sensitive tenants (L-tenants) and throughput-oriented tenants (T-tenants). L-tenants demand timely responses to their I/O requests (L-requests) [41, 53], while T-tenants prioritize batched completion of their requests (T-requests) and are less sensitive to latency [60, 74, 98]. To meet these demands, the server's operating system (OS) is expected to enforce the processing and completion of I/O requests within the timeframes stipulated by tenants' specified service-level agreements (SLAs) [30].

To facilitate the I/O services provided to tenants, cloud servers typically use locally attached SSDs as temporary storage to deliver higher I/O performance. The local SSDs are virtualized into multiple logical instances and shared among tenants, offering more timely I/O services compared to remote storage [18, 19, 61, 78]. For instance, Google accelerates its provided I/O services by directing I/O requests to local SSDs before persisting them to remote storage [18].

This cloud server setup directs multi-tenant I/O requests to the local storage, which is then expected to deliver high-performance I/O services and satisfy diverse SLAs. To this end, Non-volatile Memory Express (NVMe) SSDs [28] are widely deployed in cloud servers as local storage because of their performance potential to achieve both low latency and high throughput [5, 27, 94]. In addition, one NVMe SSD can support multiple namespaces, each functioning as an SSD partition utilized by the server OS to logically isolate

I/O requests from multiple tenants [69]. One key feature of NVMe SSDs to achieve high performance is the multi-queue mechanism [17, 99], which allows the OS to interact with the SSDs via a set of NVMe I/O queues (NQs) in parallel during I/O services.

To leverage the parallelism offered by NVMe SSDs and utilize multiple namespaces, the Linux kernel incorporates the Multi-Queue Block IO Queueing Mechanism (`blk-mq`) [24] into its storage stack. `blk-mq` scales the multi-core architecture with the NVMe multi-queue feature by establishing static bindings between CPU cores and available NQs. For each NVMe namespace, `blk-mq` creates per-core software queues (SQs) and per-NQ hardware queues (HQs). I/O requests issued from a core traverse the corresponding SQ-HQ mapping and are strictly submitted to the SSD via the NQ associated with the HQ. This structure enables simultaneous submission of requests from multiple cores to multiple NQs, thereby exploiting the high performance of NVMe SSDs.

However, the performance potential of NVMe SSDs cannot be consistently utilized when tenants with different SLAs demand I/O services from the same SSDs. In this case, tenants are prone to suffering from the *multi-tenancy issue*. Specifically, latency-sensitive I/O services of L-tenants can be severely impacted by massive I/O requests from T-tenants when their requests become intertwined within the same NQs. This is because T-requests are typically issued in batches, processing them requires more effort from the SSDs and consequently takes longer to complete. As a result, within the same NQ, the head-of-line (HOL) T-requests can delay and even obstruct the I/O services of subsequent L-requests. This issue increases the I/O latency of L-tenants, leading to SLA violations.

Addressing the multi-tenancy issue for NVMe SSDs, as indicated by its root cause above, necessitates separating L- and T-requests within NQs. Prior works have proposed both hardware and software solutions. Hardware approaches involve redesigning the SSD internal structures to allocate distinct physical resources for L- and T-tenants [8, 38, 45, 50, 54, 75, 86, 97, 98]. However, these methods face limited applicability due to the heavy reliance on amendable SSDs' internal hardware, whereas commodity SSDs remain *black boxes*. In contrast, software approaches are applicable to black-box SSDs, as they are built upon `blk-mq` and aim to enforce NQ-level request separation within the storage stack [39, 90, 98].

Nevertheless, the effectiveness of existing software approaches is hindered by two fundamental constraints of `blk-mq`: (1) inflexibility resulting from the static bindings between cores and NQs, and (2) lack of support for multi-namespace scenarios.

Inflexibility. Limited by the static bindings between cores and NQs, previous works (e.g., D2FQ [90] and FlashShare [98]) have to *statically* overprovide NQs for each core to achieve NQ-level separation. As a result, efficiency is compromised when NQs mapped by I/O-intensive cores become overloaded while others remain underutilized. To enhance flexibility, `blk-switch` [39] performs *cross-core* scheduling to move requests or tenants among CPU cores for separation. However, the reliance on cross-core scheduling to perform multi-tenancy control mixes the optimization objectives of both, leading to intricate strategies for both mechanisms and even resulting in conflicting optimization goals (§3.2).

Lack of multi-namespace support. The `blk-mq` structure is constructed for each NVMe namespace. However, since an SSD can be divided into multiple namespaces, existing multi-tenancy control within the storage stack, which operates for each `blk-mq` structure, cannot obtain a global view of the multi-tenancy issue across all namespaces. This results in sub-optimal solutions illustrated in Figure 3c, where each namespace exclusively serves either L- or T-requests, yet performance interference persists because namespaces share the same set of NQs for I/O services, causing these requests to remain intertwined within NQs (§3.2).

To address these limitations of existing storage stacks, we propose DAREDEVIL, a novel storage stack for multi-tenancy control. DAREDEVIL decouples the static mapping between CPU cores and NQs by replacing the static I/O paths with a request routing scheme, which ***flexibly*** routes I/O requests from a core to any NQs. This grants each core unrestricted access to NQs during I/O services, achieving full connectivity between cores and NQs without relying on cross-core scheduling. Moreover, without being strictly bound to cores, each NQ is treated as an independent unit. DAREDEVIL assigns each NQ to serve either L- or T-requests and dispatches the I/O service routine accordingly. As a result, achieving NQ-level separation only involves routing requests to the NQs that serve their SLAs. To efficiently select NQs for multi-tenancy control, DAREDEVIL incorporates a lightweight NQ scheduling mechanism that integrates multi-tenancy awareness and schedules the proper NQs for request routing.

In addition, the type of requests served by each NQ is solely associated with the NQ, meaning that this information is consistent across namespaces. Therefore, DAREDEVIL's request routing scheme operates uniformly across namespaces, providing ***support for multi-namespace scenarios***.

We implemented a prototype of DAREDEVIL in the Linux kernel and evaluated it on a wide range of multi-tenant cases, including single- and multi-namespace scenarios with intensive I/O pressure. The results show that DAREDEVIL achieves significant performance improvement compared with the vanilla kernel and the state-of-the-art `blk-switch` [39] under multi-tenant I/O services. In the single-namespace case, DAREDEVIL reduces the I/O latency of L-tenants by 3-170× while maintaining comparable and stable throughput, satisfying SLAs of different tenants even under high I/O pressure. In the multi-namespace case, DAREDEVIL also reduces the 99.9^{th} tail latency and the average latency of L-tenants by up to 15×

and 39×, respectively. DAREDEVIL has been open-sourced at https://github.com/HKU-System-Security-Lab/Daredevil.

2 Background

2.1 NVMe SSDs

For each SSD, the NVMe specification supports up to 64*K* NVMe I/O queues (NQs), as SSDs' performance utilization typically scales with the number of used NQs [17, 99]. These NQs are generally placed in the OS memory region shared by the *NVMe driver* and the *NVMe controller*. The OS uses its driver to transfer data to and from SSDs by placing and retrieving data in NQs. The NVMe controller handles the data flow between NQs and the SSD. Additionally, the controller can divide the SSD into multiple *namespaces*, each providing an abstraction of space isolation and being regarded as a physical device by the OS. However, these namespaces still share the same set of NQs in I/O services [28].

Figure 1 depicts the I/O service routine. Two types of NQs are specified: NVMe submission queues (NSQs) and NVMe completion queues (NCQs). Each NSQ is strictly bound to an NCQ, forming an NQ pair, while an NCQ can be bound by one or multiple NSQs. During I/O services, the driver enqueues requests to NSQs and notifies the controller (Step ❶). The controller then fetches these requests to the SSD (Step ❷). When multiple NSQs have enqueued requests, the controller decides the order of NSQ fetching based on its *queue arbitration* mechanism, which operates in a lightweight manner to facilitate the subsequent concurrent processing of NQs [28]. For generalizability, this paper assumes the default round-robin mechanism of NVMe controllers Subsequently, the controller decomposes the fetched requests and transfers them to flash chips for final services (Step ❸) [86].

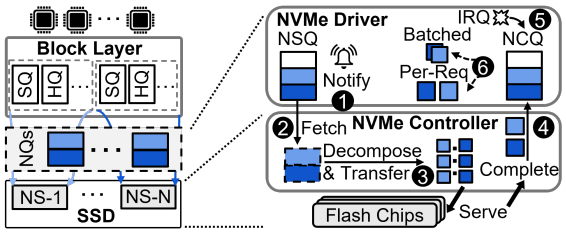


Figure 1. I/O service routine for NVMe SSDs. Different blue colors indicate I/O requests targeting different namespaces.

Upon completion, the controller places completed requests in their corresponding NCQs (Step ❹) and notifies the driver by interruption or polling [68]. We focus on the interruption approach due to its generality. Each NCQ has an interrupt request (IRQ) vector registered on a CPU core. The vector is associated with an interrupt service routine (ISR) for processing completed requests. When the controller issues an IRQ to signify request completion on an NCQ, its corresponding CPU switches to executing the associated ISR (Step ❺). Then the driver retrieves completed requests and by default, completes them in a batch (Step ❻), saving the per-request fast completion path only for those with special requirements.

2.2 The Static Kernel Storage Stack

The Multi-Queue Block IO Queueing Mechanism (`blk-mq`) [24] was introduced into the Linux kernel storage stack to exploit the NVMe multi-queue feature and mitigate the contention issue [7], in which multiple cores contend for the same NQ. `blk-mq` establishes two types of queues within its structure: software queues (SQs), each representing a CPU core, and hardware queues (HQs), each mapped to an NQ pair. As shown in Figure 3b, each SQ is exclusively mapped to one HQ while each HQ can be mapped by one or more SQs, as the Linux kernel caps the number of used NQs by available cores [59]. In this structure, requests issued from a core flow through its SQ to the mapped HQ, which then dispatches them to the mapped NQ pair. The `blk-mq` structure is created for each physical device recognized by the OS. Thus, for multi-namespace SSDs, each namespace is treated as a physical device with its own `blk-mq` structure associated.

In essence, the `blk-mq` structure constructs static bindings between CPU cores and NQs via the SQ-HQ mapping, where cores strictly use their mapped NQs in I/O services. In this way, `blk-mq` allows multiple cores to use NQs simultaneously, scaling the multi-core architecture with the NVMe multi-queue feature to exploit SSDs' superior performance. Moreover, since each core only uses one NQ for I/O services, `blk-mq` also reduces the possibility of potential contention.

2.3 The Multi-tenancy Issue

In this paper, *tenants* refer to processes requiring I/O services, including I/O containers[1] and servers' performance harvesting daemons (e.g., data analytics in computing servers). In cloud servers, tenants can share the same NVMe SSDs for their own I/O services through SSD virtualization [1, 38, 54]. For instance, Amazon Web Services (AWS) virtualizes host machines' local storage devices as multiple instance stores, which are provided to EC2 instances as temporary storage to absorb frequently updated data [78]. Google Cloud and Microsoft Azure also share their local storage of cloud servers among compute instances for high-performance services [18, 19, 61]. Additionally, an NVMe SSD can support up to 128 namespaces, each serving different tenants [67, 69, 82]. These tenants are launched by users, including cloud users (e.g., containers) and server administrators (e.g., storage performance harvesting processes [75, 100]), with specific SLAs for I/O services expected from the server OS.

Tenants with distinct SLAs behave differently. Latency-sensitive tenants (L-tenants) expect low I/O latency delivered

[1]Each container in Linux is a process sandboxed by specific Linux namespaces and kernel features [20, 23].

by the OS [39, 57, 98]. They have higher I/O rates and typically issue small requests (L-requests) that can be completed quickly, invoking frequent IRQ handling [53, 84] and thus exhibiting high CPU utilization. In contrast, throughput-oriented tenants (T-tenants) prefer batched responses for stable and comparable throughput, and can tolerate slight performance degradation [42, 74]. Their requests (T-requests) are often issued in bulk and take a long time to complete (e.g., larger than 128KB in streaming data [88])[2], consuming fewer CPU cycles due to I/O waiting.

However, the OS can fail to satisfy L-tenants' SLAs in I/O services due to the *multi-tenancy issue*, where L-requests suffer from degraded performance caused by I/O interference from T-tenants. It occurs when L- and T-requests are intertwined within the same NQs, where the head-of-line (HOL) T-requests prolong or even block the I/O services received by subsequent L-requests. Specifically, considering the bulky nature of T-requests, during submission, the NVMe controller takes longer to fetch and decompose HOL T-requests from an NSQ, and upon completion, HOL T-requests batched with L-requests within NCQs also require heavy-labor processing. Consequently, the subsequent L-requests within the same NQ experience extra in-queue latency.

Note that even though the Linux kernel employs the I/O splitting mechanism, which can decompose T-requests into multiple small ones for I/O scheduling opportunities within the block layer [11], the multi-tenancy issue persists. This is because within NQs, the split requests, consolidated together, occupy the same size of space as the original one and take up more NQ entries, mandating no less processing effort of the controller than the original bulky T-request. Thus, HOL T-requests can still incur the multi-tenancy issue.

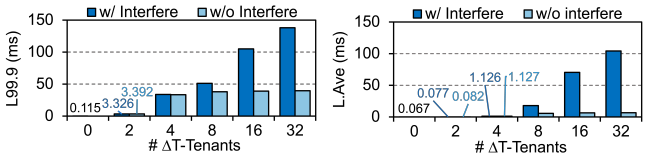
3 Motivation

3.1 The Severity of the Multi-tenancy Issue

To explore the significance of the multi-tenancy issue, we compare the performance of L-tenants with and without interfering T-tenants within the same NQs. We use the Flexible I/O Tester (FIO) benchmark [3] to launch tenants on the SV-M machine (see §7 for detailed configurations). We fix the number of L-tenants to 4, but vary the number of T-tenants. These tenants are spread across 4 CPU cores. The vanilla `blk-mq`, by design, co-locates L- and T-tenants within the same NQ and thus causes interference (*w/ Interfere*). To separate L- and T-tenants within NQs, we modify `blk-mq` so that L- and T-tenants are evenly distributed to the first and the second half of assigned NQs, respectively (*w/o Interfere*). We constrain 4 NQs for the modified `blk-mq` to align with the 4 core-NQ bindings used in vanilla `blk-mq`.

Figure 2 shows the average and the 99^{th} percentile tail latencies of L-tenants when the number of co-running T-tenants increases from 0 to 32. Compared with non-interference

[2]The size of the request is determined at the time of system calls.



(a) L-tenant 99.9^{th} tail latency. **(b)** L-tenant average latency.

Figure 2. I/O latency of L-tenants with T-tenants interfering within the same NQs (*w/ Interfere*) and using separate NQs (*w/o Interfere*).

(*w/o Interfere*), the I/O pressure from T-tenants significantly worsens the performance of L-tenants, especially when the pressure is high. The average and tail latencies of interfered L-tenants are prolonged by up to 3.49× and 15.7× (i.e., when 32 T-tenants are launched), respectively. This performance degradation evidences the longer delay experienced by L-requests and the severity of the multi-tenancy issue.

3.2 Constrained Prior Optimizations

Mitigating the multi-tenancy issue from the software level, as discussed above, requires NQ-level separation of requests with different SLAs. Prior works aim to achieve this goal but inevitably suffer from four drawbacks (see Table 1) because they are built upon the static `blk-mq` structure.

Flashshare [98] and D2FQ [90], aside from their *reliance on specific hardware support*, statically map more than one NQ per HQ, each serving either L- or T-requests (see Figure 3a). While this NQ overprovision achieves NQ-level separation, it *fails to flexibly exploit all available NQs*, leading to suboptimal cases. Specifically, under I/O-intensive scenarios, I/O-heavy cores can overload their mapped NQs, blocking subsequent requests. However, they cannot leverage the underutilized NQs mapped by I/O-light cores to alleviate I/O pressure, as the static core-NQ binding prevents one core from directly using NQs mapped by other cores for I/O services.

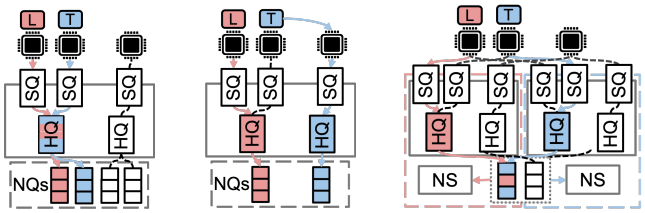
Table 1. Comparison between DAREDEVIL and prior works. "-" means the factor is not considered in the original design.

Targets	$Factor_1$	$Factor_2$	$Factor_3$	$Factor_4$
blk-mq [7]	✓	-	-	✗
Flashshare [98]	✗	✗	✓	✗
D2FQ [90]	✗	✗	✓	✗
blk-switch [39]	✓	✓	✗	✗
DAREDEVIL	✓	✓	✓	✓

$Factor_1$: hardware independence; $Factor_2$: NQ exploitation; $Factor_3$: cross-core scheduling autonomy; $Factor_4$: multi-namespace support.

Flexibly utilizing NQs within the `blk-mq` structure necessitates cross-core scheduling. In this regard, `blk-switch` [39] prioritizes I/O services of L-tenants for each SQ-HQ binding and schedules T-tenants across cores. This approach

directs T-requests to target NQs of different SQ-HQ bindings within each `blk-mq` structure, thereby separating them from L-requests (see Figure 3b). However, the *reliance on cross-core scheduling* mixes cross-core scheduling with multi-tenancy control, causing *conflicted optimization goals*, i.e., balanced CPU usage and NQ-level separation: the former goal prefers to colocate L- and T-tenants on the same cores given their complementary CPU utilization, while the latter schedules them to different cores for separation. In addition, cross-core scheduling faces constrained optimization with limited available cores as the scheduling space is small (see §7.1).



(a) NQ overprovision. **(b)** Cross-core scheduling. **(c)** Lack of multi-namespace support.

Figure 3. (a) and (b) showcase the current approaches to perform multi-tenancy control within `blk-mq`. (c) depicts their incapability to recognize the multi-namespace case.

Moreover, prior works only focus on multi-tenancy control within each `blk-mq` structure. As multi-namespace SSDs are associated with multiple `blk-mq` structures, these works *cannot observe the multi-tenant I/O services in other namespaces*, resulting in potential pitfalls. As depicted in Figure 3c, even if L- and T-tenants target different namespaces, their requests still intertwine inside NQs. This is because the HQ layer of `blk-mq` hides away the information of NQs from the block layer, which limits coordination across namespaces.

Our goal: Having identified the limitations of current storage stacks, we aim to explore a general software approach to integrate *flexibility* into the storage stack, while achieving *efficient* multi-tenancy control with *multi-namespace support*.

4 Overview of DAREDEVIL

DAREDEVIL is a generic kernel storage stack compatible with the default setting (i.e., multi-queue feature) of NVMe SSDs. It targets cloud servers with multi-tenant I/O services that access their locally attached SSDs. A typical use case is the database servers using local storage for high-performance I/O services despite the availability of remote storage [16, 93].

Figure 4 depicts the overview of DAREDEVIL. It operates transparently by determining tenants' SLAs based on standard syscalls and runtime information (§5.2), meaning that the integration into the server's OS kernel can be seamless. DAREDEVIL introduces the decoupled block layer `blex` (§5.1), based on which, two components are incorporated into the storage stack: the tenant-NQ request router `troute` (§5.2) and the NQ-level regulator `nqreg` (§5.3).

The core of DAREDEVIL to achieve flexibility and support multi-namespace is `blex`. `blex` decouples the static mapping between cores and NQs by restructuring the block layer. An SQ is still established on behalf of a CPU core but unlike `blk-mq`, it is not confined to a specific NQ pair. Each SQ has access to all NSQs with direct submission I/O paths established. Consequently, this decoupled structure allows requests to be freely routed from cores to NSQs for submission. To support multi-namespace cases, `blex` discards HQs that hide away underlying NQs and instead, exposes the states of NQs in the block layer via intermediate proxies. The proxy layer is consistently observed across namespaces, enabling multi-tenancy control to operate uniformly.

Atop `blex`, DAREDEVIL integrates `troute` and `nqreg` to handle I/O requests in multi-tenant environments. `troute` operates within the block layer and is vital in multi-tenancy control. It achieves NQ-level separation by flexibly routing L- and T-requests to different NQs. The requests are then transferred to the driver, where `nqreg` dispatches the I/O service routines based on their SLAs.

The decoupled structure of DAREDEVIL bears several immediate benefits that directly tackle the drawbacks of existing storage stacks. Besides providing consistent NQ states for *1) multi-namespace support* described above, it also achieves 2) *full exploitation of available NQs* as unconstrained accessibility to NQs is enabled, and 3) *full flexibility* because request routing operates independently to separate requests, eliminating the need for assistance of cross-core scheduling.

However, without the static bindings that establish fixed I/O paths, challenges arise in DAREDEVIL regarding its efficiency of request routing: 1) how `troute` can efficiently route requests in a lightweight manner, given the complexity of available I/O paths, and 2) how `troute` can ensure that different tenants' SLAs are effectively satisfied.

DAREDEVIL tackles the first challenge by employing the concept of *NQ heterogeneity*, which, empowered by the decoupled structure, refers to the non-uniform I/O services expected of different NQs. Specifically, because NQs are no longer confined in static bindings, DAREDEVIL regards each NQ as an independent unit. It uses `nqreg` to designate each NQ with a logical priority, which specifies the SLA of requests served by this NQ. Thus, NQ-level separation using `troute` simply involves routing issued requests to an NQ that corresponds to their SLAs (§5.2 and §5.3).

The second challenge is addressed by performing scheduling on NQs. As each NQ is treated independently, DAREDEVIL performs scheduling on them to schedule the most suitable NSQ for the upcoming requests. It integrates multiple performance-centric criteria into the scheduling process to ensure its efficiency (detailed in §5.3).

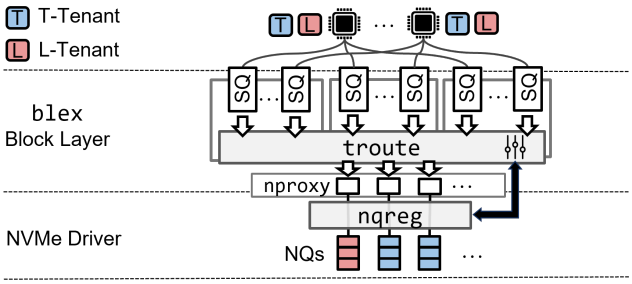


Figure 4. The overview of DAREDEVIL.

5 DAREDEVIL Design

This section details the design of DAREDEVIL. In particular, since the `blex` structure, `troute`, and `nqreg` are uniform across namespaces, we choose to describe our design on a per-namespace basis without loss of generality.

5.1 `blex`

Figure 4 shows the view of the `blex` block layer and the exposure of NQs. DAREDEVIL only exposes NSQs to the block layer since NCQs are implicitly observable due to the NSQ-NCQ binding (§2.1). However, because NSQs are managed by the NVMe driver, directly exposing NSQs blurs the boundary between the block layer and the NVMe driver, which breaks the modular implementation principle of Linux kernel [9]. To adhere to this principle, `blex` incorporates an intermediate proxy layer, `nproxy`, to cooperate with both modules while leaving the modular implementation intact.

Each `nproxy` is exclusively mapped to an NSQ. It represents the state of the NSQ, which is a combination of attributes manipulated by `nqreg` to decide the NSQ's behaviors in I/O service. Particularly, the information of each NSQ's paired NCQ, including its served SLA, is also stored in the `nproxy`. Simply put, an `nproxy` is merely a lightweight wrapper containing information of its mapped NSQ, which is used by `troute` and `nqreg` for request routing.

`blex` builds its decoupled structure atop the `nproxies`. The static binding between SQs and HQs is replaced by the full connectivity between SQs and `nproxies`. Each SQ has available I/O paths to all `nproxies`, denoting unconstrained request submission between CPU cores and NSQs. Each core can have multiple I/O paths activated, depending on the decision of `troute`. Since `nproxies` are merely wrappers around NQs, providing no queuing facility, I/O requests flowing into them are transparently propagated to NSQs for submission. Additionally, `nproxies` are device-specific and uniformly observed across namespaces.

However, this decoupled structure of `blex`, despite achieving the benefits described in §4, raises concerns regarding the potential overheads incurred by cross-core accesses to NQs: during submission, multiple cores accessing the same NSQ can contend for the NSQ entries to insert their requests; during completion, though NCQs are only processed by their associated cores, the completed requests, if submitted from different cores, are returned via cross-core interrupts. Even though such overheads have existed in existing storage stacks [7, 39], `blex` can exaggerate them as it puts no constraints on NQ accesses.

DAREDEVIL leverages two important observations to address this concern. First, request routing is vital as the choice of selected NSQs determines the severity of overheads, which in turn, serves as an important indicator of the overall fairness and efficiency in NQ usage. Second, the trend of supporting more NSQs in SSDs [83] suggests that requests from different cores can be more scattered across NSQs, reducing the possibility of contention within NSQs. Therefore, DAREDEVIL integrates multiple criteria into the NQ scheduling of `nqreg`, with `troute` considering all NQs in routing.

5.2 `troute`

`troute` is analogous to a network router, which performs multi-tenancy control by routing tenants' requests to appropriate NSQs according to their SLAs. Despite its duty being simple, `troute` is crucial in achieving efficiency as request routing occurs in the critical I/O path and determines the NQ-level separation. Thus, `troute` must perform lightweight tasks to avoid extra latency while matching the SLAs of tenants and used NQs. It carries two tasks: assessing tenants' SLAs and routing requests accordingly.

SLA assessment. As tenants are created by users, who are aware of tenants' expected SLAs, `troute` leverages the user-defined field of each tenant as the main reference for its SLA. Each tenant has an *ionice* value that signifies its SLA in I/O services. It defines the tenant's *base priority*. Tenants with real-time ionice values are treated as L-tenants, whereas those without such requirements are categorized as T-tenants. L- and T-tenants are assigned *high-* and *low-priority* as their base priority, respectively.

Requests normally inherit priorities from their parent tenants. However, T-tenants can issue *outlier* requests (i.e., synchronous or metadata ones) that have special demands for real-time responses, resulting in inconsistent SLAs between tenants and requests. Therefore, during runtime, `troute` profiles each T-tenant's I/O pattern regarding outlier requests. A T-tenant that issues at least the same order of magnitude outlier requests as normal ones is characterized as having an *outlier tendency*. Such a tenant retains its base priority as a T-tenant but receives an *outlier tag*, indicating its pattern of issuing outlier L-requests regularly. Notably, the process of determining tenants' SLAs is dynamic. Changes in ionice values or the profiling results of T-tenants adjust the SLAs determined by `troute` accordingly.

Request routing. `troute` routes issued requests to an NSQ for submission. The choice of NSQ for each request complies with the request's SLA reflected in its assigned priority. Requests with different SLAs are routed to distinct NSQs

Algorithm 1 Request Routing

```
Require: Parent tenant P; Request rq;
if P.prio == high-prio then
    route(rq, P.default)
else
    if rq.sync or rq.metadata then
        if P.tag == outlier then
            route(rq, P.outlier)
        else
            NSQ ← select_NSQ(high-prio)
            route(rq, NSQ)
```

and thereby, completed on different NCQs. This achieves NQ-level separation in multi-tenant I/O services, as `troute` ensures that requests with diverse performance demands receive corresponding I/O service within NQs.

As NSQs are managed by the NVMe driver, `troute` needs to query `nqreg` to get the selected NSQ for request routing. `nqreg`, as detailed later in §5.3, performs multi-tenancy-aware NQ scheduling to select the most suitable NSQ for `troute`. However, regularly invoking NQ scheduling can incur great overheads (§7.5). To reduce the frequency of querying `nqreg`, `troute` distinguishes different *contexts* of issued requests. Typically, issued requests are normal ones as they share the same SLAs as their tenants. Only the outlier requests issued from T-tenants require exceptional attention. The two cases are referred to as the *tenant-based* context and the *request-specific* context, respectively. The contexts determine the strategies used by `troute`.

Algorithm 1 presents the procedure of context-specific request routing. Each tenant is assigned a *default NSQ* during initialization, in which `troute` invokes NQ scheduling with the tenant's base priority and assigns the returned NSQ as its default NSQ. A tenant's default NSQ can be dynamically updated during runtime once `troute` detects changes in its base priority. `troute` initiates the update process along with the kernel's original routine of changing a tenant's ionice value, which occurs asynchronously to the critical I/O service path. Thus, in runtime updates, `troute` only introduces the overhead of one extra query to `nqreg`, avoids directly interfering with I/O services, and also synchronizes tenants' default NSQs with their changes in ionice values.

Figure 5 depicts the I/O paths used by tenants with different SLAs and contexts. In the tenant-based context, requests issued from a tenant are directly routed to its default NSQ since both share the same priority. However, in the request-specific context, the default NSQ is not eligible. The SLA of outlier requests does not align with the I/O service expected of the T-tenant's default NSQ. In this case, `troute` differentiates its request routing strategy depending on the presence of outlier tags. Tagged T-tenants imply frequent submission of outlier requests. Therefore, for such T-tenants, aside from their default NSQs, `troute` assigns each one with an *outlier NSQ* obtained by querying `nqreg` with high priority. Their outlier requests are directly routed to the outlier NSQs, while normal requests remain routed to the default NSQs. On the other hand, T-tenants without the tag infrequently issue outlier requests. Thus, `troute` treats these tenants' outlier requests as unusual conditions and queries `nqreg` with high priority to fetch an NSQ on a per-request basis.

Apart from merely querying `nqreg` in NSQ selection, `troute` provides feedback to assist in NQ scheduling. During each query, `troute` notifies `nqreg` of the calling contexts. `nqreg` utilizes this information to decide the update timing of its NQ scheduling algorithm, detailed later in §5.3. `troute` also records the distribution of CPU cores in NQ usage for `nqreg`. It maintains a bitmap for each NSQ, which tracks the cores of tenants that use this NSQ as the default or outlier NSQ. This bitmap encodes the cores that claim frequent usage of the NSQ and thereby, implies the potential submission-side contention. Leveraging this information, NQ scheduling becomes aware of potential contention and can adjust its scheduling process accordingly.

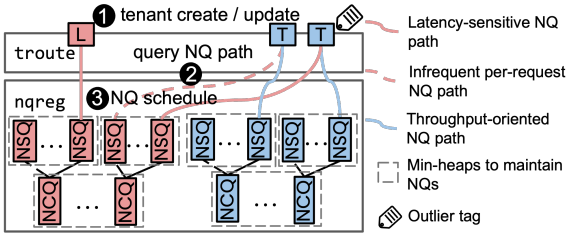


Figure 5. I/O paths of tenants atop NQ heterogeneity.

5.3 `nqreg`

`nqreg` regulates the behaviors of NQs in I/O services by deciding the following attributes of each NQ: 1) the priority, 2) the timing of notifying the controller of enqueued requests, and 3) the request completion path. The first attribute is common for both NSQs and NCQs. The second and third attributes are specified for NSQs and NCQs, respectively.

NQ heterogeneity. `nqreg` assigns each NQ with a priority and manages them accordingly. Figure 5 also illustrates the layout of NQs in DAREDEVIL. `nqreg` treats NSQs and NCQs separately and maintains a hierarchical logical layout atop the NQ heterogeneity. During the initialization of the NVMe driver, as `nqreg` cannot foresee the number of future running L- and T-tenants, it conservatively divides NCQs and their attached NSQs into two *NQGroups* with equal division. Each NQgroup is assigned a priority specifying the SLAs its NQs serve in I/O services. The high- and low-priority NQGroups are designated to L- and T-requests, respectively. Within each NQGroup, `nqreg` *logically* organizes NQs into a two-level hierarchy: the NQGroup serves as the root that branches to NCQs, with each NCQ attached by NSQs as leaves. This hierarchy facilitates the NQ scheduling below.

The NQ heterogeneity forms the foundation for the roles undertaken by nqreg: NQ scheduling to select NSQs and I/O service dispatching to accommodate different SLAs.

NQ scheduling. Upon receiving a priority from troute, nqreg selects an NSQ by scheduling within the corresponding NQGroup. The selected NSQ also implies the choice of NCQs as indicated in the NQ hierarchy (Figure 5). This ensures that requests with different SLAs are separated within NQGroups. However, ineffective NQ scheduling can lead to uneven request distribution across NQs, which results in inefficient NQ usage that degrades performance (e.g., inflated latency as certain NQs are overloaded), as well as worsens the potential cross-core overheads: imbalanced completion IRQs among cores and increased contention within NSQs.

Fortunately, cross-core overheads inherently indicate the distribution of request scattering. For instance, less contention implies more dispersed tenants' requests, and vice versa. Therefore, nqreg integrates two criteria, i.e., IRQ-balancing and contention reduction, into NQ scheduling. It adopts a two-step procedure that decouples the selection of NCQs and NSQs. Each step schedules one type of NQs based on the corresponding criterion, which is reflected in the *merits* of NQs as elaborated below.

Algorithm 2 depicts the scheduling procedure. The scheduling process is initiated by troute (cf. §5.2) with a given target priority. It operates within the NQGroup of this priority. The NSQ that best satisfies the criteria in this NQGroup is returned to troute. It first retrieves an NCQ from the NQGroup and proceeds to select an NSQ from the NSQs attached to the chosen NCQ (Line 18-20). At each step, the target NQs are organized as a priority array using a min-heap data structure [2], which is sorted based on the *merit* of each NQ. Notably, in the case of 1:1 NSQ-NCQ binding, the min-heap in the second step degenerates to a single NSQ, which is directly selected without further scheduling.

The *merit* of an NQ measures its contribution made in serving I/O requests. In each step, the NQ with the lowest merit is scheduled to serve forthcoming requests, compensating for its relatively inadequate contribution to I/O service. The calculation of merits employs the *exponential smoothing* algorithm [32, 64] (Line 7) to thoroughly consider an NQ's historical and recent contribution in I/O service. It also avoids bursts of merit values, which may result in an inaccurate assessment of contributions. The algorithm applies a decay ratio, i.e., α that ranges within $(0.5, 1)$, on the NQ's historical merit ($(1-\alpha) \times \text{merit}_{k-1}$), which utilizes the NQ's historical information, while emphasizing its contribution made within the recent period ($\alpha \times \text{merit}_k$).

Calculating merit_k varies according to the type of NQs, considering the different criteria expected of each type.

• *NCQ merit.* The merit of an NCQ quantifies its contribution in terms of IRQ-balancing (Line 4). This merit is determined using the intensity of incoming and average per-interrupt completed requests, which reflect the NCQ's future and past contributions to request completion. The incoming intensity is calculated by the number of outstanding requests divided by the queue depth (i.e., $\frac{\text{nq.in\_flight\_rqs}}{\text{nq.depth}}$), as these requests are expected to complete soon. The average intensity is the ratio of the number of completed requests to received IRQs (i.e., $\frac{\text{nq.complete\_rqs}}{\text{nq.irqs}}$). The product of these two ratios and the number of IRQs determines the NCQ's merit.

• *NSQ merit.* An NSQ's merit measures its sufferings in terms of NSQ contention (Line 6). The ratio of the time consumed in contention to the number of submitted requests (i.e., $\frac{\text{nq.in\_contention\_us}}{\text{nq.submitted\_rqs}}$) represents the per-request latency caused by contention. This ratio is multiplied by the number of cores claiming usage of this NSQ (nq.claimed_cores), which is derived from the NSQ's bitmap of CPU cores (§5.2). Their product approximates the summation of latency in the worst scenario, where each core contends for the NSQ. Thus, an NSQ with higher merit indicates potentially more severe contention, rendering itself less likely to be selected.

In each step, the merits of NQs are calculated during min-heap updates, where nqreg re-sorts the min-heap using the calculated merits of all NQs (Line 13 and 14). Each update schedules the NQ with the lowest merit as the new top. However, frequent updates can incur substantial CPU overheads. To avoid this, nqreg employs the Most-Recently-Used (MRU) policy. Each min-heap has a mru value to control the update frequency and remains unchanged until the mru is exhausted. Each time the top NQ is selected, the mru (initialized with an empirical value **MRU**) is decremented by m (Line 11), which is set by troute based on different contexts.

In the tenant-based context and the request-specific context of tagged T-tenants, troute invokes NQ scheduling

Algorithm 2 NQ Scheduling

```
1:  function MeritCalc(nq)
2:      merit_{k-1} ← nq.merit
3:      if nq.type == NCQ then
4:          merit_k ← (nq.in_flight_rqs / nq.depth + nq.complete_rqs / nq.irqs) × nq.irqs
5:      if nq.type == NSQ then
6:          merit_k ← nq.in_lock_μs / nq.submitted_rqs × nq.nr_claimed_cores
7:      merit ← α × merit_k + (1 − α) × merit_{k-1}
8:      return merit
9:  function FetchTop(merit_heap, m)
10:     top ← merit_heap.top
11:     merit_heap.mru −= m
12:     if merit_heap.mru ≤ 0 then
13:         calc_each(merit_heap, MeritCalc)
14:         merit_heap.re_sort()
15:         merit_heap.mru ← MRU
16:     return top
17: function NQSchedule(prio, nqreg, m)
18:     NQGroup ← nqreg.groups[prio]
19:     ncq ← FetchTop(NQGroup.merit_heap, m)
20:     nsq ← FetchTop(ncq.merit_heap, m)
21:     return nsq
```

to get default or outlier NSQs that regularly handle I/O requests. It sets m to MRU so that each min-heap is updated and schedules a new top NQ for future requests. This helps to distribute tenants to use different NQs. In the request-specific context of normal T-tenants, m is set to 1 in both steps as the returned NQ is accessed infrequently.

In brief, each step of NQ scheduling selects an NQ that best fulfills the selection criteria. Consolidated together, `nqreg` not only refrains from incurring potential I/O intertwining but also delivers better performance.

SLA-aware I/O service dispatching. In the submission and completion process within NQs, `nqreg` dispatches the requests' I/O service routines corresponding to their SLAs.

• *Submission.* `nqreg` accelerates the submission process in high-priority NSQs by immediately notifying the controller once L-requests are enqueued. For low-priority NSQs that handle batched T-requests, `nqreg` postpones the notification to the controller until each batch of T-requests is enqueued.

• *Completion.* With the decoupled storage stack, each NQ only serves requests of the same SLA. Therefore, upon handling request completion IRQs, `nqreg` distinguishes and dispatches completion paths simply based on the priorities of NCQs, i.e., per-request and batched completion paths for high-priority and low-priority NCQs, respectively. This approach saves `nqreg` from the burden of considering the per-request priority as seen in cinterrupts [84] and customizes the completion service routine within each NCQ.

6 Implementation

We implement a prototype of DAREDEVIL in the Linux kernel v6.1.53, with modifications to the block layer and the NVMe device driver. Our current prototype is carefully crafted to optimize only the storage stack of NVMe SSDs attached to the host via the PCIe bus. This ensures that DAREDEVIL does not affect other types of storage devices, to which it is inapplicable (e.g., SATA SSDs [70] with only one device I/O queue). However, integrating the DAREDEVIL into the Linux kernel is non-trivial, especially considering that `troute` and `nqreg` operate within the critical I/O path. We summarize the key implementation details of accommodating DAREDEVIL in the kernel below.

Multi-threaded tenants. In the Linux kernel, each tenant (i.e., process) is represented by a process descriptor, `struct task_struct`, which contains per-descriptor fields like *ionice* values [9]. Thus, for each `task_struct`, DAREDEVIL extracts its SLA information from its fields and assigns the default or outlier NSQ to it. This implementation extends the capability of DAREDEVIL to handle multi-threaded tenants at the thread granularity. Specifically, the Linux kernel treats threads as lightweight processes. Each thread spawned by a tenant is also allocated a `task_struct` associated with its parent tenant [9, 22]. Utilizing this mechanism, DAREDEVIL can recognize the spawned threads of each multi-threaded tenant and manage them at the thread granularity, assessing each thread's SLA and performing request routing accordingly.

Identifying outlier L-requests. As noted in §5.2, DAREDEVIL identifies synchronous or metadata requests issued from T-tenants as the outlier L-requests. This design aligns with how such requests are managed by the Linux kernel: at the block layer, requests flagged with `REQ_SYNC` (synchronous) and `REQ_META` (metadata) are regarded and served as high-priority ones (i.e., `REQ_HIPRIO`). Therefore, DAREDEVIL directly recognizes outlier L-requests by checking their flags.

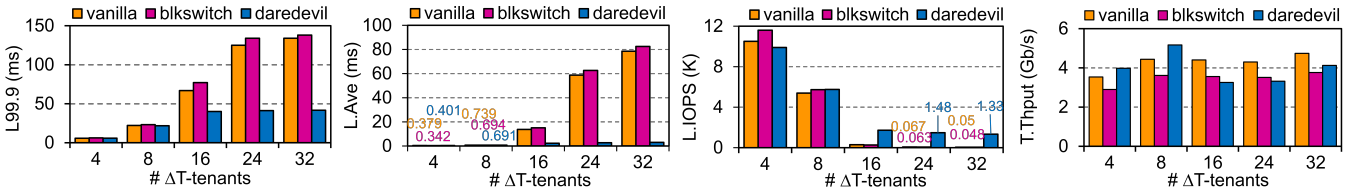
Concurrent `nqreg` queries with NQ scheduling. In request routing, multiple CPU cores can have tenants requiring default or outlier NSQs. This means that `nqreg` can be concurrently queried by `troute`, resulting in concurrent NQ scheduling, i.e., simultaneous NQ heap querying and updating. Without careful consideration, this concurrency can result in inconsistent NQ heaps, while naively using mutual exclusion locks (e.g., `mutex`) can damage DAREDEVIL's performance. To properly handle this concurrency, we utilize the lightweight Read-Copy-Update (RCU) synchronization primitive [87] to protect the NQ heap, based on the following observations: 1) RCU ensures consistent reads and does not block reads when synchronizing with updates [62, 87], which matches the objective of `nqreg` as it aims for lightweight NQ heap queries; and 2) RCU is optimized for read-mostly scenarios [62, 87], which aligns with the purpose of the MRU policy (i.e., reduce the update frequency) in NQ scheduling.

7 Evaluation

Comparison targets. We compare DAREDEVIL against the vanilla kernel storage stack and the state-of-the-art storage stack design, `blk-switch` [39]. We use the Linux v6.1.53 kernel with `blk-mq` and the default `noop` I/O scheduler enabled as vanilla. This is the baseline of our evaluation. For `blk-switch`, we carefully migrated its open-source implementation [33] from v5.4.43 to the v6.1.53 kernel.

Parameter setup. For `blk-switch`, we set its scheduling thresholds to its suggested values and enable its highest optimization level [39]. For DAREDEVIL, we set the weight α used in NQ scheduling to 0.8, as it achieves the best balance between leveraging historical and recent information in our practice. **MRU** equals the NQ depth, which is 1024 in our tested SSDs. The min-heaps are updated until accumulated outlier requests from regular T-tenants reach the NQ depth.

Server setup. We use a server machine (SV-M) equipped with 64GB DRAM and 64 physical AMD EPYC 7702P processor cores within one socket, each running at 3.3 GHz. We use an enterprise-level 3.2TB Samsung PM1735 NVMe SSD that supports at most 64 NQs and 32 NVMe namespaces. This SSD is locally attached to the server via the PCIe lanes. This configuration effectively emulates the typical cloud server setup as commonly observed in prior works [16, 30, 51, 93].



(a) L-tenant 99.9th tail latency. **(b)** L-tenant average latency. **(c)** L-tenant I/O rate. **(d)** T-tenant throughput.

Figure 6. Performance results with increasing T-pressure in SV-M. DAREDEVIL maintains in-time responses for L-tenants even under extreme T-pressure, while the vanilla kernel and `blk-switch` significantly inflate the L-tenants' I/O latency.

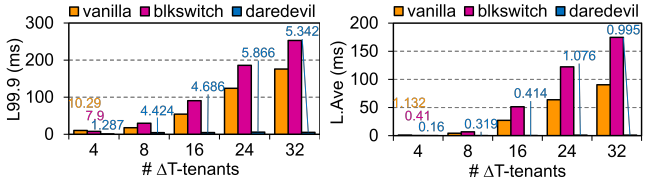
Complimentary setup. To evaluate the benefits of NQ scheduling when the supported NQs outnumber available CPU cores, which enables more fine-grained NQ scheduling as introduced in §5.3), we use a complimentary setup in §7.1. We use a workstation (WS-M) with an Intel 13th Generation Core I9-13900K processor that comprises 8 P-cores and 16 E-cores. We attach a 2TB Samsung 980Pro NVMe SSD that supports at most 128 NQs (5× the CPU core count) to this machine. Note that this setup only aims to demonstrate the effectiveness of DAREDEVIL in using more NQs. Our major evaluation is conducted on SV-M, which corresponds to the server environments targeted by this paper.

Both machines run Ubuntu 22.04 LTS and have their CPU hyper-threading disabled to minimize uncontrolled disturbance. In particular, for WS-M, we only use its P-cores to reduce interference from asymmetric cores. Before each experiment, we pre-condition the whole disk [85, 92] to ensure stable SSD internal states.

7.1 Resistance to Severe Multi-tenancy Issue

We evaluate the comparison targets' performance with the increasing severity of the multi-tenancy issue, focusing on whether different tenants' SLAs are met, especially the low latency expected by L-tenants. Following the standard practice of prior works [39, 42, 64, 84], we use the Flexible I/O Tester (FIO) benchmark [3, 4] to generate I/O requests for both L- and T-tenants. FIO jobs issuing 4KB random requests with 1 I/O depth simulate L-tenants, reflecting the random distribution of small L-requests observed in real-time workloads [58]. T-tenants are simulated using FIO jobs that issue 128KB requests with 32 I/O depth. We run 4 L-tenants for 50 minutes, during which the number of T-tenants increases every 10 minutes to impose higher performance interference on L-tenants. All tenants are distributed evenly across a shared pool of 4 cores and asynchronously issue requests using the libaio I/O engine. L- and T-tenants are assigned high (real-time) and low (best-effort) ionice values, respectively. We use one namespace for the SSD in this experiment.

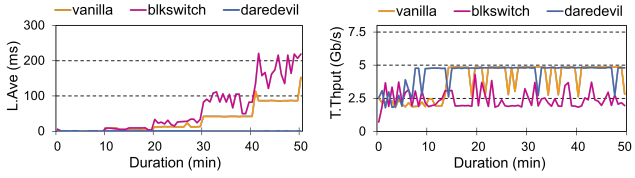
Figure 6 and Figure 7 show the performance of the targets running on SV-M and WS-M, respectively. Compared with the vanilla kernel and `blk-switch`, DAREDEVIL significantly reduces the 99.9th tail latency and average latency for L-tenants by up to 3.2× and 33× in SV-M, and 40× and 170× in WS-M. This means DAREDEVIL satisfies L-tenants' SLAs even under very high pressure of interference. Meanwhile, DAREDEVIL maintains a stable and comparable throughput to meet the SLAs of T-tenants. In contrast, both vanilla and `blk-switch` rapidly inflate the L-tenants as T-pressure rises, failing to satisfy the SLAs of L-tenants.



(a) L-tenant 99.9th tail latency. **(b)** L-tenant average latency.

Figure 7. I/O latency with increasing T-pressure in WS-M.

The flexibility of DAREDEVIL benefits efficient multi-tenancy control by being resilient to HOL blocking of T-requests. Figure 6c shows that under high T-pressure, L-tenants running on both vanilla and `blk-switch` can hardly issue any I/O requests due to blockage from excessive HOL T-requests within the same NQs. On the contrary, with the decoupled storage stack and the independent request routing, DAREDEVIL easily separates L- and T-requests within NQs, avoiding blocking L-requests.



(a) L-tenant average latency. **(b)** T-tenant throughput.

Figure 8. The performance of the comparison targets running on WS-M within the duration of increasing T-pressure.

DAREDEVIL behaves better with more NSQs available for NQ scheduling. DAREDEVIL achieves more significant

performance improvement over vanilla and `blk-switch` in WS-M (Figure 7) than in SV-M (Figure 6). This is because WS-M provides more space for request routing. WS-M supports 128 NSQs and 24 NCQs compared with 64 NSQs versus 64 NCQs in SV-M. Thus, DAREDEVIL obtains more possible NSQs in WS-M to issue its requests and has at least 5 NSQs attached to each NCQ. DAREDEVIL can scatter both L- and T-requests more spread out across the NSQs and facilitates NQ scheduling, thus achieving further improved performance.

Reliance on cross-core scheduling fails under severe performance interference. In Figure 6, `blk-switch` reduces L-tenants' latency under low T-pressure by scheduling T-tenants to proper cores when the scheduling space is relatively small (e.g., 4 cores and 4 T-tenants). However, with increasing T-pressure, `blk-switch` cannot achieve optimization and even becomes paralyzed. As shown in Figure 8, the average latency and throughput keep fluctuating in `blk-switch`. This is because `blk-switch`'s attempts to perform cross-core scheduling frequently fail due to the high number of T-tenants and limited cores, incurring overheads that result in fluctuating performance.

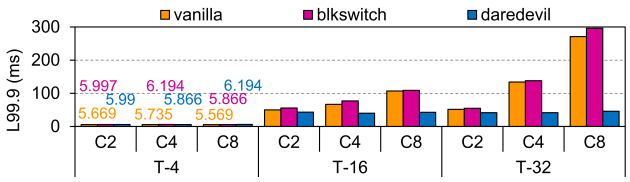


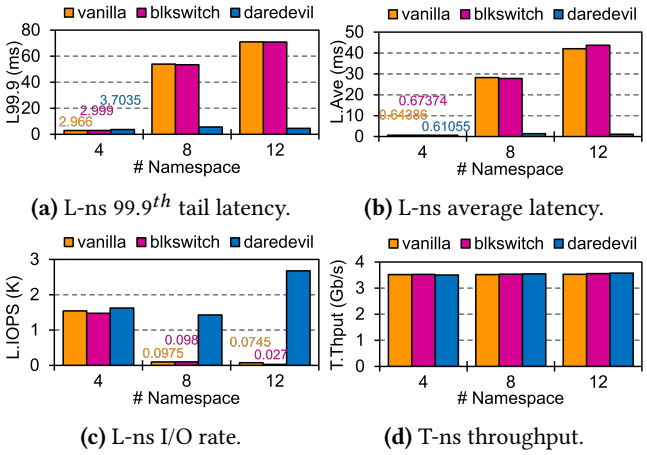
Figure 9. The 99.9^{th} tail latency of L-tenants under different T-pressure with 2,4,8 CPU cores available. The results are collected from SV-M. WS-M also shows similar patterns.

DAREDEVIL performs consistently regardless of available CPU cores. Figure 9 shows the comparison targets' sensitivity to CPU core usage, using L-tenants' tail latency as the performance indicator. As `troute` functions independently, DAREDEVIL maintains low latency for L-tenants regardless of CPU core usage. Particularly, under high T-pressure, DAREDEVIL performs better with more available cores while `blk-switch` worsens its performance. In this case, more CPU cores overwhelm the cross-core scheduling space of `blk-switch` but instead, alleviate the burden of handling the large number of T-tenants in DAREDEVIL, rendering more CPU power released to utilize its benefits.

DAREDEVIL keeps different tenants' SLAs despite some performance degradation. It incurs higher latency under low T-pressure (e.g., ~0.02-0.06*ms* in Figure 6b) and delivers lower throughput under high T-pressure (e.g., 1.7%-25.9% lower in Figure 6d). In the former case, DAREDEVIL incurs ~2.29% extra CPU costs for L-tenants due to the cross-core request completion. For the latter, T-tenants receive 0.3× less CPU utilization in DAREDEVIL because co-located L-tenants still run normally, consuming host CPU and SSD controller resources. Nonetheless, the cross-core overheads are small and outweighed by the benefits once T-pressure increases; T-tenants still receive stable and comparable throughput with only at worst one-fourth degradation.

7.2 Support for Multi-namespace Scenarios

This experiment considers the multi-namespace case, where each namespace hosts only L- or T-tenants but the multi-tenancy issue persists (§3.2). We vary the number of created namespaces by 4, 8, and 12. Using more namespaces can cause unstable performance due to SSD fragmentation [26, 71]. The ratio of namespaces that serve L- and T-tenants, i.e., L-ns and T-ns, is fixed at 1 : 3 considering the relatively small space occupied by L-tenants [58]. Each L-ns hosts 2 L-tenants, whereas each T-ns hosts 8 T-tenants.



(a) L-ns 99.9^{th} tail latency. **(b)** L-ns average latency.

(c) L-ns I/O rate. **(d)** T-ns throughput.

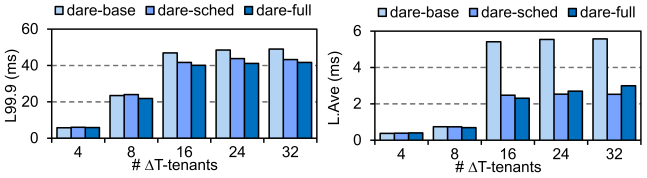
Figure 10. Performance results with different numbers of namespaces created. DAREDEVIL improves the performance for L-tenants and maintains comparable throughput.

DAREDEVIL keeps its performance promises under multi-namespace scenarios whereas vanilla and `blk-switch` do not. Figure 10 shows that with more namespaces used, DAREDEVIL keeps the L-tenants' tail latency below 10*ms* and the average latency around 1*ms*, reducing them by up to 15.3× and 39.3× compared with vanilla and `blk-switch`. It also maintains comparable throughput to that of vanilla. In contrast, vanilla and `blk-switch` inflate the I/O latency. This is because without supporting multi-namespace cases, more severe I/O interference is incurred within NQs with more namespaces created, while DAREDEVIL, being aware of this case, continues to demonstrate effectiveness.

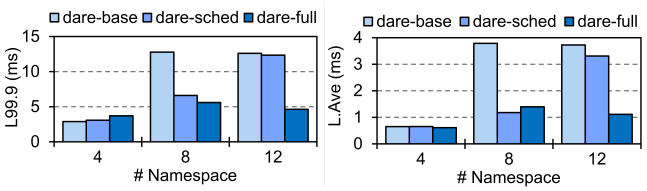
7.3 Understanding the Optimization of DAREDEVIL

We analyze DAREDEVIL's performance gains by decomposing its optimization techniques into three subsystems: dare-base that only enables the decoupled block layer and round-robins NQs for request routing, dare-sched with NQ scheduling

enabled atop dare-base, and dare-full, which is tested in §7.1 and §7.2 and adds I/O service dispatching atop dare-sched.



(a) L-tenant 99.9^{th} tail latency with increasing T-pressure. **(b)** L-tenant average latency with increasing T-pressure.



(c) L-tenant 99.9^{th} tail latency with varying namespaces used. **(d)** L-tenant average latency with varying namespaces used.

Figure 11. The decomposition of DAREDEVIL's performance according to its optimization techniques.

Figure 11 shows the latency performance of different DAREDEVIL subsystems. By adopting the decoupled structure and simple request routing, dare-base already achieves comparable tail latency as dare-full (i.e., ~47*ms* compared with ~40*ms*), and maintains a stable low average latency of below 6*ms*. A similar pattern is also observed in the multi-namespace scenario, evidencing the efficiency of decoupling the storage stack and independent request routing in resisting HOL blockage and enhancing performance.

dare-sched further improves the latency performance atop dare-base (e.g., 2-4× reduction in average latency), showing the effectiveness of NQ scheduling. However, the optimization of dare-full varies with the T-pressure. dare-full improves the tail latency except under low T-pressure, but slightly worsens average latency by at most 18% under high T-pressure. This is because accelerating L-requests improves their in-NQ tail latency but consumes extra CPU cycles, which, under high T-pressure where cores are busy switching between processing T-tenants, reduces L-tenants' CPU resources and thus increases their average latency.

7.4 Real-world Applicability

We select two representative benchmarks to simulate real-world storage workloads: the Yahoo! Cloud Serving Benchmark (YCSB) for cloud workloads and the Filebench [88] for file operations. We test YCSB workloads of types A, B, E, and F on RocksDB [29], as they cover various runtime scenarios. We use the Mailserver workload of Filebench as its operations involve direct I/O requests to the SSD. The NVMe SSD is configured with the ext4 file system. Both applications are configured per official suggestions and common practices [12, 14, 21]. Each YCSB workload uses a 64GB database and issues 4 million requests under the Zipfian distribution. The Mailserver workload constructs a 100GB directory with an average file size of 16KB. We regard the processes associated with each application as L-tenants, considering that they aim to promptly serve user requests. We colocate 8 background FIO jobs issuing streaming I/O as T-tenants along with the two applications on 4 cores.

RocksDB with YCSB. Figure 12a to Figure 12d present 99.9^{th} tail latency performance under different YCSB workloads. It improves the tail latency of updates in YCSB-A by 2× compared with `blk-switch`. In YCSB-F, DAREDEVIL achieves a reduction compared with vanilla. However, DAREDEVIL slightly worsens the latency observed by RocksDB in some workloads (e.g., increases 0.4*ms* of inserts in YCSB-E).

Mailserver. Mailserver reports the average latency of its operations, which consist of ones that mainly use ext4 page caches and ones that directly interact with SSDs. Thus, we explicitly show the latter (fsync and delete) in Figure 12e. Compared with vanilla and `blk-switch`, DAREDEVIL improves the latency by 2-3*ms* for fsync and 0.5-1.2*ms* for delete.

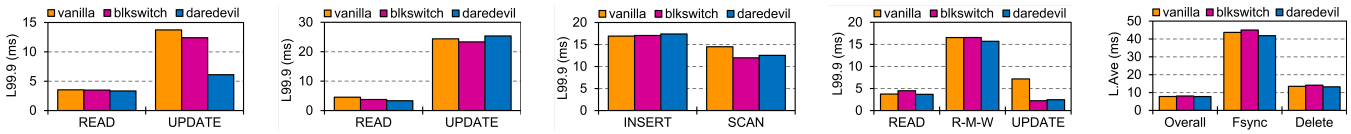
Analysis. The performance variation of DAREDEVIL depends on whether operations directly use the storage stack. RocksDB read/scan operations mostly target its internal caches, and cache-related operations account for ~77% of Mailserver workloads. These operations mainly involve CPUs accessing caches. Other operations that affect in-SSD data (e.g., RocksDB updates/inserts and fsync of Mailserver), in contrast, directly use the storage stack. Therefore, since DAREDEVIL focuses on the storage stack optimization, it only improves the performance of the latter operations, such as YCSB-A and -F with 50% updates, but exhibits little gain on CPU-intensive cases (e.g., YCSB-B and -E with 95% CPU-centric operations and Mailserver).

7.5 Overhead Analysis of DAREDEVIL

The performance overheads of DAREDEVIL arise from cross-core accesses to NQs and frequent updates of base priorities. In this section, we analyze its overheads in both cases.

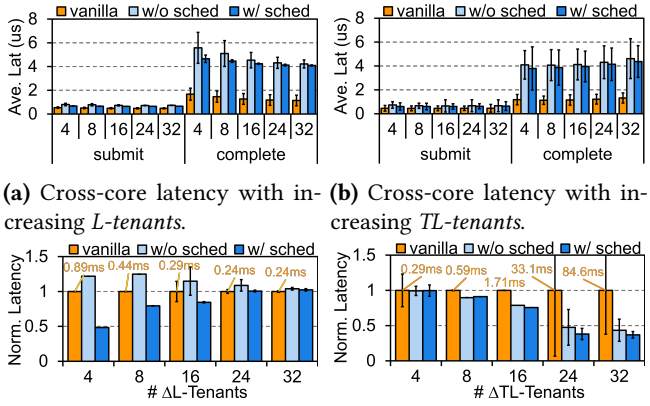
Overheads of cross-core NQ accesses. Such overheads come from NQ entry contention and request dispatching during submission and completion, respectively (§5.1). We quantify the overheads and explore their exploitability to incur unfairness. To this end, we set T-tenants' priority to the same as L-tenants, such that they (TL-tenants) share the same NQs as L-tenants to incur higher cross-core overheads. We follow the configuration in §7.1, fix the number of TL- or L-tenants to 12, vary the number of the other, and confine them to 4 cores and 16 NQs. We interleave NQ accesses in DAREDEVIL by repeatedly moving tenants across cores randomly, ensuring each NQ is accessed by multiple cores.

Figure 13 presents L-tenants' average latency under a fixed number and varying numbers of TL-tenants. As shown



(a) YCSB-A **(b)** YCSB-B **(c)** YCSB-E **(d)** YCSB-F **(e)** Mailserver

Figure 12. Performance results on real-world workloads. Figures (a) to (d) present the 99.9^{th} tail latency from YCSB workloads using RocksDB. Figure (e) shows the average latency reported in Mailserver.



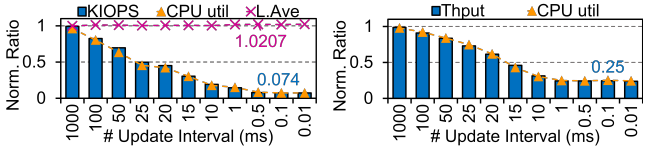
(a) Cross-core latency with increasing *L-tenants*. **(b)** Cross-core latency with increasing *TL-tenants*.

(c) Normalized total latency with increasing *L-tenants*. **(d)** Normalized total latency with increasing *TL-tenants*.

Figure 13. L-tenant average latency under fixed (a and c) and varying (b and d) TL-tenants. The black line segments represent the standard deviation of latency.

in Figure 13b and Figure 13a, while DAREDEVIL's scheduling reduces overheads and stabilizes performance, it inevitably incurs 1.4-1.6× and 3.3-3.6× higher latency during submission and completion. However, the cross-core overheads only account for at most 1.7% of the overall latency shown in Figure 13c and Figure 13d. This is because DAREDEVIL benefits from NQ utilization and monitoring cross-core overheads to efficiently distribute tenants among NQs (§5.3): despite TL- and L-tenants sharing NQs, DAREDEVIL's scheduling manages to schedule L-tenants to less contended NQs by monitoring cross-core overheads. In Figure 13c, with fewer L-tenants, it avoids using the NQs occupied by TL-tenants. In Figure 13d, its benefits are small as TL-tenants occupy nearly all NQs, leaving little space for scheduling. In this case, improvements over vanilla stem from utilizing available NQs to serve tenants.

Overheads due to updates of tenants' base priorities. As mentioned in §5.2, changing tenants' base priorities invokes re-scheduling their default NSQs, which can hinder normal I/O services. To explore the severity of re-scheduling overheads in affecting tenants' performance, we measure L- and T-tenants' performance by continuously updating *ionice* values at regular intervals (i.e., 1*s* to 10μs) for 10 minutes.



(a) L-tenant performance. **(b)** T-tenant performance.

Figure 14. Normalized performance of L- and T-tenants under decreasing update intervals of base priorities.

Figure 14 shows that L-tenants' IOPS and T-tenants' throughput decrease as updates become more frequent, gaining only 7.4% and 25% of normal performance at stable states. This is because tenants receive fewer CPU resources in I/O services (CPU util) as the frequently incurred NQ scheduling consumes them. Nevertheless, such updates exert a relatively small impact on I/O latency as NQ scheduling adopts a lightweight mechanism (§6). Moreover, the server can effortlessly limit the update rate of tenants' base priorities to avoid breaking the system performance.

8 Discussion

8.1 Limitations of DAREDEVIL

Maintaining *us*-level latency. As shown in §7, the overall latency observed by L-tenants in DAREDEVIL, though significantly reduced through NQ-level separation, still falls within the scale of *ms* instead of the *us*-level expected of NVMe SSDs (e.g., 40*ms* and 5*ms* for tail and average latencies, respectively). The similar scenario is also observed in Figure 2. This is because in addition to HOL blocking within NQs, the existence of interference inside SSDs, which is caused by intricate SSD internal structures and unique I/O behaviors of flash chips, also accounts for L-requests' inflated latency. For instance, T-requests can flood the internal queues of SSDs, blocking L-requests [86], and the erase-after-write feature of flash memory can postpone the service of small reads if large chunks of writes are present [42, 48, 91]. Such internal interference inherently hinders SSDs from providing the expected *us*-level latency to L-requests. Addressing this issue demands customized modifications to SSDs' hardware, which is beyond the ability of DAREDEVIL as a kernel storage stack. In this regard, DAREDEVIL is constrained to maintain the *us*-level latency for L-requests.

Applicability to virtual machines (VMs). Currently, DAREDEVIL does not support VMs because applications running in guest VMs are invisible to the host kernel. Extending DAREDEVIL to VMs necessitates a holistic design among the guest virtio stack, the hypervisor, and the host. Specifically, the virtio stack can adopt the same decoupled structure, making each virtqueue (VQ) serve I/O requests of the same SLA. Since the requests flowing into guest VQs are transferred to host NQs for actual I/O services, as proposed by [73], the hypervisor and the host need to maintain proper mappings between VQs and NQs, so that each VQ-NQ mapping provides I/O services consistent with their SLA. We leave this as future work.

8.2 Deployment of DAREDEVIL

Security implications. As a kernel storage stack, DAREDEVIL can be seamlessly integrated into cloud servers. Since modern servers have already incorporated various security mechanisms to defend against potential attacks, such as the privilege control [47, 76, 77] and cgroup [36] of the Linux kernel and vendor-specific defenders [63, 66], DAREDEVIL is assumed to operate in trusted environments. For instance, DAREDEVIL trusts that tenants' *ionice* values align with their SLAs because servers typically launch trustworthy checker daemons to ensure that user-launched processes are virtuous [34]. Therefore, DAREDEVIL is safe when the server security mechanisms ensure that all tenants are trusted.

However, if the trusted environments are compromised, making a denial-of-service (DOS) attack possible, DAREDEVIL itself, like existing kernel storage stacks (e.g., `blk-mq`), inevitably becomes vulnerable. This vulnerability arises because defending against such attacks requires efforts from multiple aspects of the Linux kernel [13] and thus is beyond the scope of kernel storage stacks.

Extensibility beyond NVMe SSDs. As DAREDEVIL only requires the device multi-queue feature, its design principles are not confined to NVMe SSDs but are versatile to multi-queue I/O devices. For existing multi-queue I/O devices attached to the host via the PCIe bus (e.g., smartNIC [64, 79]), DAREDEVIL is applicable with proper driver modifications to align with the decoupled block layer structure. For emerging types of SSDs, such as zoned namespace (ZNS) SSDs [6, 65] and compute express link (CXL) SSDs [44, 55, 96], DAREDEVIL can be seamlessly adopted due to their retention of the multi-queue feature. Nonetheless, as these devices incorporate features targeting schemes different from DAREDEVIL, the necessity and implementation details of integrating DAREDEVIL require case-by-case analysis.

9 Related Work

Linux I/O scheduling. Existing I/O schedulers [10, 25, 35, 37, 46, 72, 95] control the submission timing of tenants' I/O requests, balancing performance and fairness. They focus on CPU resource allocation in I/O services, which differs from the target scope of DAREDEVIL. In addition, their scheduling algorithms are built upon `blk-mq`, assuming the static core-NQ mapping, and thus inherit the same limitations as `blk-mq`. Nevertheless, DAREDEVIL is not orthogonal to existing I/O schedulers, making it non-trivial to reap the full benefits from both: adapting these schedulers to DAREDEVIL requires careful consideration of the full connectivity between cores and NSQs in I/O scheduling. We leave this as future work.

Linux kernel storage stack acceleration. The Linux kernel storage stack suffers from various drawbacks beyond the inflexibility addressed in this paper. Prior works have aimed to minimize I/O delays caused by context switches [80] and holistically collaborate multiple components within the storage stack [39, 49, 56, 101]. They also adapt the stack to utilize intrinsic device features by offloading certain functionalities to the SSD firmware [43, 86, 90, 98]. These works either target different aspects from DAREDEVIL, or require specific hardware modifications infeasible in commodity SSDs.

Software-based resource partitioning. In addition to kernel mechanisms for multi-tenancy control, cloud vendors deploy centralized resource controllers in their user-space runtime. Similar to kernel I/O schedulers, these controllers aim to fairly and efficiently distribute host CPU resources among tenants [15, 31, 41, 52, 81]. Thus, as they operate within user space, their management of user applications complements DAREDEVIL's kernel-space solution.

10 Conclusion

In this paper, we introduce DAREDEVIL, a novel kernel storage stack designed for flexible and efficient multi-tenancy control of NVMe SSDs. DAREDEVIL achieves this by decoupling the static mapping between CPU cores and NQs, integrating a lightweight request routing scheme, and incorporating the NQ scheduling mechanism. Our evaluation demonstrates that with these innovations, DAREDEVIL delivers significant performance improvement over existing kernel storage stacks in multi-tenant I/O services.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Antonio Barbalace, for their insightful comments and feedback. This work is partially supported by the National Key Research and Development Program of China under Grant No. 2023YFB4502702, the NSFC under Grant No. 62332021 and 62472007, the NSFC for Young Scientists of China (No.62202400), and the RGC for Early Career Scheme (No.27210024). Any opinions, findings, or conclusions expressed in this material are those of the authors and do not necessarily reflect the views of NSFC and RGC.

References

[1] Sungyong Ahn, Kwanghyun La, and Jihong Kim. 2016. Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/ahn

[2] Michael D Atkinson, J-R Sack, Nicola Santoro, and Thomas Strothotte. 1986. Min-max heaps and generalized priority queues. *Commun. ACM* 29, 10 (1986), 996–1000.

[3] Jens Axboe. 2017. FIO Documentation. https://fio.readthedocs.io/en/latest/fio_doc.html.

[4] Jens Axboe. 2024. FIO: Flexible I/O Tester. https://github.com/axboe/fio.

[5] Microsoft Azure. 2024. NVMe Overview. https://learn.microsoft.com/en-us/azure/virtual-machines/nvme-overview

[6] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 689–703. https://www.usenix.org/conference/atc21/presentation/bjorling

[7] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '13)*. Association for Computing Machinery, New York, NY, USA, Article 22, 10 pages. https://doi.org/10.1145/2485732.2485740

[8] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 359–374. https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling

[9] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.".

[10] Neil Brown. 2020. blk-mq: Kyber multiqueue I/O scheduler. https://lwn.net/Articles/720071/

[11] Neil Brown. 2020. Two new block I/O schedulers for 4.12. https://lwn.net/Articles/720675/

[12] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. 2017. On the Performance Variation in Modern Storage Stacks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 329–344. https://www.usenix.org/conference/fast17/technical-sessions/presentation/cao

[13] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (Shanghai, China) *(APSys '11)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/2103799.2103805

[14] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 17–32. https://www.usenix.org/conference/fast21/presentation/chen-hao

[15] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3297858.3304005

[16] Yiquan Chen, Jiexiong Xu, Chengkun Wei, Yijing Wang, Xin Yuan, Yangming Zhang, Xulin Yu, Yi Chen, Zeke Wang, Shuibing He, and Wenzhi Chen. 2023. BM-Store: A Transparent and High-performance Local Storage Architecture for Bare-metal Clouds Enabling Large-scale Deployment. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1031–1044. https://doi.org/10.1109/HPCA56546.2023.10071029

[17] Wonil Choi, Jie Zhang, Shuwen Gao, Jaesoo Lee, Myoungsoo Jung, and Mahmut Kandemir. 2016. An in-depth study of next generation interface for emerging non-volatile memories. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–6.

[18] Google Cloud. 2025. About Local SSD disks. https://cloud.google.com/compute/docs/disks/local-ssd

[19] Cloudelicious. 2020. Azure VM's and their Temporary Storage. https://www.cloudelicious.net/azure-vms-and-their-temporary-storage/

[20] Linux Containers. 2025. LXC - Introduction. https://linuxcontainers.org/lxc/introduction/

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[22] Jonathan Corbet. 2004. Kernel threads made easy. *LWN.net* (2004). https://lwn.net/Articles/65178/

[23] Docker. 2025. Docker workshop. https://docs.docker.com/get-started/workshop.

[24] The Linux Kernel documentation. 2023. Multi-Queue Block IO Queueing Mechanism (blk-mq). https://www.kernel.org/doc/html/latest/block/blk-mq.html

[25] The Linux Kernel documentation. 2024. Block IO Controller. https://docs.kernel.org/admin-guide/cgroup-v1/blkio-controller.html

[26] Drazen Zoric. 2021. How much does fragmentation affect performance? https://www.quora.com/How-much-does-fragmentation-affect-performancel)

[27] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 65–78. https://www.usenix.org/conference/nsdi19/presentation/eisenman

[28] NVM Express. 2023. Specifications - NVM Express. https://nvmexpress.org/specifications/.

[29] Facebook. 2024. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb)

[30] Jinhao Fan, Ziyue Yang, Ran Shu, Peng Cheng, and Yongqiang Xiong. 2021. Towards user-defined SLA in cloud flash storage. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China) *(APSys '21)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3476886.3477509

[31] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[32] Everette S Gardner Jr. 1985. Exponential smoothing: The state of the art. *Journal of forecasting* 4, 1 (1985), 1–28.

[33] Github. [n. d.]. blk-switch: Rearchitecting Linux Storage Stack for μs Latency and High Throughput. https://github.com/resource-disaggregation/blk-switch

[34] Masud Hasan and Cody Irwin. 2020. Anomaly detection using streaming analytics & AI. https://cloud.google.com/blog/products/data-analytics/anomaly-detection-using-streaming-analytics-and-ai

[35] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 301–314. http://www.usenix.org/conference/atc19/presentation/hedayati-queue

[36] Tejun Heo. 2015. Control Group v2 — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.

[37] Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, and Dimitrios Skarlatos. 2022. IOCost: Block IO Control for Containers in Datacenters *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 595–608. https://doi.org/10.1145/3503222.3507727

[38] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 375–390. https://www.usenix.org/conference/fast17/technical-sessions/presentation/huang

[39] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 113–128. https://www.usenix.org/conference/osdi21/presentation/hwang

[40] Taeho Hwang, Myungsik Kim, Seongjin Lee, and Youjip Won. 2018. On the I/O characteristics of the mobile web browsers. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Pau, France) *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 964–966. https://doi.org/10.1145/3167132.3167402

[41] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 519–532. https://www.usenix.org/conference/atc18/presentation/iorgulescu

[42] Lokesh N. Jaliminche, Chandranil Nil Chakraborttii, Changho Choi, and Heiner Litz. 2023. Enabling Multi-Tenancy on SSDs with Accurate IO Interference Modeling. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 216–232. https://doi.org/10.1145/3620678.3624657

[43] Kanchan Joshi, Kaushal Yadav, and Praval Choudhary. 2017. Enabling NVMe WRR support in Linux Block Layer. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotstorage17/program/presentation/joshi

[44] Myoungsoo Jung. 2022. Hello Bytes, Bye Blocks: PCIe Storage Meets Compute Express Link for Memory Expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems* (Virtual Event) *(HotStorage '22)*. Association for Computing Machinery, New York, NY, USA, 45–51. https://doi.org/10.1145/3538643.3539745

[45] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The Multi-streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. USENIX Association, Philadelphia, PA. https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kang

[46] The Linux Kernel. 2023. BFQ (Budget Fair Queueing) — The Linux Kernel documentation. https://www.kernel.org/doc/html/v6.6-rc4/block/bfq-iosched.html

[47] Michael Kerrisk. 2024. capabilities(7) - Linux manual page. https://www.man7.org/linux/man-pages/man7/capabilities.7.html.

[48] Jieun Kim, Dohyun Kim, and Youjip Won. 2022. Fair I/O scheduler for alleviating read/write interference by forced unit access in flash memory. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems* (Virtual Event) *(HotStorage '22)*. Association for Computing Machinery, New York, NY, USA, 86–92. https://doi.org/10.1145/3538643.3539753

[49] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. 2017. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 345–358. https://www.usenix.org/conference/fast17/technical-sessions/presentation/kim-sangwook

[50] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2019. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 295–308. https://www.usenix.org/conference/fast19/presentation/kim-taejin

[51] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 29, 15 pages. https://doi.org/10.1145/2901318.2901337

[52] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 345–359. https://doi.org/10.1145/3037697.3037732

[53] Akshay Kumar, Ravi Tandon, and T Charles Clancy. 2015. On the latency and energy efficiency of distributed storage systems. *IEEE Transactions on Cloud Computing* 5, 2 (2015), 221–233.

[54] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. 2020. DC-Store: Eliminating Noisy Neighbor Containers Using Deterministic {I/O} Performance and Resource Isolation. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 183–191.

[55] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. 2023. Cache in Hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSD. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage '23)*. Association for Computing Machinery, New York, NY, USA, 24–30. https://doi.org/10.1145/3599691.3603406

[56] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2019. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 603–616. https://www.usenix.org/conference/atc19/presentation/lee-gyusun

[57] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 4, 14 pages. https://doi.org/10.1145/2592798.2592821

[58] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. 2020. An In-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 37–47. https://doi.org/10.1109/IISWC50251.2020.00013

[59] LWN. 2016. blk-mq: Introduce combined hardware queues. https://lwn.net/Articles/700932/

[60] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 248–259. https://doi.org/10.1145/2155620.2155650

[61] Drew McDaniel. 2014. Virtual Machines Best Practices: Single VMs, Temporary Storage and Uploaded Disks. https://azure.microsoft.com/en-us/blog/virtual-machines-best-practices-single-vms-temporary-storage-and-uploaded-disks/

[62] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole. 2020. RCU Usage In the Linux Kernel: Eighteen Years Later. *SIGOPS Oper. Syst. Rev.* 54, 1 (Aug. 2020), 47–63. https://doi.org/10.1145/3421473.3421481

[63] Microsoft. 2024. Introduction to Microsoft Defender for Cloud. https://learn.microsoft.com/en-us/azure/defender-for-cloud/defender-for-cloud-introduction

[64] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 106–122. https://doi.org/10.1145/3452296.3472940

[65] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2023. eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 461–477. https://www.usenix.org/conference/osdi23/presentation/min

[66] Rupanjana Mukherjee and Jon Sabberton. 2024. Protecting Multi-Cloud Resources in the Era of Modern Cloud-Based Cyberattacks. https://cloud.google.com/blog/topics/threat-intelligence/protecting-multi-cloud-resources-modern-cyberattacks Accessed: 2025-01-22.

[67] NetApp. 2023. Storage namespaces endpoint overview. https://docs.netapp.com/us-en/ontap-restapi-9121/ontap/storage_namespaces_endpoint_overview.html

[68] NVM Express. 2013. NVM Express Explained. https://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf)

[69] NVM Express. 2024. NVMe Namespaces. https://nvmexpress.org/resource/nvme-namespaces/)

[70] Serial ATA International Organization. 2025. Home | SATA-IO. https://sata-io.org/

[71] Jonggyu Park and Young Ik Eom. 2021. FragPicker: A New Defragmentation Tool for Modern Storage Devices. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 280–294. https://doi.org/10.1145/3477132.3483593

[72] Stan Park and Kai Shen. 2012. FIOS: A Fair, Efficient Flash I/O Scheduler. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast12/fios-fair-efficient-flash-io-scheduler

[73] Bo Peng, Cheng Guo, Jianguo Yao, and Haibing Guan. 2023. LPNS: Scalable and Latency-Predictable Local Storage Virtualization for Unpredictable NVMe SSDs in Clouds. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 785–800. https://www.usenix.org/conference/atc23/presentation/peng

[74] Bo Peng, Ming Yang, Jianguo Yao, and Haibing Guan. 2020. A throughput-oriented nvme storage virtualization with workload-aware management. *IEEE Trans. Comput.* 70, 12 (2020), 2112–2124.

[75] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. 2022. BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 17–33. https://www.usenix.org/conference/osdi22/presentation/reidys

[76] Mickaël Salaün. 2024. Landlock: unprivileged access control. https://docs.kernel.org/userspace-api/landlock.html.

[77] Casey Schaufler. 2023. Linux Security Modules — The Linux Kernel documentation. https://docs.kernel.org/userspace-api/lsm.html.

[78] Amazon Web Services. 2025. Instance store temporary block storage for EC2 instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html

[79] Yizhou Shan, Will Lin, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. 2021. SuperNIC: A Hardware-Based, Programmable, and Multi-Tenant SmartNIC. *arXiv preprint arXiv:2109.07744* (2021).

[80] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. 2014. OS I/O Path Optimizations for Flash Solid-state Drives. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 483–488. https://www.usenix.org/conference/atc14/technical-sessions/presentation/shin

[81] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 349–362. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/shue

[82] Ross Stenfort, Lee Prewitt, Paul Kaler, David Black, Austin Bolen, Chris Sabol, and Charles Kunzman. 2023. Datacenter NVMe SSD Specification v2.5. https://www.opencompute.org/documents/datacenter-nvme-ssd-specification-v2-5-pdf Open Compute Project.

[83] Storage Performance Development Kit. 2024. SPDK: Submitting I/O to an NVMe Device. https://spdk.io/doc/nvme_spec.html)

[84] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. 2021. Optimizing Storage Performance with Calibrated Interrupts. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 129–145. https://www.usenix.org/conference/osdi21/presentation/tai

[85] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. 2018. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 49–66. https://www.usenix.org/conference/fast18/presentation/tavakkol

[86] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. 2018. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 397–410.

[87] The Linux Kernel Documentation Team. 2025. What is RCU? – "Read, Copy, Update". *The Linux Kernel Archives* (2025). https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html

[88] Vasily Tarasov. 2020. Filebench - A Model Based File System Workload Generator. https://github.com/filebench/filebench)

[89] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 364–381. https://doi.org/10.1145/3600006.3613145

[90] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. 2021. D2FQ: Device-Direct Fair Queueing for NVMe SSDs. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association,

403–415. https://www.usenix.org/conference/fast21/presentation/woo

[91] Guanying Wu and Xubin He. 2012. Reducing SSD read latency via NAND flash program and erase suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'12)*. USENIX Association, USA, 10.

[92] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotstorage19/presentation/wu-kan

[93] Jiexiong Xu, Yiquan Chen, Yijing Wang, Wenhui Shi, Guoju Fang, Yi Chen, Huasheng Liao, Yang Wang, Hai Lin, Zhen Jin, Qiang Liu, and Wenzhi Chen. 2024. LightPool: A NVMe-oF-based High-performance and Lightweight Storage Pool Architecture for Cloud-Native Distributed Database. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 983–995. https://doi.org/10.1109/HPCA57654.2024.00079

[94] Shuai Xue, Shang Zhao, Quan Chen, Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou, Keqiang Niu, Sijie Sun, and Minyi Guo. 2020. Spool: Reliable Virtualized NVMe Storage Pool in Public Cloud Infrastructure. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 97–110. https://www.usenix.org/conference/atc20/presentation/xue

[95] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. Split-Level I/O Scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 474–489. https://doi.org/10.1145/2815400.2815421

[96] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. 2023. Overcoming the Memory Wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 601–617. https://www.usenix.org/conference/atc23/presentation/yang-shao-peng

[97] Hwanjin Yong, Kisik Jeong, Joonwon Lee, and Jin-Soo Kim. 2018. vStream: Virtual Stream Management for Multi-streamed SSDs. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/yong

[98] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. 2018. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 477–492. https://www.usenix.org/conference/osdi18/presentation/zhang

[99] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. 2020. Scalable Parallel Flash Firmware for Many-core Architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 121–136. https://www.usenix.org/conference/fast20/presentation/zhang-jie

[100] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 755–770. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-yunqi

[101] Jinbin Zhu, Liang Wang, Limin Xiao, Lei Liu, and Guangjun Qin. 2023. EBIO: An Efficient Block I/O Stack for NVMe SSDs With Mixed Workloads. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 5048–5060. https://doi.org/10.1109/TCAD.2023.3296369

A Artifact Appendix

A.1 Abstract

DAREDEVIL is a Linux kernel storage stack for NVMe SSDs. It achieves flexibility and efficiency in mitigating the multi-tenancy issue by decoupling the statically structured `blk-mq` block layer. We have implemented DAREDEVIL atop the 6.1.53 Linux kernel, with modifications to the block layer and the NVMe driver. In this section, we provide description of our implementation for DAREDEVIL, along with the experimental setup to produce the results.

A.2 Description & Requirements

A.2.1 How to access. We have open sourced DAREDEVIL at https://github.com/HKU-System-Security-Lab/Daredevil, which is the preferred way, and https://zenodo.org/records/14928153 (DOI link: https://doi.org/10.5281/zenodo.14928153).

A.2.2 Hardware dependencies. DAREDEVIL only requires a functional machine with local NVMe SSDs attached via the PCIe bus. While SSDs with more available NVMe I/O queues are preferred, there are no constraints on vendors, types, and other hardware features.

A.2.3 Software dependencies. We have compiled a DAREDEVIL-enabled Linux kernel using gcc 9.4.0 and tested it on Ubuntu 22.04 LTS. However, DAREDEVIL does not require extra software dependencies other than those needed to compile a Linux kernel.

A.2.4 Benchmarks. We use Flexible I/O tester (FIO) of version 3.38.4, RocksDB, and Yahoo! Cloud Serving Benchmark (YCSB) of version 0.17.0 in our evaluation.

A.3 Set-up

We have provided a detailed setup description under the README.md file of DAREDEVIL's github repository (https://github.com/HKU-System-Security-Lab/Daredevil/blob/master/README.md). Please refer to it for setting up and testing the functionality of DAREDEVIL.

A.4 Evaluation workflow

The scripts to conduct the experiments described in this paper are provided under "Daredevil/eval". It contains a README file "Daredevil/eval/README.md" that cross-references the evaluation scripts with the figures shown in the paper.