RESEARCH-ARTICLE

Hitchhike: Efficient Request Submission via Deferred Enforcement of Address Contiguity

XUDA ZHENG, Huazhong University of Science and Technology, Wuhan, Hubei, China

JIAN ZHOU, Huazhong University of Science and Technology, Wuhan, Hubei, China

SHUHAN BAI, Huazhong University of Science and Technology, Wuhan, Hubei, China

RUNJIN WU, Huazhong University of Science and Technology, Wuhan, Hubei, China

XIANLIN TANG, Huazhong University of Science and Technology, Wuhan, Hubei, China

ZHIYUAN LI, Huazhong University of Science and Technology, Wuhan, Hubei, China

View all

Published: 22 March 2026

Citation in BibTeX format

ASPLOS '26: 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems
March 22 - 26, 2026
PA, Pittsburgh, USA

Conference Sponsors:
SIGOPS
SIGPLAN
SIGARCH

https://doi.org/10.1145/3779212.3790173
ISBN: 9798400723599

Hitchhike: Efficient Request Submission via Deferred Enforcement of Address Contiguity

Xuda Zheng
Huazhong University of Science and Technology
Wuhan, China
zhengxd@hust.edu.cn

Jian Zhou*
Huazhong University of Science and Technology
Wuhan, China
jianzhou@hust.edu.cn

Shuhan Bai
Huazhong University of Science and Technology
Wuhan, China
shuhanbai0329@hust.edu.cn

Runjin Wu
Huazhong University of Science and Technology
Wuhan, China
rainwu@hust.edu.cn

Xianlin Tang
Huazhong University of Science and Technology
Wuhan, China
tangxl@hust.edu.cn

Zhiyuan Li
Huazhong University of Science and Technology
Wuhan, China
lizy22@hust.edu.cn

Hong Jiang
University of Texas at Arlington
Arlington, Texas, USA
hong.jiang@uta.edu

Fei Wu
Huazhong University of Science and Technology
Wuhan, China
wufei@hust.edu.cn

Abstract

Modern storage systems operate under high concurrency, making large volumes of outstanding I/Os the norm. However, current I/O submission logic requires requests assigned to the same CPU core to be processed in a serialized manner, turning the software stack into a bottleneck due to high per-request overhead.

We argue that the primary cause of this inefficiency lies in *the end-to-end enforcement of the strict address contiguity validation*—the constraint that each I/O request must access a contiguous range of addresses (e.g., file offsets, sectors, and logical block addresses). Through system-level analysis, we find that while the address contiguity requirement is crucial at the device level as defined by the NVMe protocol, it is unnecessarily enforced throughout the entire I/O stack. Based on this insight, we propose *Hitchhike*, an efficient request submission logic that defers address contiguity validation to the device driver. Unlike solutions that resort to kernel-bypass or aggressive polling for performance, Hitchhike achieves high efficiency within the standard OS stack by allowing a single request to encapsulate multiple non-contiguous address ranges. This mechanism drastically reduces the number of individual requests traversing the stack, thereby amortizing the kernel's per-request overhead.

We apply Hitchhike to a graph engine, a B-tree store, and FIO. Experimental results show that Hitchhike effectively improves throughput and reduces CPU overhead, while preserving compatibility with existing kernel semantics and hardware interfaces.

CCS Concepts: • **Hardware → External storage**; • **Information systems → Storage management**.

Keywords: NVMe SSD; Parallelism; Asynchronous I/O stack

ACM Reference Format:
Xuda Zheng, Jian Zhou, Shuhan Bai, Runjin Wu, Xianlin Tang, Zhiyuan Li, Hong Jiang, and Fei Wu. 2026. Hitchhike: Efficient Request Submission via Deferred Enforcement of Address Contiguity. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779212.3790173

1 Introduction

Modern storage systems have evolved to support an unprecedented degree of parallelism across the entire I/O stack. Concurrency now permeates every layer: multi-core CPUs enable applications to generate massive numbers of concurrent requests [29, 31, 32]; software frameworks such as asynchronous programming models [43, 46] and batched submission APIs [1, 10, 12] are designed to fully exploit this concurrency; and NVMe SSDs offer tens to hundreds of hardware queues, each capable of handling thousands of in-flight operations [7, 14, 53, 54]. Consequently, maintaining a large

*Corresponding author.

ASPLOS '26, Pittsburgh, PA, USA.

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790173

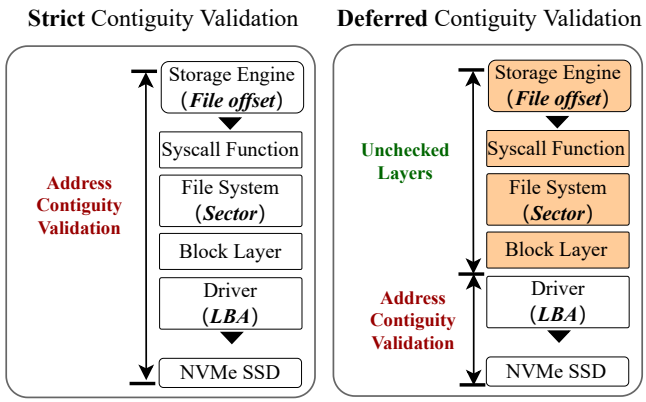


Figure 1. Strict vs. Deferred Address Contiguity Validation.

volume of **outstanding I/Os**—defined as the number of in-flight requests simultaneously active in the system—is no longer an exception, but the operational norm [18, 27]. This shift reflects a broader architectural trend: I/O throughput is increasingly achieved through parallelism, making the efficient handling of outstanding I/Os a critical design concern.

While high degrees of outstanding I/Os are essential for saturating modern storage, they expose severe inefficiencies in the Linux request submission path [12, 37, 55]. Although a single CPU core now manages thousands of in-flight requests, the kernel often fails to process them efficiently in batches. Specifically, while the kernel attempts to merge requests to reduce overhead, this optimization is strictly limited to contiguous physical addresses. Consequently, non-contiguous requests are forced to traverse the file system, block layer, and device driver independently [12, 27, 55]. This per-request execution incurs substantial CPU overhead and creates a software bottleneck that squanders the parallel potential of the hardware. This inefficiency is evidenced by the significant 2.9× performance gap we analyze in §2.2.

The Primary Cause. We argue that the primary cause of inefficiency in handling large volumes of outstanding I/Os lies in the strict address contiguity validation (§3). Specifically, as shown in Figure 1, Linux I/O stacks enforce a constraint where each request must access a contiguous range of addresses, spanning file offsets, physical sectors, and logical block addresses (LBAs). This design is rooted in the NVMe protocol's requirement [14] that each command operate on a continuous LBA region. This requirement leads to an observable performance gap: For instance, modern I/O stacks can efficiently handle large sequential reads (e.g., 128 KB per request) with minimal CPU involvement. In contrast, processing the same data volume via random reads (e.g., 4 KB per request) can require several times more CPU cycles (§2.2). The underlying reason is that each non-contiguous address range must be issued as a separate request, each incurring the full weight of the software submission overhead.

State-of-the-art I/O submission mechanisms attempt to reduce software overhead through different strategies. User-space frameworks like SPDK [51] bypass the kernel entirely, minimizing system call and context switch costs. Kernel-based interfaces like io_uring [10] introduce batching, shared memory, and efficient ring buffers to streamline submissions. However, both approaches fundamentally adhere to the same end-to-end address contiguity limitation. They treat each disjoint address range as an independent unit and maintain a rigid one-to-one mapping from the application layer down to the NVMe command [12, 37]. This raises a key question: *can we process large volumes of outstanding I/Os with the efficiency of sequential requests, while still preserving the NVMe protocol's requirement for per-request LBA contiguity?*

Insight. We make a key observation: the software I/O stack does not inherently require strict address contiguity throughout the entire submission path. Instead, this guarantee is only necessary at the device boundary, where the NVMe protocol requires each command to target a single, contiguous range of LBAs. As shown in Figure 1, this insight reveals a new design opportunity—by *deferring the address contiguity validation to the driver level*, we can lift a long-standing constraint on request merging. Such delayed validation enables the software stack to merge I/O requests that were previously unmergeable due to non-contiguous addresses. The result is a submission logic that treats a group of outstanding I/Os more like sequential ones—amortizing kernel operations, reducing per-request overhead, and greatly improving throughput and CPU efficiency.

Hitchhike. Based on this insight, we present Hitchhike, a novel and efficient request submission logic. Hitchhike decouples the strict address contiguity validation from the early stages of the I/O stack, allowing a single request, termed a Hitchhike I/O, to encapsulate multiple non-contiguous addresses. Functionally, Hitchhike aggregates the addresses of outstanding I/Os into a single offset vector, enabling the kernel to apply operations by iterating directly over the address vector—similar to how the *iov_iter* interface [9] handles multiple buffers in vectored I/O [2]. Upon reaching the device driver, Hitchhike rebinds the necessary metadata and reconstructs individual NVMe commands, each satisfying the required contiguous LBA constraint. Unlike state-of-the-art alternatives, Hitchhike delivers this performance without resorting to kernel-bypass mechanisms or aggressive polling, thereby maintaining protocol compliance and system versatility.

Hitchhike Systems. We integrate Hitchhike into three applications: the microbenchmark tool FIO [5], the graph processing framework Blaze [26], and the B-tree database LeanStore [29]. Hitchhike operates solely on the request queues that the storage engine prepares for submission, without interfering with other application logic. As a result, the integration requires minimal code changes—a few dozen lines in the application's core submission function.

We evaluate Hitchhike by comparing it against the standard Linux kernel and SPDK across a variety of workloads. Our results highlight two primary benefits. First, regarding efficiency, Hitchhike reduces the CPU cores required to saturate NVMe bandwidth by up to 75%. Second, this reduced overhead directly translates into significant gains in single-core throughput: we observe bandwidth improvements ranging from 1.43×–3.5× for micro-benchmarks and 1.17×–2.54× for real-world applications. We open source Hitchhike and our changes to FIO and applications at https://github.com/haslaboratory/Hitchhike-AE. The main contributions of this paper are:

- We provide both theoretical and empirical analysis demonstrating that the strict end-to-end enforcement of address contiguity validation is a key root cause behind the inefficiency of handling large numbers of outstanding I/O requests in modern software stacks.
- We observe that the enforcement of address contiguity validation is only necessary at the final stage of request submission—at the device boundary. Thus, the software stack can defer this constraint to the driver layer, relaxing unnecessary restrictions during earlier processing stages. Based on this insight, we propose Hitchhike, a novel request submission logic that handles outstanding I/Os more efficiently.
- We implement three Hitchhike systems: FIO, a graph processing framework, and a B-tree database. These represent both synthetic and real-world scenarios with outstanding I/Os. We show the benefits of Hitchhike systems via experiments.

2 Background and Motivation

2.1 Outstanding I/Os and Software Bottleneck

Modern storage systems are designed to support large numbers of outstanding I/O requests generated by applications, such as databases [3, 18, 31], graph processing [26, 49, 50], and vector search [17, 40], that issue increasingly outstanding I/Os. This trend is driven by the growing parallelism in hardware (e.g., multi-core CPUs, multi-channel SSDs, etc.) and software architectures (e.g., asynchronous I/O, coroutines, etc.). As a result, a single CPU core often manages tens or hundreds of concurrent I/O requests in flight. As shown in Figure 2, in graph analytics, processing a single vertex often involves accessing the adjacency lists of many neighbors, which are typically stored in separate data blocks. As a result, traversing the graph may trigger dozens or even hundreds of concurrent I/O requests. Similarly, in B+Tree-based storage engines, concurrent range queries or point lookups by multiple threads can issue I/Os to different leaf or internal nodes in parallel.

Software is becoming the bottleneck primarily due to three interrelated factors: ① The rapid advancement in storage bandwidth [11, 39], with PCIe 5.0 SSDs capable of over 2 million IOPS, has outpaced the software's ability to manage this throughput, especially for random I/O, which demands a higher number of CPU cores. ② CPU performance improvements have slowed [22], and the CPU overhead for processing the high IOPS from modern NVMe SSDs is escalating. ③ With the widespread adoption of high-performance SSDs and the rise of the *read-as-needed* paradigm [21, 31, 50], applications increasingly rely on issuing large numbers of small I/O requests. As a result, the kernel I/O stack, initially designed for slower devices, is now a critical limitation [47, 55], unable to keep up with the demands of high-performance storage systems.

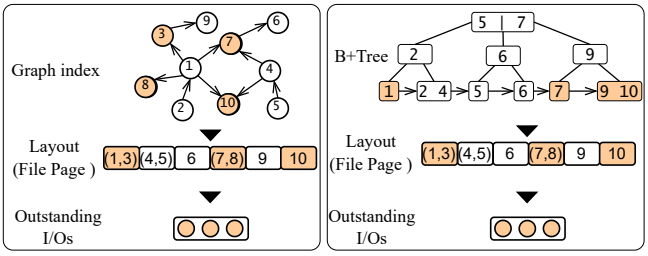
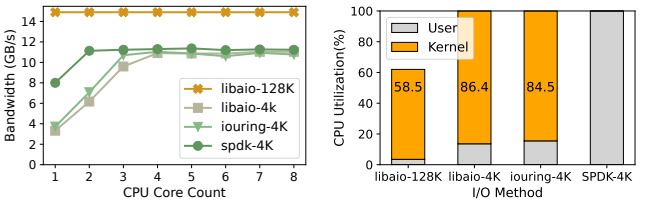


Figure 2. How outstanding I/Os arise, illustrated with graph and tree structures.

While io_uring [10] offers optimizations like shared memory and fixed buffers that reduce overheads in memory management, it cannot fully address the fundamental challenges of how the Linux kernel handles outstanding I/Os. The reason for the kernel's overhead in managing I/O operations is the necessity of traversing multiple layers of abstraction and encapsulation, which leads to extensive communication between these layers and consequently introduces latency and overhead. SPDK [51] adopts methods that bypass the kernel and allow application code to communicate directly with device drivers. It is worth noting that io_uring has integrated support for NVMe passthrough [25], which allows direct submission of requests to the kernel driver. However, bypassing kernel layers inevitably sacrifices the semantics and functionality those layers provide. For example, NVMe passthrough [25] bypasses the file system, restricting access to raw block devices only.

2.2 Software Stack Performance Analysis

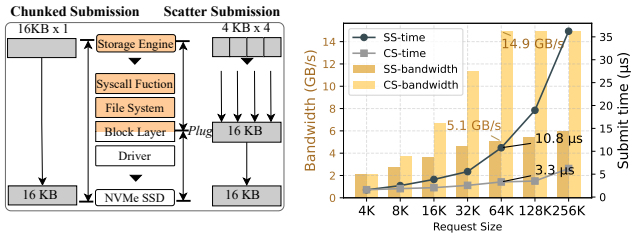
At first glance, the software I/O stack does not always appear to be the bottleneck. As shown in Figure 3, a single CPU core—utilizing only around 60%—is sufficient to saturate the throughput of a PCIe 5.0 SSD [11] under 128 KB sequential I/O. However, achieving the maximum throughput under 4 KB random I/O requires significantly more CPU resources, often occupying 2–4 cores. In this case, the kernel I/O stack alone consumes over 80% of total CPU cycles, primarily due to the overhead of managing a large number of small, independent I/O requests. This contrast reveals that the software



(a) Bandwidth with multi-threading scaling

(b) CPU utilization under single thread (kernel overhead proportion)

Figure 3. The read bandwidth and CPU utilization of PCIe 5.0 SSD[11] with FIO.



(a) Chunked submission (CS) and scatter submission(SS).

(b) Bandwidth and submission latency under the two submission methods.

Figure 4. Further analysis of the impact of software stack overhead, with the io_uring engine in FIO.

stack's efficiency is highly dependent on address contiguity: it performs well when processing large, sequential I/O, but becomes a bottleneck when facing high volumes of small, scattered requests.

To further investigate where this inefficiency arises, we compare two I/O submission patterns with io_uring in Figure 4a. In a chunked submission, a single large I/O (e.g., 16 KB) is submitted directly. In contrast, a scatter submission issues the same amount of data as separate I/Os (e.g., four 4 KB contiguous requests), which are later merged in the block layer via the plug mechanism. While both ultimately deliver the same size data to the device (e.g., 16 KB), *the key difference lies in software stack overhead above the block layer plug queue*: Chunked submission incurs this overhead once per chunk, whereas scatter submission incurs it multiple times—once per small I/O.

Experimental results confirm this intuition. Despite eventual merging at the block layer, scatter submission incurs much higher submission latency— in the context of asynchronous I/O, it captures the full software stack overhead, from user-space submission to the return of the kernel's commit routine. As shown in Figure 4b, for 64 KB I/O, scatter submission requires 3.27× the submission time of chunked submission. Consequently, its throughput is only 34.2% of that achieved by chunked submission. This shows that *under high outstanding I/O, delays in the software stack—especially above the block layer—hinder timely request submission, preventing full exploitation of SSD bandwidth.*

For sequential reads, merging I/Os into larger requests in user space is straightforward and widely practiced. In contrast, random reads are widely believed to be unmergeable: they must access data in place, and neither applications nor the kernel typically reorganize them. However, with the growing use of high-performance SSDs and the shift toward on-demand, read-intensive workloads, it is time to revisit this assumption: *why can't random read requests be merged, just like sequential ones?*

2.3 Strict Address Contiguity Validation

To identify what prevents these random outstanding I/Os from being merged, we further analyzed the metadata composition of the I/O requests. Listing 1 illustrates a typical submission interface from io_uring, where each request includes metadata fields such as *opcode, fd, offset, addr*, and *len*. This metadata guides the request's traversal through the kernel I/O stack and dictates the particular operations, such as file system inode lookups, address translation, and buffer pinning. In other words, I/O metadata is not merely descriptive—it encodes operational semantics.

Listing 1. Definition of `io_uring_prep_rw`

```
void io_uring_prep_rw( )
{
        sqe->opcode = (__u8) op;
        sqe->fd = fd;
        sqe->off = offset;
        sqe->addr = (unsigned long) addr;
        sqe->len = len;
}
```

We examine these fields one by one. The *opcode* specifies the operation type, such as read or write. We focus primarily on read requests. The *addr* and *len* fields describe the memory buffer and size for the data transfer. Importantly, modern interfaces such as *readv* [2, 9] already allow a single request to carry multiple buffers, reducing the need for separate requests based on memory layout alone. This leaves *fd* and *offset*, which together define the logical location of the data. The *fd* acts as a namespace or prefix—often representing a file or device—while *offset* specifies the position within that namespace. The file system uses both to translate the final LBA. Unlike buffers, which can already support multiple segments, each I/O request is still limited to a single, contiguous offset range—in other words, each request usually has a unique *(fd, offset)* pair.

Our analysis of the kernel I/O path identifies a pervasive design principle we term **strict address contiguity validation**. Under this model, the I/O stack enforces continuity checks end-to-end, requiring the file offset, physical sector, and final LBA to each form a continuous range. This rigid enforcement is originally driven by the NVMe protocol, which

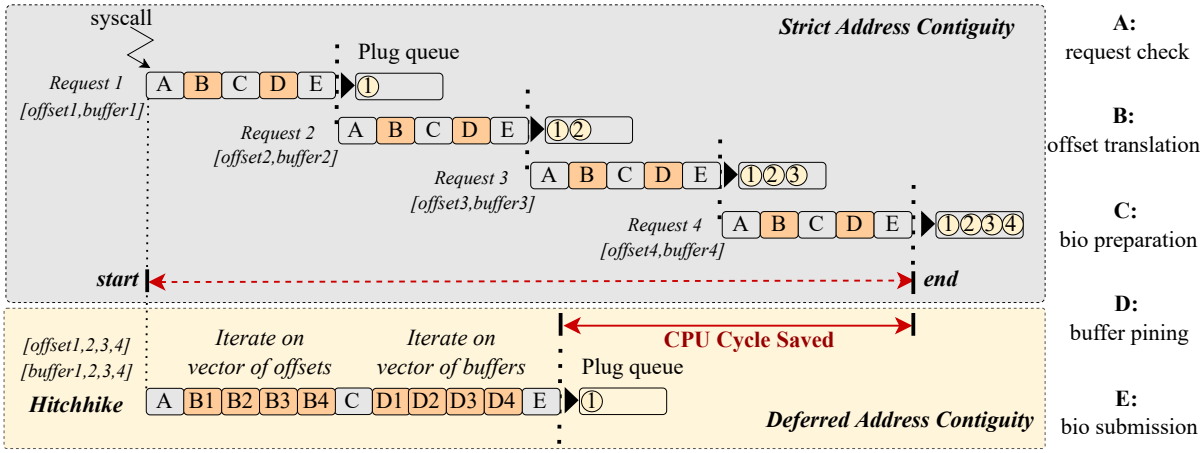


Figure 5. A comparison of the request submission paths under the strict versus deferred address contiguity models.

mandates that each hardware command describe a contiguous LBA region. However, current software stacks adopt an overly conservative strategy by extending this hardware-specific constraint to all address types throughout the entire upper software stack. Consequently, even when multiple outstanding requests target nearby data, they must be processed as separate units if their target addresses are not strictly contiguous. This limitation significantly amplifies processing overhead and restricts merge opportunities under random access workloads.

Taking graph computing as an example, when responding to a breadth-first search (BFS) query, each vertex visited will issue a batch of 4K random read requests to find subsequent vertices, all of which are collocated in a single large file [26, 33, 49]. Linux I/O submission logic treats each request as an independent unit. Each I/O request is processed independently through the entire kernel stack, repeatedly triggering identical operations. This redundant processing incurs significant CPU overhead and hampers submission efficiency. These findings call for a rethinking of I/O submission: Merging requests with different offsets can enhance performance while preserving kernel semantics.

3 Design Philosophy

3.1 Deferred Address Contiguity Validation

We observe that the requirement for strict address contiguity is only essential at the final stage—when requests are translated into NVMe commands and submitted to the hardware. Earlier stages of the I/O stack (e.g., file system, block layer) can safely operate on grouped non-contiguous addresses, as long as contiguity is restored before reaching the device. The discussion on scatter submission also demonstrates that these early kernel stages account for a vast majority of CPU overhead and are a key root cause of bottlenecks in the software stack. This insight motivates *deferred address contiguity*, which allows requests with non-contiguous addresses to be merged and processed together within the kernel, significantly reducing redundant operations without violating NVMe protocol constraints.

Figure 5 compares request submission paths under the strict and deferred address contiguity models. The path covers major stages—system call entry, file system traversal, and block layer processing (plug queue)—which we abstract into five representative steps: request check, offset translation, *bio* preparation, buffer pinning, and *bio* submission. For clarity, some minor operations (e.g., *iocb* initialization) are omitted.

In the strict model, each I/O request independently traverses all five steps, resulting in redundant execution and high overhead. In contrast, the deferred model merges multiple requests into a single Hitchhike I/O, which carries two vectors: a buffer vector (using the kernel-provided struct *iovec*) and an offset vector (maintained by our custom structure). Offset- and buffer-related steps (e.g., offset translation, buffer pinning) are iterated over the vector, while other steps (e.g., request check, *bio* preparation, *bio* submission) are executed only once for the entire group. This consolidation is Hitchhike's key advantage: by reducing redundant kernel operations, it significantly lowers CPU submission cost and accelerates request dispatch to the device.

3.2 The Hitchhike Approach

Guided by the principle of deferred address contiguity validation, we propose Hitchhike, whose architecture is depicted in Figure 6. The distinguishing feature of Hitchhike lies in its ability to encapsulate non-contiguous on-disk addresses within a single request—a capability that fundamentally differentiates it from traditional models. While Standard I/O restricts requests to contiguous buffers and addresses, and Vectored I/O (e.g., readv) relaxes only the memory buffer constraint while retaining strict address contiguity, Hitchhike removes both limitations. It allows a single submission to carry both non-contiguous memory buffers and non-contiguous

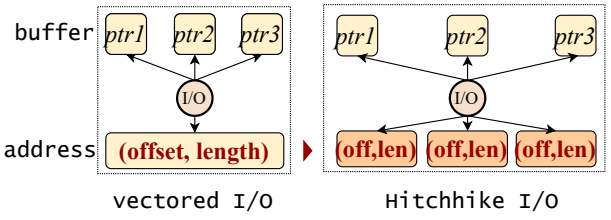


Figure 6. The Vectored vs. Hitchhike approach.

target addresses, thereby generalizing the concept of vectorization from memory to storage.

3.3 Scope of Applicability.

File-Centric Merging Scope. Hitchhike adopts a conservative design principle: grouping only requests targeting the same file descriptor (*fd*) into a single Hitchhike I/O. This decision is driven by kernel complexity; requests from different *fds* may traverse distinct I/O paths (e.g., across different mount points, namespaces, or file systems), making blind merging impractical. Despite this constraint, Hitchhike covers the vast majority of performance-critical workloads. Data-intensive applications—such as graph engines [26], B-tree stores [29], and vector search indices—typically consolidate data into monolithic files or operate directly on raw block devices. In these single-file scenarios, Hitchhike fully unleashes its capabilities. For cross-file scenarios, Hitchhike relies on grouping requests by *fd*. For instance, a batch of 64 requests spanning three distinct files is simply processed as three independent Hitchhike groups. Our experiments (§6.1.1) confirm that even smaller batch sizes yield consistent performance gains. Crucially, in the rare worst-case scenario where every request targets a unique file, Hitchhike gracefully degrades to standard I/O behavior without introducing negative side effects.

Why not SPDK? The limitations of SPDK stem not only from its semantic gap with the kernel but also from its lack of device sharing mechanisms and the prohibitive CPU consumption caused by busy polling. In contrast, Hitchhike is tailored for efficiency-critical scenarios that demand high throughput without the CPU penalty. It fully preserves kernel compatibility and shared device semantics, making it a robust alternative for environments where CPU cycles are scarce or must be balanced between application logic and I/O processing.

4 Design Overview

4.1 Overview

The Hitchhike submission path. Our analysis shows that enforcing address contiguity throughout the I/O stack is overly conservative—only the driver must ensure contiguous LBAs to meet NVMe requirements. By deferring this requirement, the kernel can process multiple requests together—even if their addresses are non-contiguous—avoiding repeated execution of identical kernel operations. As illustrated in Figure 7, leveraging this insight, we aim to design an efficient submission logic, termed Hitchhike, that enables outstanding I/Os to defer address reconstruction until the driver stage.

4.1.1 Metadata Vectorization.

The first issue is how applications submit requests based on deferred address contiguity. Hitchhike introduces a new abstraction: the Hitchhike I/O, which bundles multiple submissions of non-contiguous addresses into a single submission unit. Drawing inspiration from vectored I/O interfaces such as readv [2], which allow a single system call to operate on multiple non-contiguous buffer regions, Hitchhike applies a similar principle, but at the level of address: it vectorizes multiple requests of non-contiguous addresses, along with their associated buffers, into a single Hitchhike request.

We refer to requests submitted through the Hitchhike I/O interface as *hio*, while the merged requests are called *hitchhikers*. Theoretically, a larger merge size leads to greater efficiency and better performance. Since a Hitchhike I/O differs from conventional I/O requests in structure—carrying vectors of both offsets and buffers—it requires minimal extension to the submission interface. In §5, we detail how we extended io_uring and libaio to support Hitchhike requests with only modest changes.

4.1.2 Vectorized Metadata Handling.

In Hitchhike, the kernel handles a Hitchhike request carrying a vector of non-contiguous addresses by iterating over each address to perform necessary operations, including access verification, offset-to-sector translation, and tag allocation. Since the driver eventually maps each LBA segment to an independent NVMe command, every address must be assigned a unique tag (*command_id*) to ensure correct flow control and full protocol compliance.

4.1.3 Deferred Metadata Binding.

The NVMe protocol requires each command to access a contiguous LBA range starting at a specified *slba* [14]. To comply with this requirement, the driver splits Hitchhike requests into multiple NVMe commands—each corresponding to one LBA segment in the vector. These commands are individually written to the submission queue (SQ).

When a device interrupt is triggered upon the completion of any hitchhiker request, the kernel does not immediately perform full callbacks for each individual command. Instead, it first releases only the non-shared resources (e.g., tag) of the completed request and defers the remaining callbacks. Once all NVMe commands belonging to a single Hitchhike request have completed, the system triggers a unified callback for that Hitchhike I/O. This reduces the total number

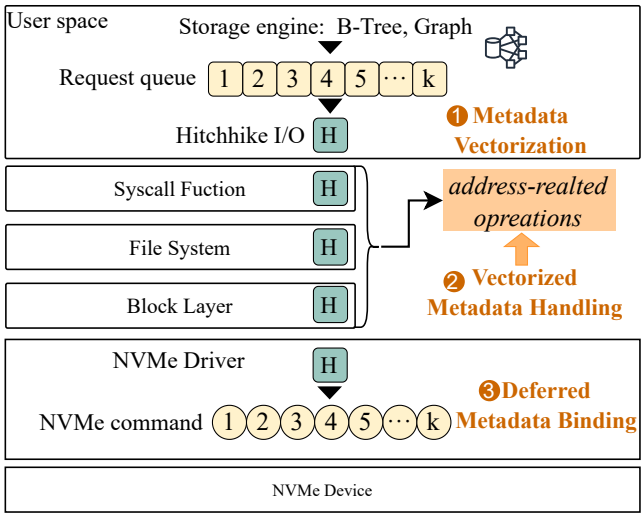


Figure 7. The overview of Hitchhike.

of kernel operations required. Although this deferred completion strategy may slightly increase the minimum latency for some requests, it significantly lowers average and tail latencies by minimizing per-request overhead—especially under high load (§6.1).

4.2 Modeling the Hitchhike Submission Logic

For operations that do not require iterating over vectorized metadata, Hitchhike effectively amortizes the cost of processing multiple requests through a single kernel operation—functionally similar to batching. Based on this observation, the performance improvement of Hitchhike can be modeled using Amdahl's Law, as shown in Equation 1. In this model, the submission time is divided into the user-space portion $(1-K)$ and kernel-space portion (K). The kernel-space portion is further split into a non-reduced portion $((1-\beta)K)$ and a reduced portion (βK), where n represents the number of requests being merged.

$$\alpha = \frac{1}{(1-K) + (1-\beta) * K + \frac{(\beta * K)}{n}} = \frac{1}{1 - (\frac{n-1}{n}) * \beta K} \quad (1)$$
$$\text{where } n >= 1;\ \beta,\ K \in (0,1)$$

Equation 1 indicates that the speedup, denoted by α, is proportional to K, β, and n. For n, a larger n implies a higher level of request merging, resulting in greater amortization of kernel overhead. However, due to diminishing returns, the benefit of increasing n saturates quickly. As for β, this parameter reflects how much of the kernel processing can be reduced. Its value depends on the nature of the workload and the implementation of the kernel stack. K denotes the fraction of submission time spent in the kernel—the portion Hitchhike aims to optimize. Thus, a larger K indicates a greater portion of total submission time spent in the kernel, and therefore a higher potential for improvement. Since user-space logic varies across applications, the value of K can differ significantly. In general, the leaner the user-space processing, the higher the K. For example, a dedicated I/O thread tends to have a much larger K than a thread that handles both computation and I/O.

To validate our model and assess the optimization potential of Hitchhike, we conduct a case study using 4KB random reads with FIO on Linux 6.5. Our profiling shows that offset-related operations account for 23.8% of kernel overhead, and buffer-related operations contribute 3.45%. Together, these non-reduced operations make up only 27.25% of the total kernel cost. This leaves 72.75% of the overhead (β=0.7275) as reduced, and thus optimizable by Hitchhike. Assuming kernel time constitutes 80% of total submission time (K=0.8), and Hitchhike merges 64 requests (n=64), Equation 1 predicts a theoretical speedup of 2.34×. In practice, Hitchhike achieves a 2.29× improvement on the same workload (§6.1.3)—closely aligning with the model and confirming its validity.

4.3 Discussion

- **Asynchronous Interface and Consistency.** Hitchhike naturally aligns with asynchronous I/O models (e.g., libaio and io_uring), which are prevalent in high-concurrency storage architectures. By leveraging these interfaces, Hitchhike allows storage engines to submit request batches efficiently. Since asynchronous I/O stacks are inherently orderless and rely on explicit synchronization primitives rather than submission order, Hitchhike's out-of-order execution introduces no correctness issues.
- **Write Requests.** Modern storage architectures, utilizing designs like LFS [38] or LSM-trees [36], typically buffer small writes in memory and flush them as large, sequential I/O batches. Since the standard software stack handles large sequential requests efficiently, the submission overhead is rarely a bottleneck for write operations. Consequently, Hitchhike primarily targets read-intensive scenarios, and our YCSB mixed workload experiments demonstrate that Hitchhike delivers performance improvements even without explicitly applying Hitchhike to write requests.
- **Direct I/O Focus**. While Hitchhike's core principles are not inherently bound to Direct I/O, our current implementation focuses on this mode to isolate the efficiency of the submission path from page cache dynamics. Extending Hitchhike to support Buffered I/O is architecturally viable but introduces complex orthogonal factors, such as cache management policies and data locality. To maintain a clear focus on our primary contribution, we reserve page cache integration for future work.

5 Implementation

Asynchronous I/O interfaces naturally support request aggregation and batching. Although Hitchhike could in principle be applied to synchronous I/O, doing so would require intrusive changes—such as explicitly grouping requests across threads—which violates our goal of preserving application logic. Therefore, we restrict our focus to applications that already use asynchronous I/O. Currently, Linux only supports two asynchronous I/O interfaces: io_uring and libaio. Although io_uring is the successor to libaio, many applications still rely on libaio. Thus, we choose to integrate Hitchhike into both libaio and io_uring. Specifically, we integrate Hitchhike into Blaze [26], a graph processing engine built on libaio; LeanStore [29], a B-Tree database using io_uring, and FIO, a widely-used microbenchmark tool. Figure 8 illustrates the specific operational details of Hitchhike under Linux asynchronous I/O stack.

5.1 Grpah Case

Scenario: Large-scale request batch submission (millions of requests). Out-of-core graph processing systems typically store data on SSDs and maintain indices in DRAM. Due to the access patterns of graph data, execution often involves numerous random 4K accesses to a single large file. Therefore, we integrated the Hitchhike interface into the graph processing system for performance validation. Blaze [26], a state-of-the-art graph processing system known for its bandwidth performance and used as a benchmark in the literature [34, 44].

Hitchhike does not interfere with the operation of the storage engine but instead intervenes when the I/O thread is ready to process and submit the request queue. As shown in Figure 8, in the libaio interface, request parameters are encoded in *iocb* structures before being submitted to the kernel. Since Linux already supports vectorized buffers via struct *iovec*, we only need to handle offsets separately. Our goal with the Hitchhike-aio interface is to allow random I/O requests, despite their inherent randomness, to be submitted to the kernel in a manner similar to sequential I/O, where fewer *iocb* are needed.

Due to the strict 64-byte limit of the *iocb* structure in libaio, it cannot store additional metadata such as per-request offsets. To address this, we introduce *struct hitchhiker*, which holds an array of offsets and the number of merged requests. This structure is passed to the kernel via a pointer and retrieved using standard memory copy operations (e.g., *copy_from_user*), similar to vectored buffer handling.

Hitchhike preserves existing libaio interfaces (*io_setup()*, *io_getevents()*, etc.) for compatibility. During initialization, extra memory is allocated alongside each *iocb* to create and configure its associated *hitchhiker* structure, including the maximum merge size. Instead of submitting one *iocb* per request, Hitchhike accumulates offsets of multiple requests into the *hitchhiker* structure. When the merge limit is reached, both the *iocb* and the hitchhiker pointer are submitted together via a new syscall, *io_submit_hit()*, enabling efficient batching of random I/O requests.

5.2 KV Store Case

Scenario: Batch submission of small-scale requests (from a dozen to several dozen). To further validate the applicability of Hitchhike in the KV domain, we integrated Hitchhike into another B-Tree storage engine, LeanStore, as WiredTiger does not support asynchronous I/O interfaces. LeanStore's support for asynchronous I/O and its high performance have made it a popular choice in various studies [4, 18–20, 29, 30, 35, 42, 57].

We replaced LeanStore's io_uring interface with the hitchhike -uring interface. LeanStore's request submission method differs significantly from Blaze; in Blaze, each group can consist of up to millions of requests, while in LeanStore, the number of requests per group is determined by the number of tasks. Moreover, in some YCSB workloads, the proportion of write requests is much higher than in Blaze. Since Hitchhike does not merge write requests, the number of requests it can merge is much smaller than Blaze. In practice, LeanStore typically groups only a few to a few dozen requests at a time.

Due to the io_uring interface using shared memory for data transfer, we introduce the *IORING_SETUP_HIT* flag to register shared memory for the *hitchhiker* structure via mmap. The size of this memory is determined by the user-defined merge size. Additionally, we have modified the liburing library to simplify the retrieval and encoding of *hitchhikers* with (*io_uring_get_hite()*) interface. To mark a request as a hitchhike request, we add the *IOSQE_HIT* flag to io_uring.

5.3 Kernel Operations Handling

As shown in Figure 8, to avoid affecting the execution of normal requests, we add a Hitchhike flag (*hitchhike_enable*) to all relevant structures (*iocb*, *sqe*, *dio*, *bio*, *request*) to indicate whether a request is a Hitchhike request. Address-related operations are vectorized only when this flag is set. In the driver, the results of the vectorized execution—DMA addresses (physical addresses obtained via DMA mapping of buffers), LBAs (device addresses derived from offset translation), and tags (per-request identifiers in the hardware queue)——are matched in sequence. This generates a separate NVMe command for each Hitchhiker, ensuring that the NVMe protocol's LBA contiguity requirement is satisfied at this stage.

5.4 FIO

As discussed in the context of Amdahl's Law, Hitchhike's optimization potential is closely tied to the proportion of kernel overhead, which varies depending on the level of user-space optimization in different applications. Therefore, we integrate two Hitchhike-based asynchronous I/O interfaces into

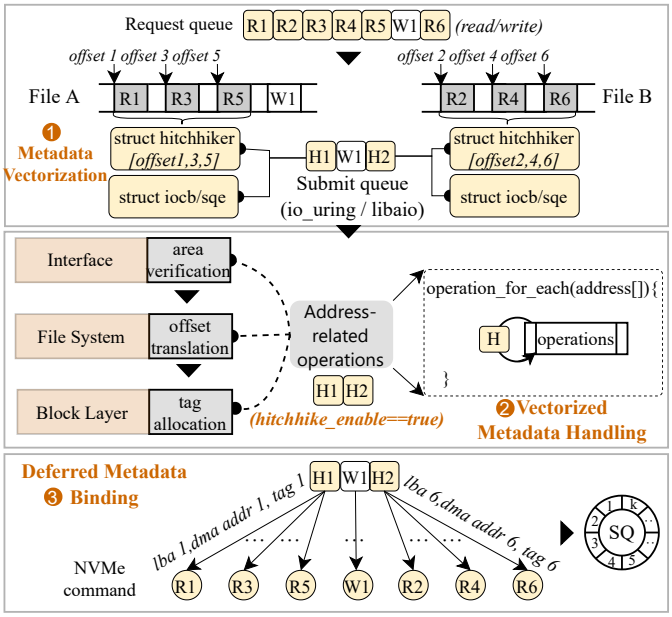


Figure 8. The implementation of Hitchhike.

FIO, using microbenchmarks to demonstrate Hitchhike's theoretical maximum performance. This setup also allows us to analyze the impact of various parameters on Hitchhike, such as merge degree, queue depth, thread count, and whether a file system is used.

6 Evaluation

Methodology. We conducted our experiments on a server machine with two Intel (R) Xeon (R) Gold 6430 CPUs, using a Linux kernel version 6.5. We evaluated both PCIe 4.0 and 5.0 NVMe storage devices, specifically testing three SSDs: Samsung PM1743 (PCIe 5.0), PM9A3 (PCIe 4.0), and Dapustor H5300 (PCIe 5.0). Due to the superior performance of the H5300, unless otherwise specified, we will conduct tests on the H5300 by default. To avoid interference during testing, hyper-threading was disabled, and the number of threads was kept within the number of CPU cores to ensure each thread had exclusive access to a core. The NVMe devices were configured to use the default *none* scheduler.

Baseline. We integrated Hitchhike into both libaio (*hitchhike-aio*) and io_uring (*hitchhike-uring*). We exclude XRP [55] and BypassD [47] as they target orthogonal low-queue-depth (QD=1) scenarios. SPDK [51] remains the state-of-the-art baseline for the high-concurrency workloads addressed by Hitchhike. We compare Hitchhike against the following baseline approaches :

- **libaio** [1]: Linux's native asynchronous I/O interface.
- **io_uring** [6]: A more recent asynchronous I/O interface that eliminates context switches by utilizing ring buffers for communication between the user space and the kernel. For maximum performance, we enable fixed buffers (i.e., *fb*), submission queue polling (SQPoll), and completion queue polling (IOPoll).
- **io_uring_cmd** [25]: A new NVMe passthrough I/O path based on io_uring. We enable fixed buffers (*fb*) and completion queue polling (IOPoll).
- **SPDK** [51]: A toolkit designed for developing low-latency, high-throughput user-mode storage I/O stacks, primarily used for block device testing because it does not support kernel file systems.

Workloads and parameters. Our evaluation included synthetic microbenchmarks and real-world workloads. Using FIO-3.37 [5], we conducted microbenchmarks to evaluate Hitchhike's throughput, latency, and CPU utilization by performing 4KB random read operations with *O_DIRECT*, targeting the same file or device in all tests. For real-world scenarios, we concentrated on an out-of-core graph processing application (based on libaio) [26] and a B-tree KV store application (based on io_uring)[18]. libaio, io_uring, and io_uring_cmd default to a batch size of 32, as this size achieves the maximum bandwidth (Figure 9a).

6.1 Microbenchmarks

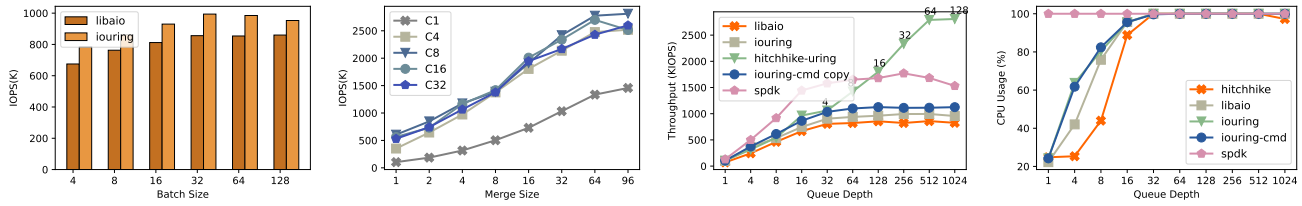
6.1.1 Key Parameters.

We first examine two key parameters in Hitchhike's I/O stack: queue depth, the maximum concurrency of individual I/O operations at the device level, and merge size, the number of requests merged into one Hitchhike request (fixed at 1 for the baselines).

Merge size. Figure 9b shows that Hitchhike's performance peaks at a merge size of 64. The highest throughput is also achieved when the Hitchhike request concurrency reaches at least 4, with only marginal gains beyond this level. Therefore, in subsequent experiments, whenever feasible, we configure the merge size to at least 64 and the concurrency to at least 4.

Queue depth. We compared Hitchhike with three I/O engines—libaio, io_uring, and io_uring_cmd—at different queue depths. As shown in Figure 9c, Hitchhike does not outperform libaio/io_uring at depths below 8, as its small merge size (under 4) limits its merging capability. However, once the queue depth exceeds 32, Hitchhike's throughput increases significantly, especially as the merge size grows from 4 to 64 (and beyond). While io_uring and libaio throughput plateaus, Hitchhike continues to scale, ultimately reaching 2.8 M IOPS. This shows that Hitchhike benefits from deeper queue depths supported by modern Linux systems.

CPU utilization. As shown in Figure 9d, while all four approaches reach full CPU utilization at a queue depth of 32, Figure 9c reminds us that Hitchhike offers a much higher throughput performance. That is, once CPU utilization reaches its maximum, libaio and io_uring struggle to process more I/O requests effectively, even with deeper queues, leading



(a) Batch size for io_uring and libaio

(b) Impact of merge size and concurrency level (C1,...C32)

(c) Performance at different queue depths (labels are merge size)

(d) CPU utilization with different queue depth

Figure 9. The performance impact of batch size, merge size, and queue depth.

their throughput to level off at the queue depth of 32. In contrast, at full CPU utilization, Hitchhike is able to significantly boost its throughput from 1.2 M to 2.8 M IOPS by increasing the merge size from 4 to 64. This highlights Hitchhike's key advantage: reducing CPU overhead through merging.

SPDK. For fairness, SPDK was configured with a queue depth of 256 to ensure it reached peak performance. As shown in Figure 9c, SPDK outperforms Hitchhike at low queue depths due to its polling design; however, at high queue depths, Hitchhike overtakes SPDK. This result may seem counter-intuitive, as SPDK achieves extremely low submission latency, which is roughly 10% of standard io_uring. However, submission latency alone does not dictate maximum throughput; rather, it is the total CPU overhead per I/O—encompassing memory management, reactor loop cycles, and busy polling costs—that determines the saturation point. Section 4.2 presents an analytical model grounded in the io_uring framework, employing Amdahl's Law to quantify Hitchhike's theoretical gains.

These results demonstrate that Hitchhike is well-suited for modern Linux systems that support greater queue depths, where it can take full advantage of the higher concurrency available at the device level. By merging outstanding requests, Hitchhike effectively reduces the per-request submission overhead, leading to significantly lower CPU utilization. This ability to trade queue depth for efficiency can make Hitchhike beneficial in workloads that generate large numbers of concurrent I/Os.

6.1.2 Raw Disk I/O Path.

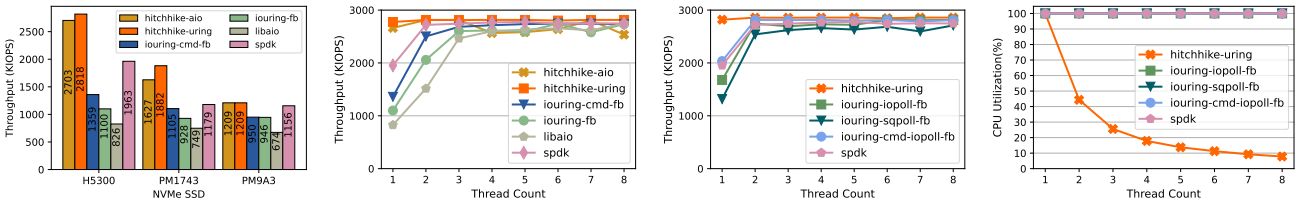
Based on the parameters from the aforementioned tests, we further evaluated Hitchhike's throughput, latency, and CPU utilization performance on raw disk (without file system).

Throughput. Hitchhike demonstrates a significant performance advantage over the baselines in both single-threaded and multi-threaded scenarios, as shown in Figure 10a and 10b. In single-thread tests, hitchhike-uring reaches 2.8 M IOPS, which is 3.5× that of libaio (0.8 M IOPS), 2.6× that of io_uring-fb (1.1 M IOPS), and 2.1× that of io_uring_cmd-fb (1.3 M IOPS). Notably, even compared to SPDK, Hitchhike achieves 43% higher throughput, demonstrating its ability to outperform both traditional kernel-based interfaces and user-space I/O frameworks. In multi-threaded tests, Hitchhike requires 1/4 to 1/3 fewer threads than traditional interfaces to reach near-maximum SSD bandwidth, achieving 2.7 M IOPS with just 1 thread for both hitchhike-aio and hitchhike-uring. Moreover, hitchhike-uring maintains 2.8 M IOPS across all thread counts from 1 to 8, reducing CPU overhead and offering better scalability in multi-threaded workloads.

Hitchhike vs. Poll. We tested polling-based baselines, as shown in Figure 10c and 10d, including io_uring with iopoll, sqpoll, SPDK, and io_uring_cmd with iopoll. Enabling iopoll boosted io_uring's performance from 1.1 M IOPS to 1.6 M IOPS and io_uring_cmd's from 1.3 M IOPS to 2.0 M IOPS, matching SPDK. SQpoll provided a smaller boost, increasing io_uring's IOPS to 1.3 M. Notably, Hitchhike achieves 2.7 M IOPS on a single thread without relying on polling. Even though polling can provide performance gains, its drawbacks are significant. In multi-threaded tests, the per-thread request rate decreases as the device approaches its peak throughput. However, because each application thread occupies a dedicated core, all polling-based baselines consumed 100% CPU on the application core. Note that the CPU usage in Figure 10d measures application threads. However, io_uring's SQpoll mode requires an extra kernel polling thread, which effectively occupies another dedicated core not shown in this metric. In contrast, Hitchhike's per-thread CPU overhead decreases as the number of threads increases.

Queue depth-latency. As shown in Figure 11a, Hitchhike consistently achieves the lowest 99th percentile latency. The reason is twofold: (1) Thanks to request merging, Hitchhike achieves a lower effective software queue depth, enabling faster submission of requests to the device. (2) Figure 11b shows that Hitchhike maintains lower average and maximum latencies, offering more consistent performance despite not having the lowest minimum latency.

Throughput-latency. We further analyzed the relationship between IOPS and latency at different queue depths. Figure 11c shows that Hitchhike excels, especially at higher IOPS, achieving 2.8 M IOPS with 160 µs latency—nearly 2.1×



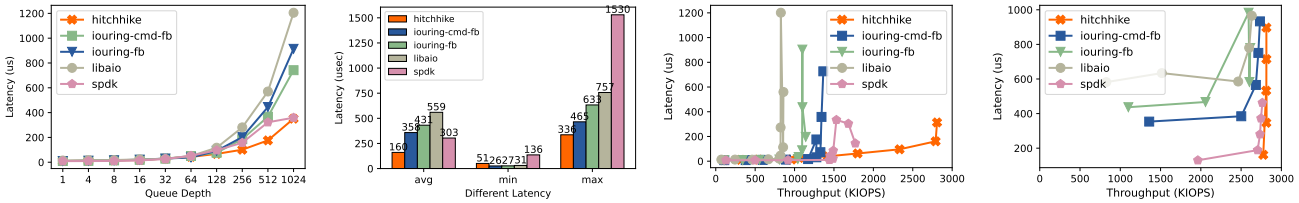
(a) Throughput in different NVMe SSDs with single thread

(b) Throughput under multiple threads

(c) Throughput with different polling methods

(d) CPU utilization with polling

Figure 10. The throughput performance and CPU overhead. In multi-threaded tests, we assign a dedicated core to each thread.



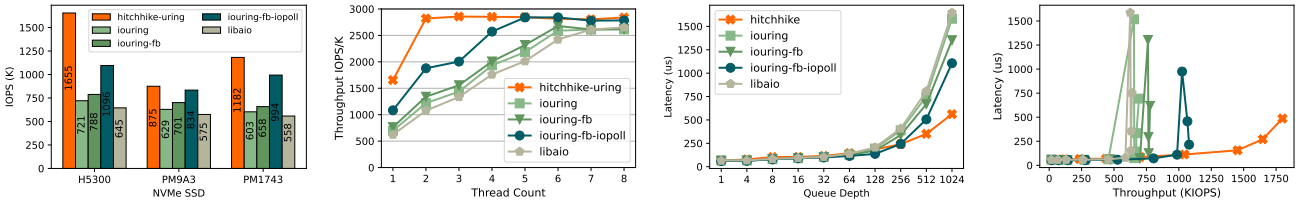
(a) The 99th latency under different queue depth

(b) min-max-avg latency at a queue depth of 512.

(c) Average latency at different IOPS.

(d) Average latency at different IOPS under multiple threads.

Figure 11. Latency analysis: (a), (b), and (c) latency under single-thread (IOPS obtained from varying queue depths of 1 to 1024); (d) Throughput–latency analysis under multi-thread (IOPS obtained from varying thread counts of 1 to 5).



(a) Throughput in different NVMe SSDs with single thread

(b) Throughput under multiple threads.

(c) 99th latency at different queue depth.

(d) Average latency at different IOPS.

Figure 12. The file I/O performance.

the throughput of SPDK while maintaining comparable latency. At moderate IOPS (1 to 1.5 M), Hitchhike performs competitively, surpassing SPDK and traditional methods with 1.8 M IOPS at 62 µs latency. At lower IOPS, Hitchhike's latency is slightly above SPDK's but still close, indicating minor delays due to request merging. Hitchhike maintains stable latency across a wide range of IOPS (1 to 2.8 M), due to its efficient request merging and lower queue depth.

In our multi-threaded tests, Figure 11d highlights two key findings: 1) Hitchhike achieved the highest bandwidth (2.8 M IOPS) with 160 µs latency, comparable to SPDK, but significantly outperformed libaio, io_uring, and io_uring_cmd, which reached similar throughput only at much higher latencies (780 µs, 582 µs, and 565 µs, respectively). 2) Under multi-threaded scaling, Hitchhike—similar to io_uring—experienced a sharp increase in latency due to kernel I/O stack limitations, whereas SPDK was able to maintain low latency under the same conditions.

6.1.3 File I/O Path.

Throughput. as shown in Figure 12a, hitchhike-uring demonstrated significant improvements, especially on high-performance drives. On the H5300, hitchhike-uring achieved 1.6 M IOPS, outperforming libaio by 2.6×, io_uring by 2.3×, and io_uring-fb by 1.9×, with a 1.5× gain over io_uring-fb-iopoll. On the PM9A3 PCIe 4.0 SSD, hitchhike-uring reached 875 K IOPS, which is the maximum random IOPS capacity of the PM9A3, surpassing libaio by 1.5× and io_uring by 1.4×. The PM1743 saw hitchhike-uring reach 1.2 M IOPS, exceeding libaio by 2.1×, iouring by 2×, and iouring-fb by 1.8×. These

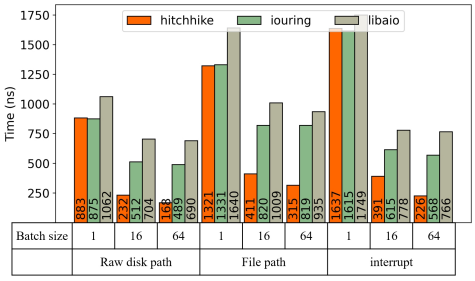


Figure 13. Submit time and NVMe interrupt time in different batch or merge sizes.

results indicate that while hitchhike-uring offers substantial throughput advantages, especially on high-performance SSDs, the file system's overhead somewhat limits its optimization potential, with the performance gains being more pronounced in raw disk I/O compared to file I/O.

In the multi-threaded scenario, as shown in Figure 12b, hitchhike-uring continues to demonstrate its superior thread scalability, fully utilizing all available bandwidth with just 2 threads and maintaining stable performance under multi-threading. In contrast, both io_uring and libaio require 6 CPU cores, which is 3× that of Hitchhike, and the assistance from fb and iopoll is not significant.

Latency. As shown in Figure 12c, benefiting from low software queue depth, Hitchhike maintains consistently low 99th percentile latency across all file I/O operations. Figure 12d further indicates that even though the presence of a file system reduces Hitchhike's throughput, Hitchhike achieves more than double the throughput of io_uring at the same latency and about 1.5× the improvement over the iopoll mode.

6.1.4 Software Overhead.

To evaluate the reduction in kernel processing time, we measured the submission time and the time spent on the NVMe interrupt. To eliminate interference from asynchronous interrupts, we configure libaio and io_uring so that the number of submitted requests in each batch matches the minimum number of completions. For Hitchhike, we merge an equivalent number of requests into a single Hitchhike operation. As shown in Figure 13, increasing the batch/merge size reduces the amortized submission time for libaio, io_uring, and Hitchhike. However, libaio and io_uring are limited by system call and scheduling overheads, so their latency levels off—e.g., 489ns (io_uring) and 690ns (libaio) for raw disk I/O, and 819ns and 935ns for file I/O. In contrast, Hitchhike achieves much lower latencies: 169ns (raw disk I/O) and 315ns (file I/O), reducing the overhead to 24–39% of traditional interfaces. Furthermore, merging reduces Hitchhike's amortized interrupt processing time from 1637ns to 226ns, which is 29.5% of libaio's and 39.8% of io_uring's. This reduction significantly contributes to Hitchhike's low latency performance.

Summary. Our evaluation on both raw disk and file I/O paths shows that Hitchhike consistently outperforms kernel-based and user-space I/O frameworks in terms of throughput. For latency, Hitchhike delivers better performance than kernel-based frameworks, matches SPDK in single-threaded scenarios, but falls behind SPDK under multi-threaded workloads. Its key advantage lies in achieving higher throughput with fewer CPU resources while maintaining more stable latency across diverse queue depths and throughput levels.

6.2 Graph Processing

We evaluated the execution time of Blaze using six graph datasets, including both synthetic and real-world graphs. The evaluation focused on four graph workloads: Breadth-First Search (BFS), Weakly Connected Component (WCC), Betweenness Centrality (BC), and PageRank (PR). As shown in Figure 14, compared to the libaio interface, hitchhike-aio reduces the execution time by 30% to 66%.

The performance differences observed in Blaze compared to the FIO in Section 5.2 primarily stem from two factors. First, these measurements reflect end-to-end application latency, where the proportion of time spent on actual I/O operations is lower than in FIO; according to Amdahl's Law, this naturally limits the potential gains from Hitchhike. Second, differences in algorithm behavior arise from Blaze's `EDGEMAP` API, where execution is divided into multiple rounds (sparse or dense). Algorithms with a higher proportion of sparse rounds, such as BFS and BC, incur more user-space lookup overhead, reducing Hitchhike's impact, whereas WCC and PR benefit from more concentrated requests per round and show greater relative improvement.

6.3 KV Store

We configured LeanStore with default key-value sizes (8B for keys and 120B for values) and tested it with cache sizes of 256 MB, 512 MB, 1 GB, and 2 GB. LeanStore was evaluated using YCSB, and Hitchhike demonstrated performance improvements ranging from 17% to 34% across workloads A, B, C, and F. Note that we did not test LeanStore's scan operation, as it does not appear to be batched.

Additionally, we draw the following two conclusions: **1)**. Hitchhike achieves lower speedup with LeanStore than with Blaze because Blaze uses dedicated I/O threads, while LeanStore handles both I/O and computation in one thread, resulting in a smaller fraction of kernel overhead. **2).** Hitchhike shows no performance gain on YCSB-D, as its latest-items distribution serves most reads from in-memory buffers rather than storage.

7 Related Work

Recent storage stack breakthroughs have highlighted kernel overheads' significance, prompting researchers to reevaluate and enhance the I/O architecture.

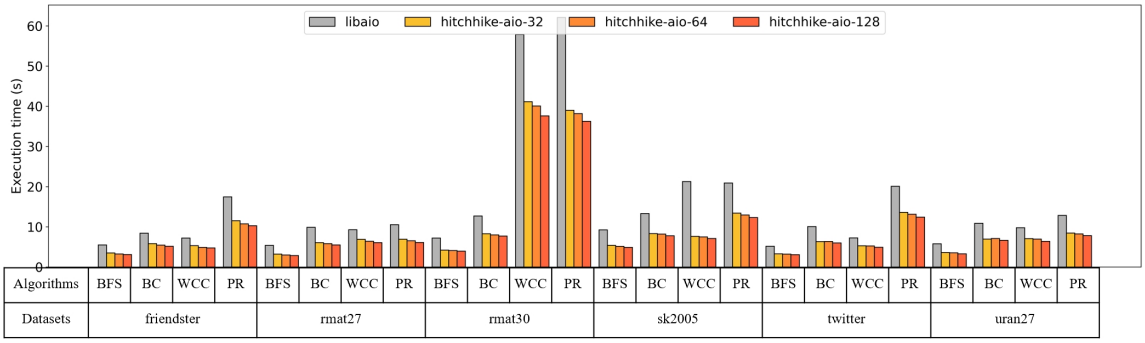


Figure 14. Execution time comparison for Blaze: Hitchhike-aio vs. libaio. The numbers 32, 64, and 128 denote merge sizes.

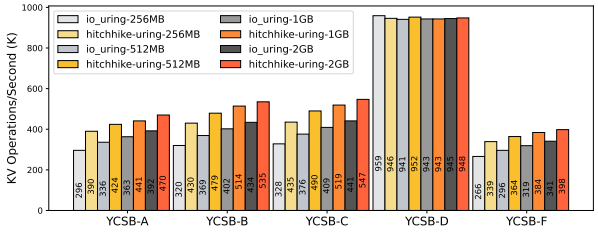


Figure 15. YSCB test for Leanstore: Hitchhike-uring versus io_uring with the standard I/O stack.

Kernel bypass. Kernel bypass can be broadly categorized into two approaches: The first, exemplified by SPDK [51], bypasses the entire kernel, implementing file systems, drivers, and other components in the user space. However, it places the burden of block management and file system implementation on applications, which can add considerable complexity. The second approach, represented by XRP [55], bypasses only specific parts of the Linux kernel. XRP allows the driver to access the file system to retrieve LBA and directly submit requests to devices, bypassing system calls and the block layer. BypassD [47] offloads file semantics to IOMMU to achieve fast userspace access to shared storage devices. I/O passthru [25], based on the io_uring command, directly submits requests to the driver, bypassing the file system. I/O passthru has been integrated into the mainline Linux kernel, but currently supports only block access, and its improvement on throughput remains limited. Hitchhike reduces kernel overhead without the need for bypassing, offering a more balanced and flexible solution.

Kernel modification. To tackle the software bottleneck in I/O performance, researchers have focused on optimizing the kernel I/O software stack, and proposed several optimizations to the kernel I/O, such as queue scheduling [8, 23, 53], reduced context switch overhead [41, 48, 56], and lightweight block layers [28]. These optimizations primarily focus on specific components of the I/O stack and are limited by their inability to eliminate the overheads imposed by the kernel's layered design. Additionally, they fail to address the inefficiencies that arise when dealing with small, random I/O requests, which still dominate many real-world workloads. This is where Hitchhike comes into play, tackling the broader issue of kernel-level redundancy by merging similar I/O operations, thereby reducing overall CPU consumption.

Computational storage devices. A completely different approach to avoiding kernel overhead is to reduce the frequency of device interactions by offloading storage functions directly to the device, as seen in several Computational Storage Device (CSD) systems [13, 15, 16, 24, 45]. The CSD devices have limited computational power, which makes handling complex operations difficult. They also require substantial changes to applications, and offloading tasks to the device is not always efficient [52], reducing flexibility and complicating integration with existing systems.

8 Conclusion

As storage hardware scales to support massive numbers of outstanding I/Os, the strict address contiguity validation in the software stack becomes a major bottleneck. Hitchhike addresses this inefficiency by deferring contiguity validation to the device driver, enabling flexible merging of non-contiguous requests while maintaining NVMe compliance. This design significantly reduces redundant kernel processing and CPU overhead. Our evaluation demonstrates that Hitchhike delivers substantial performance gains across both microbenchmarks and real-world workloads.

Acknowledgments

We would like to thank our shepherd, Ryan Stutsman, and the anonymous reviewers for their valuable feedback. This work was supported in part by the National Key R&D Program of China under Grant 2022YFB4501100; in part by the National Natural Science Foundation of China under Grant 62372197 and U22A2071; in part by the 111 Project (No. B07038); and in part by the Natural Science Foundation of Shandong Province (No. ZR2024LZH004).

References

[1] 2003. *Linux AIO*. https://github.com/littledan/linux-aio Accessed: 2024-08-30.

[2] 2006. Asynchronous I/O and vectored operations. https://lwn.net/Articles/170954/ Accessed: 2024-08-30.

[3] 2012. WiredTiger storage engine. https://docs:mongodb:com/manual/core/wiredtiger/ Accessed: 2024-08-30.

[4] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 6 (2023), 1426–1438. doi:10.14778/3583140.3583157

[5] Jens Axboe. et al. 2012. Flexible I/O Tester. https://github.com/axboe/fio Accessed: 2024-08-30.

[6] Jens Axboe. et al. 2024. *Liburing*. https://github.com/axboe/liburing Accessed: 2024-08-30.

[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. doi:10.1145/2485732.2485740

[8] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 385–395. doi:10.1109/MICRO.2010.33

[9] Jonathan Corbet. 2014. *The iov_iter interface*. https://lwn.net/Articles/625077/ Accessed: 2024-08-26.

[10] Jonathan Corbet. 2019. Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703/ Accessed: 2024-08-30.

[11] DapuStor. 2023. *H5300*. https://en.dapustor.com/product/9.html Accessed: 2024-08-26.

[12] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '22)*. Association for Computing Machinery, New York, NY, USA, 120–127. doi:10.1145/3534056.3534945

[13] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. 2019. Programmable solid-state storage in future cloud datacenters. *Commun. ACM* 62, 6 (May 2019), 54–62. doi:10.1145/3286588

[14] NVM Express. 2022. NVM Express Base Specification. https://nvmexpress.org/specification/nvm-express-base-specification/ Accessed: 2024-08-30.

[15] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: a framework for near-data processing of big data workloads. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 153–165. doi:10.1145/3007787.3001154

[16] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: a framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 153–165.

[17] Hao Guo and Youyou Lu. 2025. Achieving low-latency graph-based vector search via aligning best-first search algorithm with SSD. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation* (Boston, MA, USA) *(OSDI '25)*. USENIX Association, USA, Article 10, 16 pages.

[18] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 9 (May 2023), 2090–2102. doi:10.14778/3598581.3598584

[19] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. *Proc. VLDB Endow.* 17, 11 (2024), 3442–3455. doi:10.14778/3681954.3682012

[20] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 877–892. doi:10.1145/3318464.3389716

[21] Jun He, Kan Wu, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. Read as Needed: Building WiSER, a Flash-Optimized Search Engine. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 59–73. https://www.usenix.org/conference/fast20/presentation/he

[22] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. doi:10.1145/3282307

[23] 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 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 113–128. https://www.usenix.org/conference/osdi21/presentation/hwang

[24] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 373–384. doi:10.1109/HPCA.2017.15

[25] Kanchan Joshi, Anuj Gupta, Javier Gonz´lez, Ankit Kumar, Krishna Kanth Reddy, Arun George, Simon Lund, and Jens Axboe. 2024. I/O Passthru: upstreaming a flexible and efficient I/O path in Linux. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST '24)*. USENIX Association, USA, Article 7, 16 pages.

[26] Juno Kim and Steven Swanson. 2022. Blaze: fast graph processing on fast SSDs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 44, 15 pages.

[27] 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 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, Geoff Kuenning and Carl A. Waldspurger (Eds.). USENIX Association, 345–358. https://www.usenix.org/conference/fast17/technical-sessions/presentation/kim-sangwook

[28] 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 *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 603–616.

[29] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proc. VLDB Endow.* 17, 12 (2024), 4536–4545. doi:10.14778/3685800.3685915

[30] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (2023), 7:1–7:25. doi:10.1145/3588687

[31] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 447–461. doi:10.1145/3341301.3359628

[32] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. 2008. Merge: a programming model for heterogeneous multi-core systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, Susan J. Eggers and James R. Larus (Eds.). ACM, 287–296. doi:10.1145/1346281.1346318

[33] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, Geoff Kuenning and Carl A. Waldspurger (Eds.). USENIX Association, 285–300. https://www.usenix.org/conference/fast17/technical-sessions/presentation/liu

[34] Shuhao Liu, Yang Liu, and Wenfei Fan. 2024. PrismX: A Single-Machine System for Querying Big Graphs. *Proc. VLDB Endow.* 17, 12 (2024), 4485–4488. doi:10.14778/3685800.3685906

[35] Lam-Duy Nguyen and Viktor Leis. 2024. Why Files If You Have a DBMS?. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 3878–3892. doi:10.1109/ICDE60146.2024.00297

[36] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. doi:10.1007/S002360050048

[37] Zebin Ren and Animesh Trivedi. 2023. Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (Rome, Italy) *(CHEOPS '23)*. Association for Computing Machinery, New York, NY, USA, 35–45. doi:10.1145/3578353.3589545

[38] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.* 10, 1 (1992), 26–52. doi:10.1145/146941.146943

[39] Samsung. 2023. *PM1743*. https://download.semiconductor.samsung.com/resources/white-paper/PM1743_SSD_Whitepaper.pdf Accessed: 2024-08-26.

[40] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishnaswamy, and Harsha Simhadri. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS 2019*.

[41] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. 2022. Optimizing Storage Performance with Calibrated Interrupts. *ACM Trans. Storage* 18, 1, Article 3 (March 2022), 32 pages. doi:10.1145/3505139

[42] Demian E. Vöhringer and Viktor Leis. 2023. Write-Aware Timestamp Tracking: Effective and Efficient Page Replacement for Modern Hardware. *Proc. VLDB Endow.* 16, 11 (2023), 3323–3334. doi:10.14778/3611479.3611529

[43] Leonard von Merzljak, Philipp Fent, Thomas Neumann, and Jana Giceva. 2022. What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 36–46. http://www.adms-conf.org/2022-camera-ready/ADMS22_merzljak.pdf

[44] Rui Wang, Weixu Zong, Shuibing He, Xinyu Chen, Zhenxin Li, and Zheng Dang. 2024. Efficient Large Graph Processing with Chunk-Based Graph Representation Model. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang (Eds.). USENIX Association, 1239–1255. https://www.usenix.org/conference/atc24/presentation/wang-rui

[45] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 717–729. doi:10.1145/3445814.3446763

[46] Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, and Tei-Wei Kuo. 2024. How to Steal CPU Idle Time When Synchronous I/O Mode Becomes Promising. In *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC 2024, San Francisco, CA, USA, June 23-27, 2024*, Vivek De (Ed.). ACM, 150:1–150:6. doi:10.1145/3649329.3655929

[47] Sujay Yadalam, Chloe Alverti, Vasileios Karakostas, Jayneel Gandhi, and Michael Swift. 2024. BypassD: Enabling fast userspace access to shared SSDs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 35–51. doi:10.1145/3617232.3624854

[48] Jisoo Yang, Dave B. Minturn, and Frank T. Hady. 2012. When Poll Is Better than Interrupt. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*. USENIX Association, San Jose, CA.

[49] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. 2021. Random Walks on Huge Graphs at Cache Efficiency. 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, 311–326. doi:10.1145/3477132.3483575

[50] Tsun-Yu Yang, Yizou Chen, Yuhong Liang, and Ming-Chang Yang. 2024. Seraph: Towards Scalable and Efficient Fully-external Graph Computation via On-demand Processing. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 373–387.

[51] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 154–161. doi:10.1109/CloudCom.2017.14

[52] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. 2023. λ-IO: A Unified IO Stack for Computational Storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 347–362.

[53] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. 2014. Optimizing the Block I/O Subsystem for Fast Storage Devices. *ACM Trans. Comput. Syst.* 32, 2, Article 6 (June 2014), 48 pages. doi:10.1145/2619092

[54] Jie Zhang, Miryeong Kwon, Michael M. Swift, and Myoungsoo Jung. 2020. Scalable Parallel Flash Firmware for Many-core Architectures. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 121–136. https://www.usenix.org/conference/fast20/presentation/zhang-jie

[55] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 375–393.

[56] Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou, and Zhou Li. 2023. Userspace Bypass: Accelerating Syscall-intensive Applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 33–49.

[57] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 685–699. doi:10.1145/3514221.3526187

A Artifact Appendix

A.1 Abstract

This artifact provides the source code, scripts, and experimental workflows required to reproduce the key results presented in the paper "Hitchhike: Efficient Request Submission via Deferred Enforcement of Address Contiguity". The artifact includes the modified Linux kernel (v6.5) with Hitchhike support, adapted versions of standard storage benchmarks (FIO, SPDK), and integrated applications (Blaze, LeanStore).

A.2 Artifact Check-List

- **Program:** C, C++, Python, Shell scripts.
- **Compilation:** GCC, G++ (requires v7 for Blaze, v11 for others), Make.
- **Hardware:** x86_64 server with NVMe SSDs (PCIe 5.0 recommended). Evaluation uses 32 physical CPU cores.
- **Data set:** Generated workloads (YCSB) and Graph datasets (R-MAT, Twitter, etc.).
- **Operating System:** Ubuntu 22.04 LTS.
- **Kernel:** Modified Linux v6.5 (source provided).
- **Metrics:** IOPS, Throughput, Latency, Execution Time.
- **Output:** Console logs and parsed figures (Fig 8-11, 13, 14).
- **Experiments:** FIO microbenchmarks, LeanStore (YCSB), Blaze (Graph processing).
- **How to access:** Publicly available on GitHub.

A.3 Description

A.3.1 How to Access. The source code, scripts, and detailed documentation are hosted on GitHub:
https://github.com/haslaboratory/Hitchhike-AE

A.3.2 Hardware Dependencies. Hitchhike relies on NVMe SSDs. To fully reproduce the high-throughput benefits, PCIe 5.0 SSDs (e.g., Dapustor H5300, Samsung PM1743) are recommended. The experimental setup in the paper utilizes three NVMe drives to demonstrate scaling and different device characteristics.

A.3.3 Software Dependencies.

- **OS:** Ubuntu 22.04 LTS.
- **Compiler:** `g++-7` is specifically required for compiling Blaze; standard compilers are used for other components.
- **Dependencies:** SPDK, liburing (included as submodules).

A.4 Installation

The installation process is automated via scripts provided in the repository. Please refer to the `README.md` for the exact script names and execution order. The general procedure consists of three phases:

1. **Kernel Setup:** Clone the repository and run the kernel build script to compile and install the modified Linux v6.5 kernel. A system reboot into the Hitchhike kernel is mandatory.
2. **Device Configuration:** A configuration script is provided to identify and register the NVMe devices (PCIe 5.0 recommended) used for testing.
3. **Dependency Compilation:** Build the userspace components, including the modified FIO, SPDK, LeanStore, and Blaze, using the provided installation scripts.

A.5 Experiment Workflow

All evaluation scripts are organized in the `evaluation/` directory, categorized by the figure numbers in the paper.

- **Microbenchmarks (FIO):** The artifact includes scripts to reproduce the throughput, latency, and CPU utilization results (Fig. 8–11). Users need to run the setup scripts to prepare the raw disk or filesystem environment before executing the test scripts in each figure's subdirectory.
- **Application Benchmarks:**
  - **Blaze (Fig. 13):** Scripts are provided to download graph datasets and execute the graph processing workloads.
  - **LeanStore (Fig. 14):** Scripts are provided to compile the database and run the YCSB workload.

Detailed instructions for running specific experiments and parsing the results are documented in the repository.

A.6 Evaluation and Expected Results

The experiments are expected to demonstrate that Hitchhike significantly improves throughput and reduces CPU overhead compared to standard Linux asynchronous I/O interfaces, particularly in high-IOPS scenarios on PCIe 5.0 SSDs.