Selective On-Device Execution of Data-Dependent Read I/Os

Submission #290

Abstract

Recent studies have demonstrated the benefits of employing on-device and in-kernel storage functions. On-device functions are primarily used to preprocess data within storage devices, effectively reducing the amount of I/O. In contrast, in-kernel functions are proposed to expedite sequences of data-dependent read I/O requests, particularly useful for applications traversing on-disk data structures. In this work, we investigate the unexplored potential of using on-device functions for data-dependent read I/O requests on read-only on-disk data structures. The results are promising: on-device I/O functions enable applications to issue I/O requests more rapidly and integrate seamlessly with in-kernel functions to efficiently manage high volumes of requests. We developed a prototype of this on-device function atop NVMeVirt, a state-of-the-art storage emulator. We demonstrate that on-device function enhances performance through experiments utilizing a simple B^+-tree key-value store and WiredTiger, a widely used log-structured merge tree-based key-value store. Use of the on-device function improves the throughput of the B^+-tree key-value store by up to 41%, and reduces WiredTiger's 99-percentile tail latency on YCSB C by up to 3.39%, compared to the host-only in-kernel storage function.

1 Introduction

Near- and in-storage computing is getting traction for its advantage in performance, energy efficiency, and scalability [42, 71, 72, 76]. The increasing amounts of data require large storage capacity with high bandwidth. Storage vendors are fulfilling this demand by developing devices with higher bandwidth and lower latencies, and applications often use remote storage to have even larger storage spaces. Consequently, storage devices are located relatively far from the primary processors (*i.e.*, CPUs). Even the local storage devices are usually connected over Peripheral Component Interconnect Express (PCIe), whose round-trip time is at a microsecond scale. Because of this remoteness, processors located on the storage side of the interconnect are expected to benefit from higher bandwidth and lower latency compared to primary processors such as CPUs.

In applications using PCIe-connected local storages, prior works focus on placing data-intensive kernels on the storage side inside the device [8, 27, 61, 72]. The storage device, often a solid-state disk (SSD), is the only available choice for this compute offloading in the local setting. It is also a natural target because an SSD is a separate computer system with processors and memory in addition to the flash memory as the persistent data store [65]. A pioneering study [64] suggested the potential advantage in access latency when offloading computes to an SSD device. However, most recent studies [42, 61, 72] aimed to take advantage of higher bandwidth within the device. Specifically, these systems are primarily optimized for specific kernels, designed to produce relatively small results from a large volume of data or generate extensive outcomes from minimal input.

Recent studies, XRP [76] and BypassD [71], revisited another direction where they utilize the lower latency of near-storage computing, motivated by the introduction of low-latency storage devices. Specifically, XRP proposed placing compute at the lowest possible layer on the host software stack, just above or within the device driver, to issue I/O requests. XRP is particularly beneficial to a workload pattern named *resubmission* of I/Os [76] where a storage function places a data-dependent I/O request, for example, to traverse on-disk data structures. BypassD demonstrated the possibility of bypassing the kernel with the help of architectural support for access control, effectively reducing the distance between user-level software and the storage device.

We explore the remaining direction, demonstrating that computing within the storage devices also expedites I/O resubmissions for data-dependent reads when collaborating with in-kernel storage functions. Recent studies using on-device computing primarily target throughput-centric workloads for several reasons. First, storage devices are expected to have relatively low computing power, especially regarding the processor's operational frequency. As a result, each task is likely

to experience higher latency on the device than in the kernel, losing the advantage of lower data access latency. Second, existing commercial devices that can run such on-device compute [62] are not designed for the potential advantage in access latency. The internal storage and on-device computing resource, which comprises a Field-Programmable Gate Array (FPGA), are connected via a PCIe interconnect. Consequently, the access latency from the on-device FPGA is not significantly lower than that from the host processor. These two observations highlight the constraints under which on-device computing can benefit from low access latency. First, the dispatched task must not have long latency; it must produce the subsequent storage read request quickly enough. Otherwise, the benefit of low data access latency will be outweighed by the increased task execution time. Second, the processor must be located within the device, possibly within the same System-on-a-Chip (SoC) as the storage device controller.

This paper demonstrates that applications performing frequent data-dependent storage reads, especially those using on-disk data structures, can benefit from an on-device computing environment that satisfies the two constraints mentioned above. As an example, we design and evaluate the *Selective On-Device Execution (*SODE*)*, which enables the adaptive use of on-device and in-kernel storage functions. SODE runs the *resubmission task*, a storage function performing data-dependent reads to place a subsequent read I/O request if needed, taking the result of the previous read as the input. As the name suggests, SODE resubmits data-dependent I/O requests within a storage device only when doing so is expected to improve performance.

To this end, we find the following design choices essential for unleashing the full potential of on-device computing resources. First, SODE runs the resubmission task on the device only if the on-device processors can readily perform it. If the on-device processors are already occupied and the delay is expected to be longer than the host-device latency, SODE falls back to the in-kernel path, *i.e.*, *reverse (R)-offloads* the resubmission task to the host. Second, we choose to resubmit optimistically using cached file system metadata despite the possibility of host-device inconsistency. On-device resubmission tasks must translate file offsets into the logical block address, which requires access to the latest file metadata. However, retrieving the metadata from the host for every resubmission cannot be an option because it accompanies a PCIe round trip, the one that SODE is designed to avoid. We overcome this by learning from an observation made by an earlier work [76]. Resubmission requests are mostly made to obtain a specific data block from the disk and to read only or mostly read files. That is, the file system metadata remains unchanged for almost all cases during the resubmission. With this observation, SODE chooses to examine if the metadata has changed only at the end of each resubmission chain when the response from the disk arrives at the host. Third, SODE enables parallelized resubmission tasks to broaden its versatility. As the first constraint suggests, the latency of the resubmission task is a critical factor in affecting whether an application of its operation can benefit from on-device resubmission. We observe that resubmission tasks often determine whether or not to resubmit the request and its address by traversing on-disk data blocks, and such tasks can efficiently be parallelized to lower the execution time. In addition to these three design choices, SODE prototype also considers sandboxing of resubmission tasks within devices using the Extended Berkeley Packet Filter (eBPF) and in-process isolation techniques, following the design of prior works [25, 47, 72]. Resubmission tasks must be sandboxed within the storage because they could arbitrarily read or corrupt the underlying software stack and may also place an arbitrary I/O request, bypassing the file system's access control.

We implemented SODE on top of a state-of-the-art storage emulator, NVMeVirt [33]. We chose to use this emulator to explore the scenario where wimpy processors run inside the storage device right next to the controller, using modern low-latency storage technologies. NVMeVirt has already been shown to emulate various NVMe-based storage devices precisely, and we could emulate wimpy processors by running the emulating threads with limited operational frequency (*e.g.*, 1.2GHz). Our evaluation using this prototype shows that SODE improves the throughput of a simple B^+-tree key-value store from the prior work [76] by up to 41% and reduces the 99-percentile tail latency of WiredTiger on YCSB [9] C up to 3.39%.

We acknowledge that we will open-source our implementation of SODE and the experimental settings to support open science.

2 Backgrounds and Motivation

2.1 In-storage Computing

Recent studies have witnessed the potential benefit of in-storage computing [20, 35, 37, 58, 64, 72]. One well-known advantage of computing in storage is the reduced data movement between the host and storage devices. Many data-intensive operations such as filtering, selection, or reduction produce a small amount of data from a large amount of input data. Performing such data inside the storage significantly reduces the data transfer latency from the device to the host, improving the end-to-end latency of the task. This advantage is evidenced by the success of recently proposed in-storage computing systems. For example, Summarizer [35] presented the potential benefit of utilizing in-SSD computing resources, Insider [61] enables to deploy FPGA-based accelerators in the SSDs, λ-IO [72] provides a unified abstraction to applications to deploy in-storage computes only when the data actually resides in the storage devices, and Sand [20] delegates data preprocessing for video-based deep learning to storage.

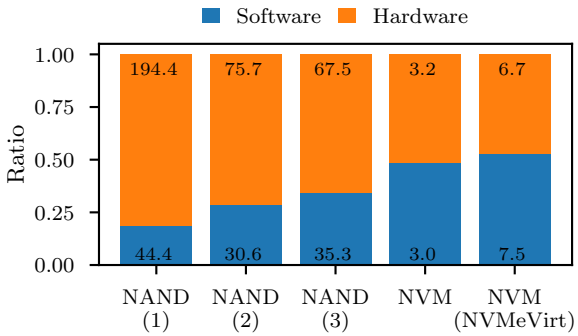


Fig. 1: The breakdown of 512 B random read latency (μs) when using five kinds of storage devices. NAND (1), NAND (2), and NAND (3) are the results obtained using Intel D3-S4520, Samsung 980 Pro, and Samsung Z-SSD, all using NAND as their storage elements. NVM is the result that is reported from XRP [76] using Intel Optane P5800X, and NVM (NVMeVirt) is the result we obtained using NVMeVirt [33] that emulates Intel P4800X SSD.

2.2 Express Resubmission Path

In- or near-storage computing brings the benefit of access latency as well as bandwidth. Willow [64] has already pioneered the benefit of low access latency when running data-dependent logic in storage, such as data-dependent reads. Splinter [37] demonstrated a similar advantage when pushing compute to remote storage servers. Most recently, XRP [76] has shown that the storage device driver inside the host kernel is considerably closer to the storage than the user space when an application interacts with modern low-latency NVMe-based storage. As Fig. 1 shows, a reduction in the storage device latency relatively magnifies the impact of the software layer's latency, calling for mechanisms to bypass the software stack. In response, XRP demonstrated that pointer-chasing workloads over an extensive in-storage database can be expedited by being performed at the device driver level, which is the lowest possible layer in the host storage stack.

2.3 Extended Berkeley Packer Filter (eBPF)

eBPF [2] is a framework that enables executing a small piece of user-written code as a part of the Linux kernel. As the name suggests, eBPF has been developed as a means to filter network packets [19] but is now used as a general means to execute application's code in the kernel for performance measurement [16], scheduling [21, 29], or security checks [1]. Filters in eBPF are represented using virtual registers and instructions that the framework defines. The kernel invokes the filters using the virtual machine or after being compiled into the machine code by the kernel's Just-In-Time (JIT) compiler. The filters undergo strict verification to ensure they cannot be leveraged to manipulate the kernel behavior maliciously. Although recent studies are still ongoing to evaluate the possibility of an eBPF filter being misused [24], recent studies show that eBPF filters can be effectively quarantined using in-process isolation techniques [5, 25, 44, 47]. These characteristics of being an in-kernel quarantined execution environment make eBPF a viable option for delegating resubmission functions closer to the storage device, just above the device driver [76]. We also build SODE using the same building block by choosing eBPF filters as the form of resubmission task to be executed on-device and in-kernel because the resubmission task must be quarantined when running on the device. If not quarantined, an adversarial application may deploy a malicious resubmission task to manipulate the on-device software stack or arbitrarily access the storage content.

2.4 Potential Benefits of On-device Resubmission

This work explores the potential benefits of using on-device processors to execute resubmission tasks. The two observations described below suggest the possibility of improving resubmission task performance using on-device processors, but none of the prior studies demonstrated the benefit. First, modern SSDs still have general-purpose processors within their controller SoC [65]. Though the processor's computing power is expected to be limited, it is significantly closer to the storage element than the in-kernel storage function, which runs at least across the PCIe interconnect. Second, resubmission within a storage device brings advantages beyond the round-trip latency of PCIe interconnects. A recent study on optimizing the storage stack [39] reported that it again takes microsecond-scale time for the device driver to complete an NVMe request, which individual in-kernel resubmission must go through. In other words, on-device resubmission is expected to have advantages in storage access latency beyond the PCIe interconnect round trip.

3 Design

Design Goals. We aim to explore the potential of using on-device processors to enhance the performance of applications that benefit from near-storage resubmission. An example of such applications is key-value stores, which traverse on-disk data structures to retrieve values associated with queried keys. For these applications, we designed SODE to simultaneously improve throughput, average latency, and tail latency of target operations. We do not aim to trade off one metric for another in this work, as all three metrics are critical when assessing the performance of key-value stores.

Overview. Fig. 2 shows an application using SODE by invoking a new system call, `read_sode`. From the application's perspective, this system call's semantics is the same as `read_xrp` that a previously proposed in-kernel storage function framework [76] provides. The difference is how the

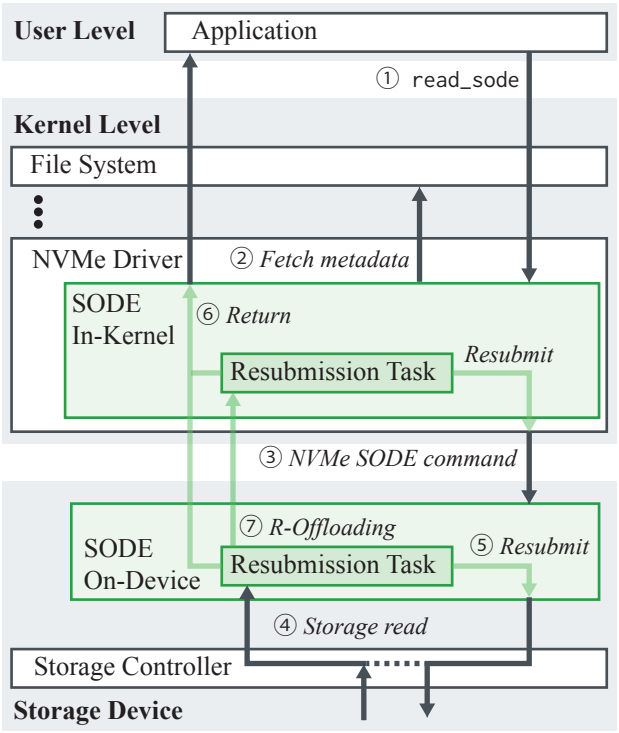


Fig. 2: The overview of SODE.

kernel serves the request in collaboration with the on-device runtime. Receiving a `read_sode` call (①), SODE delivers the request to the on-device runtime along with metadata that SODE needs for the request's resubmission (see §3.2). Specifically, SODE first obtains a copy of the mapping from file offsets to the logical block address from the file system (②). It then submits a special NVMe command, called NVMe SODE command, (③) that contains the information about the first read position, the digest of mapping, and the storage function to run within the device. After completing the first read in the device (④), SODE's on-device runtime executes the resubmission task to determine if it should read again using a newly computed file offset (⑤) or respond to the application (⑥). If the on-devices processors are too busy and expected to delay incoming resubmission requests, SODE falls back to the in-kernel resubmission path by responding to the host with a special flag (⑦), which we call *Reverse-(R)-Offloading* (see §3.1).

SODE is built on three design choices that are indispensable in achieving the design goal.

D1: Hybrid Resubmission. Storage devices are expected to have processors with limited computing power (*e.g.*, processors operating at low clock frequency). To overcome this challenge, we propose to offload the resubmission task from the device to the host kernel, utilizing the existing in-kernel resubmission path, as detailed in §3.1.

D2: Optimistic Resubmission with Cached Metadata. An I/O request can be resubmitted only after the translation of file offsets into the logical block address. The translation can be done only with the file metadata, which the on-device runtime does not have direct access to. To avoid locking the file and suffering from the PCIe round trip latencies, SODE chooses to optimistically execute a resubmission chain using a copy of metadata cached on the device and examine the validity at the time of completion. §3.2 details this design choice.

D3: Support for Parallel Resubmission Tasks. SODE also enables to write resubmission tasks as multiple eBPF filters running in parallel. As Table 1 shows, limited computing power on the device leads to increased task execution latency, potentially nullifying the advantage of near-storage resubmission. As §3.3 details, the resubmission tasks are likely to have some parallelism, which SODE enables the developer to exploit to reduce the on-device execution time of the resubmission tasks, thereby enjoying the benefits of on-device resubmission.

Table 1: Latency of the in-kernel and on-device resubmission.

Workload	In-Kernel	On-Device
BPF-KV	418ns	633ns
WiredTiger	1867ns	4403ns

3.1 Hybrid Resubmission with On-device Runtime

SODE uses its on-device path only as an auxiliary resource complementing the in-kernel path to take advantage of both worlds. Upon receiving an NVMe command generated from a `read_sode` request, its on-device runtime determines where to execute the resubmission task. SODE assigns the task to the on-device worker threads if the runtime can execute it promptly; otherwise, the task is forwarded to the host's in-kernel path.

Behind this design choice are our observations on the sources of worst-case lookup latency, the assumption of wimpy on-device processors, and our goal to avoid sacrificing tail latency. If no processors are promptly available to handle a resubmission task, it should be queued either within the device or in the kernel. The time a task waits in a queue determines the latency of worst-case resubmissions, which affects the tail latency if the resubmission tasks have the same execution time and other storage-related latencies. The queuing delay is determined by the number of items in the queue and their total execution time. This delay is exacerbated on the device, where the execution time of the resubmission task becomes longer, thereby justifying our design choice. For this reason, the aforementioned design choice where SODE queues a resubmission task only at the in-kernel path outperforms in general, *i.e.*, exhibits lower latency and higher throughput.

```
struct bpf_sode {
    // Fields inspected outside BPF
    char *data;
    int done;
    uint64_t next_addr[16];
    uint64_t size[16];
    // Field for BPF function use only
    char *scratch;
    // Field for parallel resubmission
    int job;
};
uint32_t BPF_PROG_TYPE_SODE(
    struct bpf_sode *ctxt);
```

Fig. 3: Signature of BPF programs representing a resubmission task written for SODE. We intentionally remain compatible with the signature used by XRP's resubmission tasks.

BPF Hook. SODE's on-device path is designed to run resubmission tasks written as eBPF filters. It creates a new BPF type (`BPF_PROG_TYPE_SODE`) as shown in Fig. 3, whose structure is the same as the one from XRP [76], the in-kernel path of SODE. When running on the device, `scratch` points to a memory region within the device, containing the application-specific data. The inputs (*e.g.*, lookup key) from the application is delivered through this region to the resubmission task, and the result is also retrieved from here. The `data` field points to a memory region where the on-device runtime places the data from the disk. SODE allocates one 4KB page for each of these fields. We also use the eBPF verifier from XRP because resubmission tasks written for SODE can be expressed within the constraint of XRP's verifier.

3.2 Optimistic Resubmission with Cached Metadata

SODE optimistically performs resubmission using cached file metadata and examines its validity when completing a series of resubmissions. Resubmission tasks compute the file offset of the subsequent disk read using the content of previously obtained data. This file offset must be translated into the logical block address, and SODE needs the file system metadata for this purpose. For example, the ext4 file system uses metadata called *extent status tree* for address translation, and it is all SODE needs to perform the translation. SODE creates a copy of this metadata when it begins to handle a resubmitting request (*i.e.*, an invocation of `read_sode`) and sends it to the device as a part of an NVMe command. It runs the resubmission tasks assuming that the metadata remains unchanged and later examines if the metadata has not been changed until the completion of the entire resubmission request. The result of execution is forwarded to the application only if the metadata is found unchanged; otherwise, SODE aborts the request because the metadata is supposed to remain unchanged. If the application ensures that only one thread, which has issued a resubmission task to a file, reads from the file during the resubmission, the metadata remains unchanged and thus SODE is unlikely to result in aborting the request. Note that we have not observed any metadata mismatch in our experiments because our resubmission requests are sent exclusively to read-only files.

The combination of two observations about the target workload and the execution environments led us to this design choice, which differs from the ones found in two previous works, XRP [76] and λ-IO [72]. SODE's resubmission tasks may run either in the kernel or the device like λ-IO, necessitating it to send the metadata to the device. However, files to which the application places resubmission requests are assumed to be updated infrequently, if ever, enabling SODE not to follow λ-IO's design choice where it locks the file during the execution of on-device compute. XRP's design choice is also not an option because SODE may run the resubmission tasks on the device, which the file system cannot promptly push the metadata updates to. In summary, the difficulty of prompt metadata synchronization mandates the use of cached copies, and the assumption of infrequent updates enables the optimistic resubmission with a lazy metadata validity check.

NVMe Command Extended for SODE. We assume the existence of a new NVMe command that SODE can use to dispatch resubmission tasks and implement it by extending NVMeVirt. We extend the NVMeVirt to support a new opcode, representing an NVMe command for a read request with a resubmission task. The SODE command contains an address of the shared buffer from which NVMeVirt obtains additional information about the command, such as the eBPF program or file metadata through direct memory access (DMA). Specifically, the command carries the address in one of the reserved fields of an NVMe read/write command. Our current implementation uses this command to attach the eBPF program and file metadata to every SODE command, but we anticipate that we can improve this by caching them on the device in future versions.

3.3 Support for Parallel Resubmission Tasks

SODE also allows the application to invoke resubmission tasks composed of multiple eBPF filters running in parallel. As mentioned earlier, we make this design choice because 1) the execution time of a resubmission task increases when dispatched to the device, and 2) on-device resubmission improves the performance only when the difference in resubmission task execution time is smaller than the advantage of avoiding host-device round trip. SODE assumes that the application writes the parallel resubmission tasks as a homogenous eBPF filter and invokes a separate system call, `read_sode_parallel`, to request a parallel resubmission. Specifically, we design SODE's parallel resubmission support with the typical parallelizable resubmission pattern, which is to look

up the offset or address for the subsequent request submission. Such a task is assumed to take a data block as an input and to produce the subsequent request as the output. When a parallel resubmission task is invoked, each of its instances receives a part of the data block to search for the address for the subsequent request, and records the result to a scratch page allocated by the on-device runtime. One of the instances, which is designated as the leader, is responsible for copying each result of instances to the main scratch page and awaits all other instances to terminate.

3.4 Sandboxing On-Device Function Executions

SODE must sandbox the on-device resubmission tasks. The task's capabilities should be strictly limited to the resources assigned to itself, including the memory and disk. A resubmission task must be constrained to use the memory region allocated for it, and unauthorized disk access attempts must be prohibited as specified in the file metadata. SODE uses mechanisms introduced in prior works, XRP and λ-IO. It executes resubmission tasks written as eBPF filters, naturally integrating existing sandboxing techniques. After verifying the safety of these filters using the existing eBPF filter verifier, we utilize the eBPF runtime to compile the filters into machine code. SODE uses the cached file metadata it receives to ensure that the logical block address generated from the resubmission task lies within the granted range.

We acknowledge that recent studies have identified vulnerabilities in the eBPF subsystem, which could be exploited to compromise the Linux kernel or software running an eBPF filter [24, 51–56]. Consequently, additional sandboxing techniques, such as software fault isolation (SFI) [68] or protection keys [46], are necessary for enhanced security. These techniques have been shown to sandbox eBPF filters [5, 25, 44, 47] effectively, and SODE can adopt them.

Our prototype accounts for the potential performance impact of such sandboxing mechanisms by entering and exiting a hardware-enforced sandbox before and after executing a resubmission task. Specifically, our prototype uses the Memory Protection Key (MPK), where each memory page can be associated with one of 16 protection keys and the permissions of the currently running thread can be modified by writing to a special register called `PKRU`. Before executing a resubmission task, SODE preserves the content of `PKRU` using the `RDPKRU` instruction. It then modifies `PKRU` using the `WRPKRU` instruction to restrict the thread's access to only the memory regions permitted for the resubmission task. Upon task completion, SODE restores the original `PKRU` content.

4 Implementation

We implemented a prototype of SODE by extending NVMeVirt to emulate computational storage devices that run resubmission tasks written as eBPF filters. NVMeVirt [32, 33] is a recently developed software-based emulator that enables rapid prototyping and performance analysis of new storage devices. NVMeVirt has been shown to correctly simulate the performance characteristics of NVMe-based storage devices when evaluating various storage and file system benchmarks. The emulator is written as a loadable kernel module (LKM) and uses a reserved DRAM space to store data. It runs on its own socket with the socket's DRAM on a two-socket machine, where another socket is dedicated to running the host. Among several available emulation targets of NVMeVirt, we use the one targeting an Intel OptaneDC SSD [23] because SODE is designed for emerging low-latency storage devices.

4.1 On-device Resubmission Path

We implement an on-device resubmission path on NVMeVirt by adding *resubmission threads* that emulate the behavior of the on-device resubmission path. The threads interact with two kinds threads in NVMeVirt, *dispatcher thread* and *I/O thread*, that emulate the target NVMe device. The dispatcher thread takes the incoming NVMe packets from the submission queue and handles the request by forwarding it to the I/O thread. The I/O thread emulates the behavior of a storage device by reading from and writing to the memory space reserved for NVMeVirt. It also emulates the access latency by adding additional delays when needed. After handling the request, the I/O thread places the completion packet on the storage device's completion queue.

Leaving these core components of NVMeVirt intact for precise emulation, we let the resubmission threads issue the requests to the I/O thread and work with the results. After handling a read request, the I/O thread forwards the result to one of the resubmission threads instead of directly writing to the completion queue. The resubmission thread then runs the task using the eBPF filter implementing the task, the file metadata for address translation, and the data page containing the result of the previous read. After the execution, the resubmission thread issues the subsequent resubmitting request to the I/O thread if needed or writes the result to the completion queue. This design preserves NVMeVirt's latency emulation because in NVMeVirt, the I/O thread marks a request as handled only after a specific period of time, if necessary, for latency emulation. The latency of individual requests from either the submission queue or the resubmission threads is also emulated separately because the I/O thread does not distinguish between these two. We also minimize the interaction between the resubmission threads and the other kernel threads by binding each of the four resubmission threads to its processor core and letting them yield their cores as little as possible.

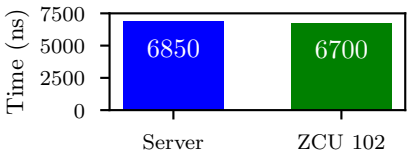


Fig. 4: Simple `get` operation latency on the server with its CPU operating at 1.2GHz and on a Zynq UltraScale+ MPSoC ZCU 102 development board, which has a quad-core ARM Cortex-A53. This Cortex-A53 operates at 1.2–1.334Ghz.

4.2 Emulating Wimpy On-device Cores

We account for the computing power that the on-device path affords in emulation. Computational storage devices will prefer not to consume significantly more energy or incorporate advanced cooling systems, limiting the performance of on-device processors. Specifically, we assume that on-device resubmission tasks run on processors whose performance is similar to low-power application-profile ARM processors, such as the ARM Cortex-A53. This assumption aligns with the testbed configuration in a recent work [72] on computational storage devices, where the device is emulated with a System-on-a-Chip (SoC) having four Cortex-A53 processors.

We emulate the performance of on-device processors using frequency scaling. The Xeon processors that we use in our evaluation can be configured to operate at 1.2 GHz, which we find appropriate for our performance emulation. To validate this configuration, we compared the performance of our Xeon processor running at 1.2 GHz with that of a system having four Cortex-A53 processors. We implemented a microbenchmark that mimics the behavior of the lookup operations to key-value stores. We ran this benchmark on our server processor operating at 1.2 GHz and a Zynq ZCU120 development board [70], which has four Cortex-A53 processors running at 1.2 GHz. Fig. 4 shows that our configuration reasonably emulates the performance of on-device processors as both exhibit similar execution times when running our microbenchmark.

4.3 Extent Caching and Validity Check

SODE creates a copy of the extent tree when it begins to handle an invocation of a resubmission request. Our current implementation chooses to copy the entire extent tree and sends it to the device along with every NVMe SODE command because the size of the extent tree is small (< 2 KB) in practice. The size of the extent tree depends on the number of fragments comprising a file and may grow if the file continues to be modified, but SODE targets rarely updated files where the file is unlikely to become more fragmented. Caching the extent tree within the storage device even after handling one invocation for reuse would help overcome the additional latency increase when SODE faces a file with a large extent tree. However, our current implementation does not have the feature and caches the metadata only for one resubmission request, which could involve multiple disk reads.

SODE ensures the validity of the extent tree during a resubmission chain by examining whether it remains unchanged when it completes a resubmission request. Specifically, each resubmission request also has a copy of the extent tree version number, which the file system updates its own copy whenever it modifies the extent tree. SODE compares the copied version number with the one managed by the file system when completing a resubmission request to compare if two extent trees remain the same quickly. As described in §3.2, our prototype aborts the resubmission request if the two version numbers do not match. During our evaluation, we did not observe any such version mismatches.

4.4 NVMeVirt Configuration Details

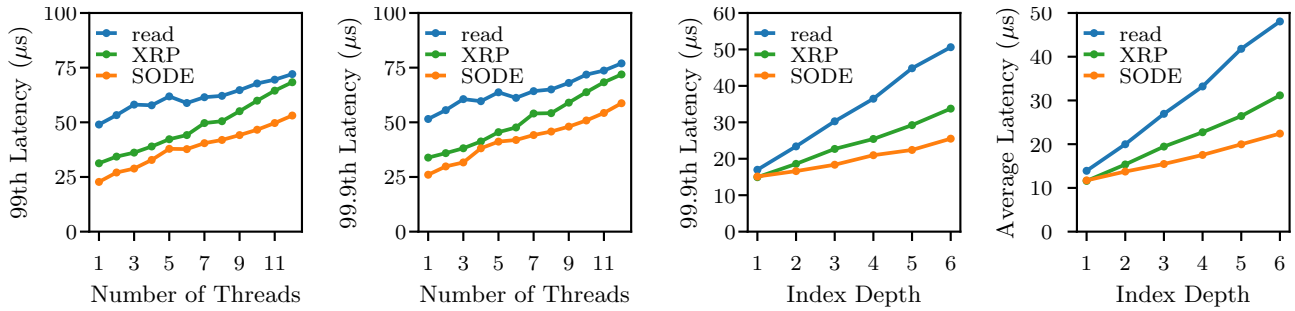
We use the NVMeVirt emulating an Intel Optane DC SSD [23] in this work because we aim to evaluate the potential benefit of SODE when using emerging low-latency storage devices. This model is also called the simple model in NVMeVirt. As discussed earlier, we leave the core parameters and building blocks, such as the target latency and two threads, intact except for the means NVMeVirt uses to copy the data to and from its reserved memory space. The NVMeVirt we use copies data between the reserved memory and the other buffers using `memcpy` instead of the Intel I/OAT [59] DMA engine because, at the time of this work, the version of NVMeVirt, whose commit ID is `b4c3c9d`, did not support I/OAT DMA. We confirmed that the use of `memcpy` instead of Intel I/OAT DMA does not hinder the target latency emulation when we use the Intel Optane DC SSD model, by measuring the latency of the `memcpy`-based configuration.

4.5 Adapting WiredTiger for Parallel Resubmission

We implemented a parallelized resubmission task for our experiments using WiredTiger by modifying 40 LoC (Lines of Code) of the resubmission task written for XRP. When compared to the XRP's resubmission task, the parallelized one conveys the start address and range of each subtask's key lookup in the unused bits in the page header. This change also requires 70 lines of changes to the WiredTiger. Only with these start addresses, the subtasks are enabled to perform lookup from the middle of the data page. We present the eBPF filter written for WiredTiger further in the supplemental material.

5 Evaluation

To demonstrate the potential performance benefit of SODE, we ran experiments to answer the following questions in this section:



(a) 99th-percentile latency with index depth 6

(b) 99.9th-percentile latency with index depth 6

(c) 99.9th-percentile latency with single thread

(d) Average latency with single thread

Fig. 5: Latency of random key lookup operations when running BPF-KV with `read`, XRP, or SODE.

- Does SODE improve the read performance of on-disk key-value stores? (§5.1)
- Does the benefit of SODE persist when a key-value store is heavily loaded? (§5.1.1)
- Does SODE improve the performance of a persistent key-value store, WiredTiger? (§5.2)

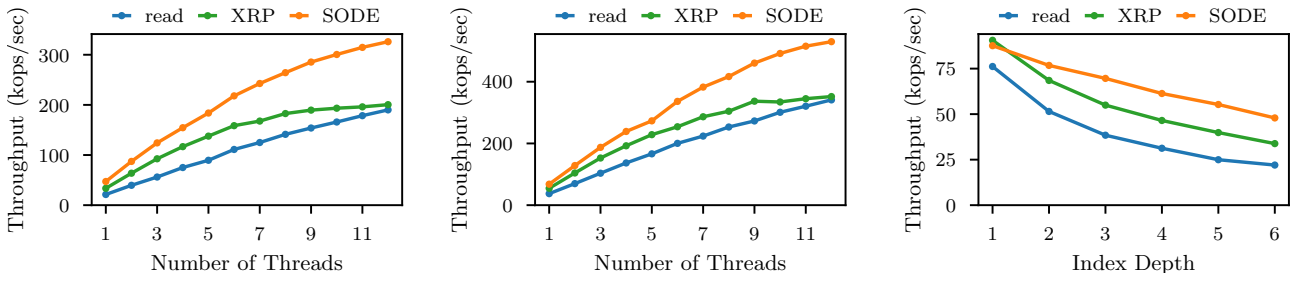
Environment. To evaluate SODE, we used a server equipped with Intel Xeon Gold 6136 processors running at 3.00GHz in a Non-Uniform Memory Access (NUMA) configuration. This server has two NUMA nodes, each having 12 cores and 192 GB of memory. The operating system is Ubuntu 18.04 with Linux kernel 5.12.0, that is adapted for SODE. When measuring the baseline performance using the `read` system call and XRP, we switched to the XRP's Linux kernel 5.12.0 with unmodified NVMeVirt. We ran all experiments using `O_DIRECT` after turning off hyper-threading, processor C-states, and turbo boost. Kernel page table isolation (KPTI) remains enabled in all experiments. The performance governor was set to maximum performance for the cores of node 0 (used for the host programs). We partially configured the cores of node 1 (assigned for NVMeVirt) to 1.2GHz using the user-mode governor to emulate the performance of embedded processors of storage devices as we explained in §4.2. Our experiments use the BPF-KV provided by XRP and WiredTiger [57] 4.4.0. To be specific, we used XRP from commit ID `9be399a`. Unfortunately, we were unable to include a comparison with SPDK [74] in our study, as NVMeVirt does not support SPDK as of commit ID `b4c3c9d`.

NVMeVirt Configuration. We followed the default configuration of NVMeVirt for Intel Optane SSD in all experiments as detailed in §4.4, with the following machine-specific details. Our machine has two processors in two sockets, and we dedicate one processor (node 1) of these two to NVMeVirt, which is extended for SODE's on-device path. Among the cores in node 1, we assign one core to a dispatcher thread, four cores to I/O threads, and four cores to resubmission threads, emulating a quad-core processor in the presumed in-storage computing environment. The remaining cores in the node 1 are turned offline. Although NVMeVirt originally used only one I/O worker thread, we increased the number of I/O threads to achieve more accurate latency emulation. This change was necessary because the high volume of requests was causing delays, which interfered with accurate latency emulation. The 192 GB of memory in node 1 is reserved as the storage space where NVMeVirt stores the contents. The other processor (node 0) is used to emulate the host system where the application threads and the host Linux kernel run, utilizing 192 GB of memory. We use `numactl` [12] to bind the application's processor and memory. NVMeVirt supports up to 72 submission and completion queue pairs, so each core in the host-emulating processor (node 0) has its own queue pair. For the kernel I/O scheduler, we configured SODE to use the `noop` scheduler, following the approach of XRP [76].

5.1 BPF-KV

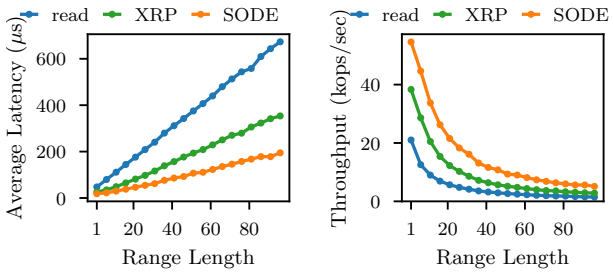
We used the BPF-KV of XRP to evaluate SODE after replacing `read_xrp` with `read_sode`. BPF-KV is a simple key-value store that can store many tiny objects and provide good read performance even under uniform access patterns. The key-value store has fixed-sized keys (8 B) and values (64 B) and uses a B$^+$-tree index to locate the objects. It also has a user-space DRAM cache for index blocks and objects, where it holds the least recently used (LRU) object.

Latency. Fig. 5 presents the random single key lookup latency of BPF-KV using SODE compared to those using either the `read` system call or XRP. The four graphs show that SODE reduces the average and tail latency of the lookup operation. Fig. 5(a) and Fig. 5(b) show that SODE outperforms XRP regardless of the number of application threads in terms of tail latency. 99th and 99.9th percentile tail latency of SODE is 11.6–37.5% and 8.2–30.1% lower than `read` or XRP, respectively. Fig. 5(c) shows that SODE remains to be superior to XRP or `read` by up to 32.3%, when the index depth (*i.e.*, the depth of the tree) changes, except for the case where the index depth is 1. The benefit of SODE diminishes when used on shallow trees (low index depth) because each lookup involves fewer resubmissions, which SODE expedites.



(a) Multiple threads with index depth 6 (b) Multiple threads with index depth 3 (c) Single thread with varying index depth

Fig. 6: Throughput of random key lookup operations when running with `read`, XRP, or SODE.



(a) Average Latency (b) Throughput

Fig. 7: Average read latency and throughput of BPF-KV with `read`, XRP, and SODE when performing a range query over a varying number of range lengths.

Fig. 5(d) shows that SODE also reduces the average latency of operations, with the impact becoming more significant as the tree depth increases.

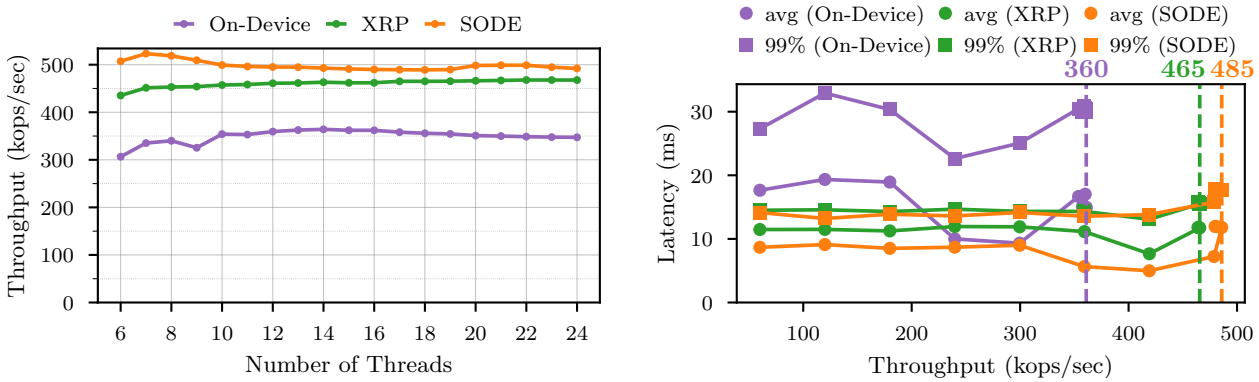
Throughput. Fig. 6(a) and Fig. 6(b) show the throughput of BPF-KV with two different index depths, 6 and 3, when running with `read`, XRP, or SODE. SODE's throughput remains 55.3–121.5% higher than that of `read` and 16.4–38.5% higher than XRP. Fig. 6(c) shows that the benefit of SODE remains significant for BPF-KV with the other index depths (up to 41%), except for depth 1, similar to the tail latency result. We also confirm that SODE meaningfully contributes to the performance improvements. Most I/O resubmissions are handled by the on-device path in when we run BPF-KV with one thread, but SODE utilizes both the in-kernel and on-device path. When running with 12 threads with index depth 6, SODE handles about 62% of resubmissions using the in-kernel path and uses on-device path for the remaining 38%.

Range Query. We evaluated the performance impact of SODE on range queries using a benchmark that first searches for the initial object and then traverses leaf nodes in a tree with an index depth of 6. Fig. 7 shows that SODE outperforms both XRP and `read` in range query performance. Fig. 7(a) demonstrates that the average latency of SODE is lower than that of `read` or XRP, and Fig. 7(b) shows that SODE's throughput remains higher than that of the other two. As with XRP, the size of the scratch page is 4 KB, and both XRP and SODE can fetch only up to 32 objects with a single resubmitting system call. Despite this limitation, SODE outperforms both `read` and XRP.

5.1.1 Open-Loop Load Generator

We also evaluate SODE on BPF-KV when it is heavily loaded with an open-loop load generator written for and used to evaluate XRP. The results show that SODE's on-device resubmission path improves both throughput and latency, and that this on-device path must be used in conjunction with the in-kernel path to achieve a performance advantage. Fig. 8(a) shows the throughput of SODE, XRP, and on-device with a varying number of BPF-KV threads, where on-device is a variant of SODE using the on-device resubmission path only. The throughput of SODE is about 10.5% higher than that of XRP, while that of the on-device variant remains lower. SODE's advantage in throughput comes from the use of both the host and on-device resubmission paths. When fully loaded and if the loads are balanced gracefully, SODE brings the on-device computing resources into the resource pool for executing resubmission tasks, increasing the throughput. As the on-device result shows, SODE can never achieve such an advantage in throughput if it relies only on on-device computing, whose computing power is fundamentally limited.

Fig. 8(b) further demonstrates the benefit of using two paths together by presenting the average and tail latency of three different resubmission mechanisms as the load changes. The on-device shows the highest tail latency with the lowest peak throughput. SODE's tail latency is similar to that of XRP because, in this workload, the worst-case latency is related to the time each request is queued for resubmission by the in-kernel path. The benefits of SODE here are in the peak throughput and average latency. Similar to the earlier result, SODE achieves a higher peak throughput due to additional computing resources and the latency advantage of its on-device resubmission path. These improvements enable faster



(a) Throughput with varying number of application threads

(b) Latency-throughput graph with 12 threads

Fig. 8: Throughput and latency of BFP-KV having a tree with index depth 6, when running with XRP, SODE, and the on-device variant of SODE when evaluated with an open-loop load generator.

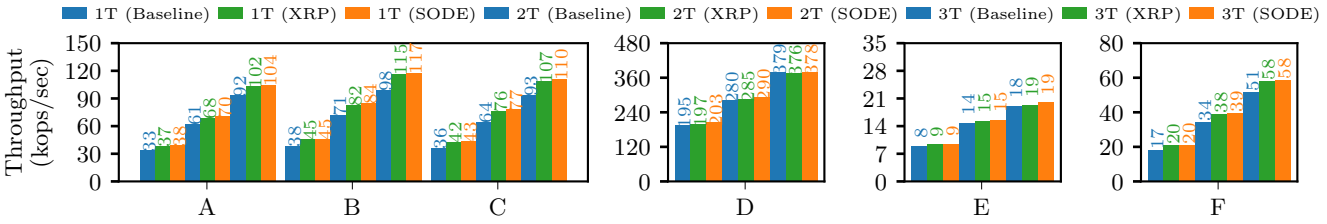


Fig. 9: Throughput of WiredTiger with YCSB [9] with 1, 2, and 3 threads (marked as 1T, 2T, and 3T) and 512 MB cache

handling of some requests, resulting in lower average latency.

5.2 WiredTiger

We use WiredTiger [57], a widely used Log-Structured Merge (LSM) tree-based persistent key-value store, to evaluate the performance impact of SODE on real-world applications. As the workload, we use the YCSB [9], which is also widely used to evaluate the performance of key-value stores. For this evaluation, we adapted WiredTiger to use a parallelized on-device resubmission task where it runs four eBPF filters in parallel, as described in §4.5. Specifically, we adapted the XRP's resubmission task by parallelizing the task and replaced the occurrences of `read_xrp` with `read_sode` and `read_sode_parallel`. We measured the performance of XRP using its adaptation and of unmodified WiredTiger that uses only the `read` system call. We note that our changes to adapt WiredTiger do not exhibit performance changes when running without SODE. Unless otherwise specified, we set the cache size to 512 MB, which is the default of WiredTiger, and used all combinations or workload mixes (*i.e.*, A, B, C, D, E, and F). We also use the default for other configuration parameters. A, B, C, and F issue 10 million operations each, D issues 50 million, and E issues 3 million.

Throughput. Fig. 9 shows that SODE improves the throughput of WiredTiger when running workloads from YCSB by 1.56–3.29% compared to XRP. Similar to the BPF-KV results, the on-device processors become an additional resource for SODE to execute the resubmission tasks, contributing to the throughput improvement. In addition, Most I/O resubmissions are handled by the on-device path when WiredTiger runs with one thread, but SODE uses both in-kernel (31%) and on-device (69%) paths otherwise. We obtained these numbers from the experiment with workload C running with 3 threads. We also confirmed that SODE's performance benefit remains when the cache size changes. The supplemental material contains the full experimental results from varying cache sizes.

Tail Latency. We measured the 99th-percentile tail latency of WiredTiger when running the six workloads with `read`, XRP, or SODE under a 20 kop/s fixed load per client thread (for A, B, C, D, and F) and under a 5 kop/s per client thread (for E), which are the default of YCSB. As Fig. 10 shows, SODE's tail latency is more or less the same as that of XRP, achieving the design goal. SODE outperforms XRP under some configurations, but the difference is hardly noticeable in the others. The latency reduction of SODE is less significant on WiredTiger when compared to BPF-KV because the read requests in the workload infrequently involve long resubmission chains. For example, 30% of resubmission requests

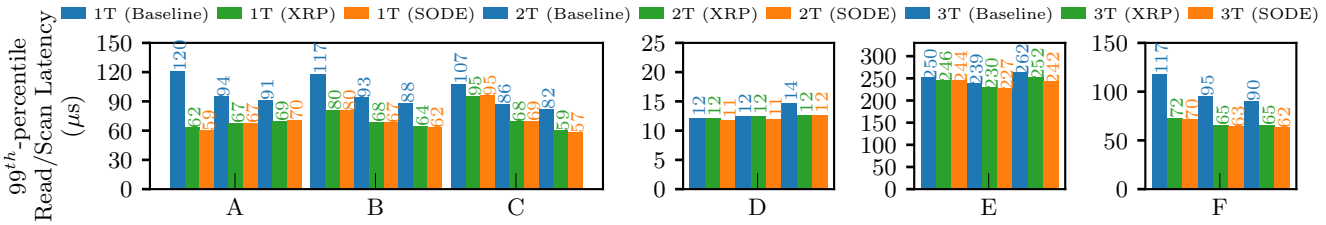


Fig. 10: 99th-percentile read/scan latency of WiredTiger with YCSB [9] with 1, 2, and 3 threads (marked as 1T, 2T, and 3T) and 512 MB cache

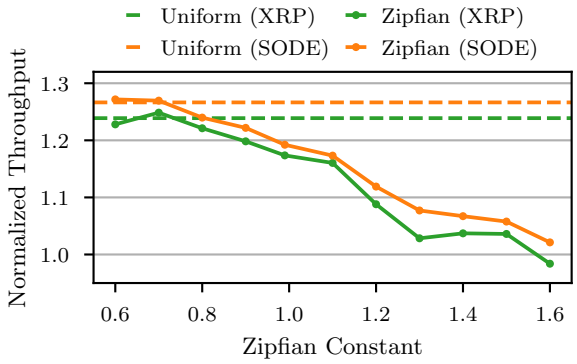


Fig. 11: The throughput of WiredTiger on YCSB C normalized to the throughput of unmodified WiredTiger using the `read` system call, across varying Zipfian constants and a uniform distribution

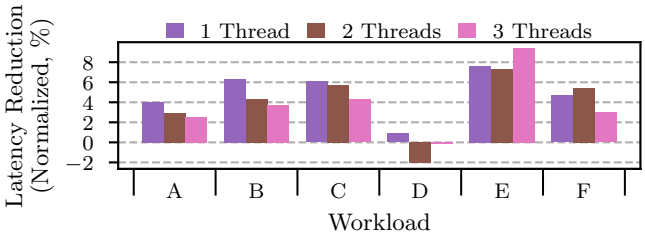


Fig. 12: Normalized latency reduction of SODE's resubmitting system calls compared to the XRP's. 6% on the workload C with 1 thread means that the average latency of SODE's system call is 6% lower than the latency of XRP's.

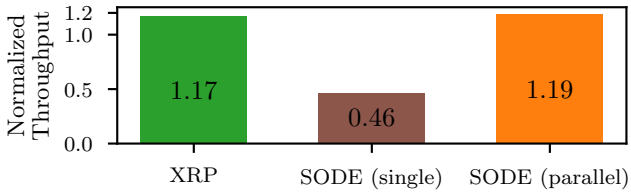


Fig. 13: Normalized throughput of WiredTiger on YCSB C with 0.99 Zipfian constant running with XRP, SODE (single), and SODE (parallel)

resubmit only twice, 22% resubmit three times, 23% resubmit four times, and 25% resubmit five times. As we noticed in the BPF-KV results (see Fig. 5(c)), the benefit of SODE diminishes when the resubmission chain becomes shorter.

Skewness. Fig. 11 demonstrates that SODE's advantage over the in-kernel resubmission path is consistent across all access distributions, with an improvement in throughput of 2–5%. Similar to XRP, the benefit of SODE diminishes as the Zipfian constant increases (*i.e.*, as the workload becomes more skewed). The diminishing benefit is due to hot entries being frequently served from the in-memory cache in skewed workloads. This reduces the number of operations benefiting from SODE's on-device or in-kernel resubmission, thus limiting its overall impact.

Impact on Resubmission Latency. The performance improvement by SODE stems from reducing resubmission latency compared to XRP. To further evaluate this improvement, we compared the latency of resubmitting system calls when running SODE and XRP. As Fig. 12 shows, SODE's resubmission system call latency is up to 9.4% lower than that of XRP, indicating that SODE performs the same operation more quickly. The only exception is workload D, where the on-disk tree depth is low, thus most resubmission requests involve a small number of resubmissions, limiting the benefit of both SODE and XRP. We confirmed this by measuring the contribution of resubmission operations to the application's `search` operation, which was adapted to use resubmitting system calls when available. The total latency of resubmission tasks constitutes 40–60% of the total `search` time for five out of the six workloads, except workload D. For workload D, resubmission contributes to only 3% of the search time.

Parallelism. Fig. 13 shows that enabling parallelized resubmission tasks is essential for improving throughput when running WiredTiger. We run WiredTiger with XRP, SODE (single), or SODE (parallel) using YCSB C as the workload to evaluate the contribution of parallelizing resubmission tasks to the throughput. SODE (single) uses the single-threaded resubmission tasks for on-device resubmission. Without parallelized resubmission tasks, the throughput when using SODE is about 54% lower that WiredTiger running with `read`. As

explained in §3.3 and in Table 1, the execution time of the resubmission tasks for WiredTiger is much longer than that of BPF-KV, and the latency increases by more than 2.5μs when it moves from the host to the device processor. This difference nullifies the potential latency benefit of on-device resubmission, resulting in the performance degradation shown in the figure. Parallelizing the resubmission task reduces its latency on the device from 4403 ns to 993 ns on average, enabling SODE to improve the throughput by 19% and 2% when compared to `read` and XRP, respectively.

6 Related work

Our study investigates the potential performance gains from moving resubmission tasks to processors on the storage device side, and SODE utilizes eBPF to represent and execute these tasks in a quarantined environment. This section discusses prior works on these topics, which are closely related to the motivation, use cases, and core components of SODE.

6.1 Near-Storage Computing

The concept and study of in-storage computing has a long history, predating the invention of SSDs [3, 22, 31, 60]. Although the idea was not widely adopted with hard disks, it has been revisited with SSDs as the storage medium [8, 11, 17, 30, 35, 64, 67]. The community has identified a variety of applications that can benefit from in-storage computing, such as database operations [7, 10, 26, 69], graph processing [28, 40, 50], file systems [4, 48, 73], and deep learning [34, 38, 43].

Studies also explore various processing elements suitable for in-storage computing, such as FPGAs [18, 61, 66], and application-specific integrated circuits (ASICs) [41, 42, 45, 49, 63]. The increasing popularity of in-storage computing has also motivated the consideration of programming models and abstractions [15, 72]. Among these, the studies most closely related to our work are λ-IO [72] and Delilah [18]. These two works present and solve the challenges of pushing computations from general-purpose processors to storage devices, using eBPF filters to write and execute the computations. However, neither of these studies is designed to push resubmission tasks that aim to capitalize on latency advantages. λ-IO aims to provide a unified programming model of I/O stack for applications by adaptively running tasks either above the virtual file system layer using the page cache or inside the storage device. Delilah proposes running eBPF filters on storage-side FPGAs. In contrast, SODE focuses on the execution of resubmission tasks within storage in collaboration with the in-kernel resubmission path [76]. In fact, SODE can also work with λ-IO's kernel runtime, which we consider for future work.

The prevalence of network-attached storage has also motivated in-storage computing techniques [36, 37, 75]. Splinter [37] targets pointer-chasing tasks, which are similar to resubmission tasks, as one of its primary use cases. Network-attached storage systems typically experience longer random access latencies, making near-storage computing an attractive method to expedite resubmission tasks. Consequently, the advantage of lower latency is considered a given, and Splinter focuses on sandboxing and load-balancing techniques. In contrast to Splinter, our study is motivated by the observation that the latency advantage is less obvious in local in-storage computing. Therefore, we focus on developing techniques to reveal and leverage this advantage.

6.2 Using eBPF to Lower User-defined Code

Many studies utilize eBPF to run user-defined code within a privileged context [6, 13, 14, 19, 36, 75–77]. Among these, XRP [76] and BPF-oF [75] are most closely related to SODE. XRP discusses resubmission tasks as a workload that can benefit from lower disk access latency. With the observation that the contribution of software stack latency becomes higher when the system employs a low-latency storage device, XRP proposed to run the task somewhere closer to the storage, right above the device driver. BPF-oF takes this approach further by moving the tasks even closer to the storage in the context of network-attached storage. Inspired by XRP, we explore the possibility of moving resubmission tasks closer to the storage. As explained in this paper, the latency advantage is less obvious when considering local storage connected over PCIe. To leverage this advantage, careful scheduling and parallelization are required, which distinguishes our study from those focusing on scenarios where storage is located over the network.

7 Conclusion

This paper presents our study about running resubmission tasks on wimpy processors close to the storage medium. To the best of our knowledge, we are the first to explore in depth the possibility of running resubmission tasks using the on-device processors to take advantage of the lower latency from their proximity to data. Execution on the side of storage devices brings latency benefits beyond the PCIe round-trip time. However, the limited performance of on-device processors makes it non-trivial to unleash their potential, calling for a mechanism to blend them with existing means to expedite resubmission tasks, the in-kernel path. To this end, this paper presents our design of SODE that selectively uses the on-device resubmission path along with the in-kernel resubmission path. Our experiments using two benchmarks traversing on-disk data structures show that SODE improves both throughput and latency of the benchmarks by carefully scheduling the resubmission tasks using these two resubmission paths, optimistically continuing to resubmit requests, and allowing parallelization of the on-device resubmission tasks.

References

[1] Seccomp bpf (secure computing with filters), 2023.

[2] ebpf documentation, 2024.

[3] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, page 81–91, New York, NY, USA, 1998. Association for Computing Machinery.

[4] Hanyeoreum Bae, Donghyun Gouk, Seungjun Lee, Jiseon Kim, Sungjoon Koh, Jie Zhang, and Myoungsoo Jung. Intelligent ssd firmware for zero-overhead journaling. *IEEE Computer Architecture Letters*, 22(1):25–28, 2023.

[5] Daniel Borkmann. Bpf and spectre: Mitigating transient execution attacks–daniel borkmann, isovalent, 2023.

[6] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on fpga nics. *Commun. ACM*, 65(8):92–100, jul 2022.

[7] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. POLARDB meets computational storage: Efficiently support analytical workloads in Cloud-Native relational database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 29–41, Santa Clara, CA, February 2020. USENIX Association.

[8] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 91–102, New York, NY, USA, 2013. Association for Computing Machinery.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1221–1230, New York, NY, USA, 2013. Association for Computing Machinery.

[11] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. Programmable solid-state storage in future cloud datacenters. *Commun. ACM*, 62(6):54–62, may 2019.

[12] Linux Documentation. numactl (8): Linux man page, 2021.

[13] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. Partition-aware packet steering using xdp and ebpf for improving application-level parallelism. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, ENCP '19, page 27–33, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.

[15] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, and Myoungsoo Jung. Dockerssd: Containerized in-storage processing and hardware acceleration for computational ssds. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 379–394, 2024.

[16] Brendan Gregg. *BPF performance tools*. Addison-Wesley Professional, 2019.

[17] 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. Biscuit: a framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 153–165, Seoul, Resublic of Korea, 2016. IEEE Press.

[18] Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangjin Lee, and Ken Friis Larsen. Delilah: Ebpf-offload on computational storage. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, DaMoN '23, page 70–76, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International*

Conference on Emerging Networking EXperiments and Technologies, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Uitaek Hong, Hwijoon Lim, Hyunho Yeo, Jinwoo Park, and Dongsu Han. Sand: A storage abstraction for video-based deep learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '23, page 16–23, New York, NY, USA, 2023. Association for Computing Machinery.

[21] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, page 73–86, USA, 2004. USENIX Association.

[23] Intel. Intel® optane™ ssd 9 series. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html, 2022. [Accessed 06-05-2024].

[24] Di Jin, Vaggelis Atlidakis, and Vasileios P. Kemerlis. EPF: Evil packet filter. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 735–751, Boston, MA, July 2023. USENIX Association.

[25] Di Jin, Alexander J. Gaidis, and Vasileios P. Kemerlis. BeeBox: Hardening BPF against transient execution attacks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 613–630, Philadelphia, PA, August 2024. USENIX Association.

[26] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proc. VLDB Endow.*, 9(12):924–935, aug 2016.

[27] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. Bluedbm: An appliance for big data analytics. *ACM SIGARCH Computer Architecture News*, 43(3S):1–13, 2015.

[28] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 411–424, 2018.

[29] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Yangwook Kang, Yang-suk Kee, Ethan L. Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2013.

[31] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, sep 1998.

[32] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. Empowering storage systems research with nvmevirt: A comprehensive nvme device emulator. *ACM Trans. Storage*, 19(4), oct 2023.

[33] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. NVMeVirt: A versatile software-defined virtual NVMe device. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 379–394, Santa Clara, CA, February 2023. USENIX Association.

[34] Shine Kim, Yunho Jin, Gina Sohn, Jonghyun Bae, Tae Jun Ham, and Jae W. Lee. Behemoth: A flash-centric training accelerator for extreme-scale DNNs. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 371–385. USENIX Association, February 2021.

[35] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 219–231, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. Safe and efficient remote application code execution on disaggregated nvm storage with ebpf, 2020.

[37] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-Metal extensions for Multi-Tenant Low-Latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, Carlsbad, CA, October 2018. USENIX Association.

[38] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. Hardware/Software Co-Programmable framework for computational SSDs to accelerate deep learning service on Large-Scale graphs. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 147–164, Santa Clara, CA, February 2022. USENIX Association.

[39] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019*, Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, pages 603–616, Renton, WA, 2019. USENIX Association.

[40] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H. Peter Hofstee, Gi-Joon Nam, Mark R. Nutter, and Damir Jamsek. Extrav: boosting graph processing near storage with a coherent accelerator. *Proc. VLDB Endow.*, 10(12):1706–1717, aug 2017.

[41] Yunjae Lee, Jinha Chung, and Minsoo Rhu. Smartsage: training large-scale graph neural networks using in-storage processing architectures. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 932–945, New York, NY, USA, 2022. Association for Computing Machinery.

[42] Siqi Li, Fengbin Tu, Liu Liu, Jilan Lin, Zheng Wang, Yangwook Kang, Yufei Ding, and Yuan Xie. Ecssd: Hardware/data layout co-designed in-storage-computing architecture for extreme classification. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[43] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for In-Storage data retrieval. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 395–410, Renton, WA, July 2019. USENIX Association.

[44] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. Unleashing unprivileged ebpf potential with dynamic sandboxing. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 42–48, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. Inspire: in-storage private information retrieval via protocol and architecture co-design. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 102–115, New York, NY, USA, 2022. Association for Computing Machinery.

[46] Linux. Memory protection keys. https://docs.kernel.org/core-api/protection-keys.html, 2024. [Accessed 06-05-2024].

[47] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. MOAT: Towards safe BPF kernel extension. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1153–1170, Philadelphia, PA, August 2024. USENIX Association.

[48] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, San Jose, CA, February 2013. USENIX Association.

[49] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, Rachata Ausavarungnirun, Nandita Vijaykumar, Mohammed Alser, and Onur Mutlu. Genstore: a high-performance in-storage processing system for genome sequence analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 635–654, New York, NY, USA, 2022. Association for Computing Machinery.

[50] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. Graphssd: graph semantics aware ssd. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 116–128, New York, NY, USA, 2019. Association for Computing Machinery.

[51] MITRE. Cve-2017-17856. https://nvd.nist.gov/vuln/detail/CVE-2017-17856, 2017. [Accessed 06-05-2024].

[52] MITRE. Cve-2021-31440. https://nvd.nist.gov/vuln/detail/CVE-2021-31440, 2021. [Accessed 06-05-2024].

[53] MITRE. Cve-2021-33200. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33200, 2021. [Accessed 06-05-2024].

[54] MITRE. Cve-2021-3490. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3490, 2021. [Accessed 06-05-2024].

[55] MITRE. Cve-2022-0264. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0264, 2022. [Accessed 06-05-2024].

[56] MITRE. Cve-2022-23222. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23222, 2022. [Accessed 06-05-2024].

[57] MongoDB. WiredTiger Storage Engine - MongoDB Manual v7.0 — mongodb.com. https://www.mongodb.com/docs/manual/core/wiredtiger/, 2024. [Accessed 08-05-2024].

[58] Neha Prakriya, Yu Yang, Baharan Mirzasoleiman, Cho-Jui Hsieh, and Jason Cong. Nessa: Near-storage data selection for accelerated machine learning training. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '23, page 8–15, New York, NY, USA, 2023. Association for Computing Machinery.

[59] Stephen M Brenner Quoc-Thai V Le, Jonathan Stern. Fast memcpy with spdk and intel® i/oat dma engine. https://www.intel.com/content/www/us/en/developer/articles/technical/fast-memcpy-using-spdk-and-ioat-dma-engine.html, 2017. [Accessed 18-05-2024].

[60] Erik Riedel1 and Garth Gibson. *Active Disks - Remote Execution for Network-attached Storage*. AD-a341 735. Carnegie-mellon univ pittsburgh pa school of computer Science, 1997.

[61] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage computing system for emerging High-Performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.

[62] Samsung. Samsung smartssd, 2023. [Accessed 08-05-2024].

[63] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with fpgas. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[64] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, Broomfield, CO, October 2014. USENIX Association.

[65] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. Leaftl: A learning-based flash translation layer for solid-state drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 442–456, New York, NY, USA, 2023. Association for Computing Machinery.

[66] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. Rm-ssd: In-storage computing for large-scale recommendation inference. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1056–1070, 2022.

[67] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: Towards Energy-Efficient, In-Situ data analytics on Extreme-Scale machines. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, San Jose, CA, February 2013. USENIX Association.

[68] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, dec 1993.

[69] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: an intelligent storage engine with support for advanced sql offloading. *Proc. VLDB Endow.*, 7(11):963–974, jul 2014.

[70] AMD Xilinx. Zcu102 evaluation board user guide, 2023.

[71] Sujay Yadalam, Chloe Alverti, Vasileios Karakostas, Jayneel Gandhi, and Michael Swift. 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*, ASPLOS '24, page 35–51, New York, NY, USA, 2024. Association for Computing Machinery.

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

[73] Zhe Yang, Youyou Lu, Erci Xu, and Jiwu Shu. Coinpurse: A device-assisted file system with dual interfaces.

In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[74] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, Hong Kong, China, 2017. IEEE Press.

[75] Ioannis Zarkadas, Tal Zussman, Jeremy Carin, Sheng Jiang, Yuhong Zhong, Jonas Pfefferle, Hubertus Franke, Junfeng Yang, Kostis Kaffes, Ryan Stutsman, and Asaf Cidon. Bpf-of: Storage function pushdown over the network, 2023.

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

[77] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. Bpf for storage: an exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 128–135, New York, NY, USA, 2021. Association for Computing Machinery.